Guardrail: High Fidelity Correctness Checking of Device Drivers for Safeguarding I/O Operations

OLATUNJI RUWASE* **PHILLIP B. GIBBONS†** **MICHAEL A. KOZUCH†**
TODD C. MOWRY*

December 2012
CMU-CS-12-149

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
†Intel Labs, Pittsburgh, PA, USA

Keywords: dynamic analysis, virtualization, reliability

Abstract

Device drivers are an Achilles' heel of modern commodity operating systems, accounting for far too many system failures. Previous work on *driver reliability* has focused on protecting the kernel from unsafe driver side-effects by interposing an invariant-checking layer at the driver interface, but otherwise treating the driver as a black box. In this paper, we propose and evaluate Guardrail, which is a more powerful framework for run-time driver analysis that performs *decoupled*, *instruction-grain* dynamic correctness checking on arbitrary kernel-mode drivers as they execute, thereby enabling the system to detect and mitigate more challenging correctness bugs (e.g., data races, uninitialized memory accesses) that cannot be detected by today's fault isolation techniques. Our implementation of Guardrail demonstrates that it can find serious data races, memory faults, and DMA faults in native Linux drivers that required fixes, including previously unknown bugs. Also, we show that with hardware logging support, Guardrail can be used for online protection of persistent device state from defective drivers with minimal impact on the end-to-end performance of standard I/O workloads.

1 Introduction

Device drivers have received significant attention in recent years [3, 20, 54, 5, 16, 6, 50, 58, 17, 15, 28, 22, 33, 42, 47, 27] because they are critical pieces of system software that account for roughly 70% of the Linux code base [9, 33] and cause a large fraction of system crashes [42, 19, 49, 33]. While researchers have explored a variety of different strategies for improving device driver robustness, including *static analysis* [9, 3, 22, 33], *specification* [28, 43, 54], *type safety* [38, 47, 58], and *user-level drivers* [5, 20, 27, 54], our focus in this paper is on performing sophisticated *run-time analysis* of driver software to detect and mitigate the impact of bugs. Run-time analysis of driver software complements the other strategies mentioned above since it can potentially catch problems that the other techniques may miss due to practical limitations.

The main focus of run-time driver analysis to date has been on *fault isolation* [20, 54, 5, 16, 6, 50], where the goal is to augment the driver interfaces to prevent a buggy driver from corrupting the OS kernel. The basic idea behind fault isolation is to interpose a run-time checking layer at the driver interface that performs a sanity check before the driver is allowed to proceed with performing side effects outside of the driver (e.g., writing to kernel memory [50]).

1.1 Limitations of Existing Fault Isolation Techniques

While existing fault isolation techniques improve device driver reliability relative to systems without fault isolation, their effectiveness is still limited by the fundamental property that they *only check invariants at the driver interfaces*, and they treat the bulk of the driver's execution as a *black box*. For example, most fault isolation techniques ignore driver *reads* (since normal reads do not have side-effects), which means that they are unable to recognize problems such as *data races* within drivers. In other words, existing fault isolation techniques do not focus on whether the driver software is executing correctly (at a fundamental level), but rather on whether the driver has obviously harmful side-effects beyond its interface.

While fault isolation research has focused on the driver's interface with the kernel, arguably the driver's interface to its *hardware device* is equally important (if not more important) since rebooting the kernel may do little good once persistent device state has been corrupted. In contrast with the driver/kernel interface, which tends to be relatively uniform across drivers, the driver/device interfaces are far more diverse and device-specific, which makes it far more challenging to interpose and successfully check invariants across this latter interface [54].

1.2 Our Approach: Decoupled Dynamic Instruction-Grain Driver Analysis

Rather than treating the bulk of the driver execution as a black box, we propose a more powerful framework, called *Guardrail*, where the interposition layer's decision of whether to allow the driver to proceed with a side-effect-causing operation is driven not simply by invariant checks at the driver's interface, but rather by instruction-grain dynamic analysis of the driver software as it executes, as illustrated in Figure 1. Indeed, Guardrail typically identifies correctness problems within the driver before they even reach the driver's interface. Thus, we enable a more comprehensive analysis of whether the driver software is behaving *correctly* or not than what is practical

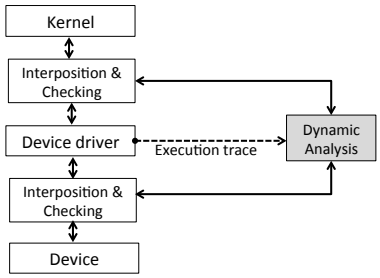

Figure 1: Incorporating dynamic analysis to protect the system from driver faults.

today by simply monitoring the driver's interfaces. For example, a driver that contained either a data race or a memory bug might store the wrong value in a legitimate target location in either kernel memory or its device.

To achieve this higher fidelity of dynamic correctness checking without sacrificing driver performance, we propose a *decoupled* approach to performing the dynamic instruction-by-instruction analysis of the driver as it executes. In our decoupled approach, an execution trace of the driver software is captured (e.g., via a hardware-assisted logging mechanism [7, 51] or through binary instrumentation [34, 17]) and stored in a buffer that is consumed asynchronously by the dynamic analysis tool which runs concurrently on a separate thread in its own VM. Because the dynamic analysis tool can lag behind the driver in our decoupled approach, the interposition layer stalls any side-effect-causing operations at the driver interface until the dynamic analysis is able to catch up.

Guardrail effectively achieves a "sweet spot" between *synchronous* instruction-grain analysis (which results in too large of a performance overhead for latency-critical driver operations such as interrupt handling) and *offline* (or post-mortem) instruction-grain analysis (which avoids runtime overhead but occurs too late to prevent faulty drivers from corrupting persistent state).

We present three example analysis tools for Guardrail: a memory checker, a data race checker, and a DMA checker. While our memory checker, *DMCheck*, is fairly straightforward, our race detector, *DRCheck*, requires a number of new techniques to address the concurrency complexities of kernel-mode drivers, and our DMA checker, *DMACheck*, is to our knowledge the first analysis tool for detecting DMA errors.

Related Work

As described already, our work complements earlier research on *fault isolation* [20, 54, 5, 16, 6, 50] by not only using interpositioning to prevent harmful side-effects from escaping from the driver, but also by "opening the black box": i.e. using instruction-by-instruction dynamic analysis of the driver software to hopefully identify problems that are not obvious to interface invariant checks. Also, in contrast with the previous proposal for isolating *devices* from driver faults [54], which required modifying the driver and moving it into user-space, our approach is transparent to both the driver and the device, and therefore our approach works with arbitrary driver binaries and devices.

Regarding dynamic checking for faults *within* drivers, Safe-Drive [58] and KAddrcheck [17] perform run-time checks to detect memory addressibility issues in kernel code, including drivers. In contrast with SafeDrive [58], which instruments drivers at compile-time, our approach works directly on binaries and does not require access to driver source code. In contrast with KAd-

drcheck [17], our approach uses *decoupled* analysis to reduce the impact on driver performance and can detect problems with memory *initialization* (in addition to addressability). Moreover, within the same Guardrail framework, a wide variety of tools are readily supported.

Finally, DataCollider [15] detects data races through a sampling-based approach by stalling kernel threads in critical sections and using data breakpoints to detect conflicting accesses in other threads. There are three fundamental differences between DataCollider and our *DRCheck* tool (which we describe in detail later in this paper). First, DataCollider can only detect whether a data race occurred in a specific observed interleaving, whereas *DRCheck* can detect race conditions that might occur in other interleavings (since *DRCheck* models synchronization protocols used in the driver). Second, DataCollider uses *sampling* to reduce run-time overheads, whereas *DRCheck* uses decoupled analysis to reduce overhead while still checking all driver invocations for potentially harmful behavior. Third, DataCollider's stalling approach is not suited for threads servicing time-critical interrupts, making it less effective for drivers, which are frequently in interrupt contexts, than *DRCheck*.

1.3 Contributions

This paper makes the following contributions:

- We propose and implement a novel framework, *Guardrail*, for detecting incorrect driver behavior at run-time and preventing the faulty driver from corrupting the rest of the system (including persistent state on hardware devices). In contrast to previous proposals, Guardrail performs instruction-grain correctness checking as the driver executes, and uses a decoupled VM-based approach to provide isolation and minimize the impact on driver performance. Guardrail supports arbitrary driver binaries and devices for kernel-mode drivers in commodity operating systems.

- Within our framework, we demonstrate instruction-grain correctness checking tools that detect unsafe use of uninitialized data, data races, and DMA faults (none of which is supported by existing driver fault isolation techniques). Our data race tool improves upon prior approaches by minimizing false positives and avoiding false negatives, while handling the complexities of kernel-mode drivers.

- Our experimental results demonstrate that our correctness checking tools are more effective at catching driver bugs than previous tools, e.g., finding a bug in the popular *qla2xxx* SCSI driver that had eluded detection for years. Moreover, our results show that Guardrail, and the proposed correctness checkers have negligible performance impact over the base Xen system for most of the I/O workloads, given hardware logging support.

2 System Design

To foster a principled approach while designing Guardrail, we developed a set of high-level design goals. In particular, Guardrail should:

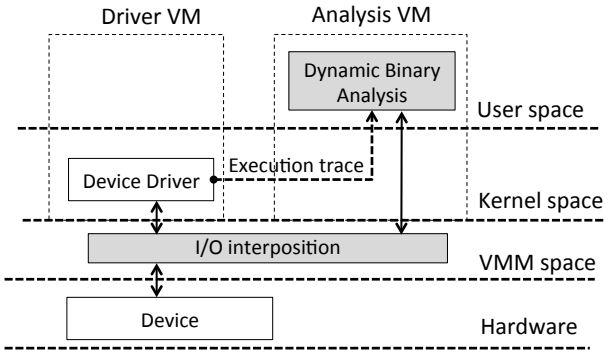

Figure 2: Decoupled driver analysis for protecting I/O devices from driver faults.

(generality) support the monitoring of unmodified driver binaries running in common computing environments (e.g. stock multithreaded OS, arbitrary applications and runtimes, etc);

(detection fidelity) enable fine-grain correctness-checking and identification of errors, while supporting a wide variety of monitoring tools;

(containment) provide mechanisms capable of preventing detected driver errors from erroneously affecting external state;

(response flexibility) allow users to control what Guardrail does on detecting an error (e.g., disable I/O operations from the driver, or simply record information for post-mortem analysis); and

(trustworthiness) rely on a minimal trusted computing base for containment.

The system architecture that resulted from these goals is shown in Figure 2. To simultaneously satisfy the *containment* and *generality* goals, we adopted a virtual machine-based system. The driver(s) of interest, along with the stock OS (Linux, in our prototype) and related applications, executes in one virtual machine (VM), labeled the "Driver VM" in the figure. The virtual machine monitor (VMM) provides the interposition mechanism. I/O operations are intercepted in this layer, and should an error be detected, the VMM prevents the error from propogating outside the driver VM by simply not delivering it to the physical hardware.

While the driver executes, a trace of its operations is collected and delivered to the "Analysis VM." An instruction-level trace supporting high *detection fidelity* can be captured through one of several mechanisms: binary translation [34, 17] in the driver VM, VMM-based monitoring [56, 10], or monitoring hardware [7, 52, 51]. Because driver code is potentially executed by an unbounded number of kernel threads, logical logs are maintained per virtual processor, rather than per kernel thread, in the "Driver VM", to avoid scalability issues.

The execution trace is streamed, possibly with some buffering delay, to the Dynamic Binary Analysis tool, which runs in user space in the Analysis VM. This tool consumes the execution trace and checks for driver errors, such as data races or memory access violations, to help the VMM determine when (or if) an intercepted I/O operation can be safely dispatched to the device. If a fault

is identified in the driver's execution then it is potentially unsafe to dispatch the intercepted I/O operation to the device. However, the appropriate course of action in this situation often depends on the peculiar requirements of the user (e.g., willing to sacrifice system availability to ensure persistent data integrity). Therefore, to accomodate the variety of constraints in production sites, end users have the *response flexibility* of configuring Guardrail to operate in one of 3 modes: (i) *stringent*, (ii) *permissive*, and (iii) *triage*. In *stringent* mode, Guardrail blocks the intercepted and subsequent I/O operations from the driver, effectively disabling the I/O device. *Permissive* mode is the other extreme, where after performing user specified actions (e.g., alerting the user, taking a system checkpoint, enabling more detailed analysis etc.) Guardrail dispatches the I/O operation to the device and resumes normal execution. Moreover, *permissive* Guardrail records information to enable post-mortem analysis of a resulting system failures. *Triage* mode represents a middle ground between these two extremes, where Guardrail performs a best-effort estimation of the safety of completing the I/O operation by automatically triaging the fault [29, 23]. If the I/O operation is deemed safe, Guardrail behaves like *permissive* mode, otherwise it behaves like *stringent* mode. Although this flexibility allows Guardrail to be configured in interesting ways for different real-world deployment scenarios, our current work is however focused on *stringent* Guardrail.[1]

Note that in this design, the *trustworthiness* of the containment mechanism is maintained because any complexity associated with tracking the driver state, emulating device-specific logic, or correctness checking is managed in the dynamic analysis tool. Consequently, device-independent I/O interpositioning may be effected through a simple addition to the VMM layer; less than 500 lines of C code were required to retrofit a commodity VMM (Xen [4]) with I/O interpositioning. The complexity of the checking tool may be non-trivial, however, primarily because the system was designed to accomodate arbitrary correctness-checking to cope with the wide variety of bug types that plague device drivers [9, 19, 33]. Fortunately, these tools run in user space of the analysis VM– easing their development and deployment.

2.1 Analysis Scope

An important question that arises in our design is: which events should be captured in the execution trace? For example, the trace could capture all instructions events in the driver VM, all kernel-level events only, or solely events associated with the driver. Naturally, capturing a larger set of events than necessary incurs a performance overhead, so ideally, the driver analysis tool would only need to process events generated by the driver. In our case, this would mean instructions whose addresses belong to the loaded driver module.

However, we soon observed that many operations critical to determining whether a driver is behaving correctly are in fact performed outside the driver. In particular, the I/O subsystem (or protocol stack) (e.g. network, SCSI, sound), which manages the driver, provides certain invariants upon which the driver writer may rely. For example, the network stack will acquire certain

[1] Permissive and Triage modes only affect Guardrail's response to suspected driver correctness issues *within* the context of the driver VM. The interposition layer always enforces the virtual machine definition. For example, an attempt to read/write past the end of a virtual disk will be strictly enforced under all modes.

```
HARD_TX_LOCK(dev, cpu);
 . . .
 rc = dev->hard_start_xmit(nskb, dev);
. . .
HARD_TX_UNLOCK(dev);
```

Figure 3: The Linux interface to network drivers serializes packet transmission by locking invocations of `hard_start_xmit()`.

locks prior to driver execution to protect shared data accesses within the driver, as illustrated by the code snippet from Linux 2.6.18 in Figure 3. Here, the network stack serializes packet transmission by locking the execution of the driver's `hard_start_xmit()` callback. A race detector focused solely on the driver's execution would not observe the lock acquire, which happens outside the driver context, and hence would incorrectly flag as data races all pairs of accesses in `hard_start_xmit()` by different threads with at least one writer.

In order to address this issue, Guardrail monitors and analyzes operations occurring in the relevant portions of the I/O subsystem (e.g., the *scsi_mod* module in the Linux SCSI subsystem) as well as those originating in the driver, itself. Extending the scope to include this interface captured all such "critical" operations that we observed. Our goal is not to determine whether there are errors in the interface, but rather to detect operations that are critical to driver correctness, and indeed this extension was useful for both our memory fault and data race detectors. A possible drawback of our approach is that interface changes across kernel versions will require corresponding modifications to our checking tools—such changes are likely infrequent because they often require corresponding modifications to the entire driver code base, not just to our tools.

2.2 Analysis Scheduling

Decoupled correctness checking requires careful scheduling of the analysis thread(s) to balance two conflicting performance goals: (i) minimizing the delay of consuming log entries, and (ii) minimizing the physical CPU utilization when the log(s) are empty. While log consumption delay could be minimized by polling the log(s), it leads to an undesirable waste of physical CPU cycles, because most drivers account for only a small fraction of the system-wide (i.e., "Driver VM") execution. Guardrail addresses this issue as follows. First, analysis thread(s) are put to "sleep" when execution leaves the "analysis scope" (Section 2.1) in the "Driver VM", and the log(s) become empty (saving CPU resources). Conversely, as execution returns to the "analysis scope", leading to log production, Guardrail employs inter-processor interrupts to wake the corresponding analysis thread(s) in the "Analysis VM" (reducing log consumption delay).

2.3 I/O Interposition Details

Since devices are controlled by reading/writing device registers, the interposition layer prevents driver errors from propogating beyond the VM boundary by: (i) by intercepting all[2] device register

[2]Some performance improvements could be obtained by not intercepting I/O operations that do not affect externally-visible state, such as side-effect free reads, but such optimizations would require scrutiny of the opera-

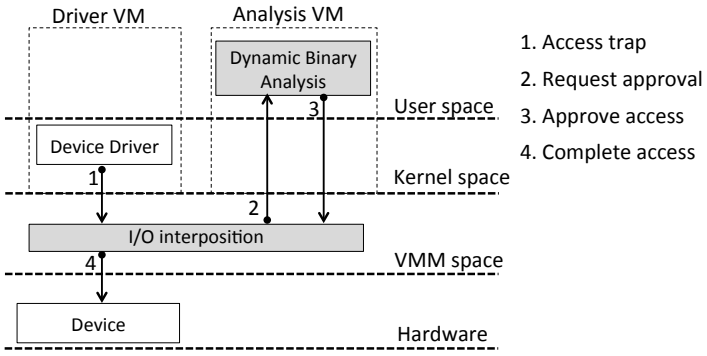

Figure 4: Transparent mediation of device register access.

accesses, (ii) coordinating with a decoupled correctness checker to determine the safety of the accesses, and (iii) ensuring their timely eventual completion as soon as they are deemed safe. Because the device is directly assigned to the driver VM [55], the interposition layer is transparent to both the driver and device, and therefore supports arbitrary drivers and devices. Figure 4 depicts the steps associated with how the interposition layer transparently handles a device register access.

Intercepting device register access Device register accesses from the driver are intercepted by ensuring that device register accesses from the driver VM fault to the VMM. In virtualized x86 environments, the I/O port address space is typically considered to be privileged by default and accesses to this space will fault. Many modern devices, however, are managed through memory-mapped I/O registers that are accessed through regular load and store instructions. Because these operations are subject to the usual address translation mechanisms, we can intercept accesses to the device registers by configuring the page tables of the driver VM such that these accesses fault to the VMM. The page faults resulting from this interposition can be distinguished from normal memory management page faults based on the faulting address. Note that interposition only affects communication originating from the driver VM; interrupts *to* the driver VM may be delivered normally.

Coordinating with decoupled correctness checking To limit the performance penalty of I/O interposition, intercepted device accesses should be verified and re-issued as soon as possible. If correctness checking is coupled with I/O interposition [54], this can be relatively straightforward; however, in our decoupled checking approach, additional coordination is required between the interpostion and checking components. After intercepting a device register access, the interposition layer uses a memory-based communication channel to request approval from the checker to complete the access. Details of the faulting instruction (e.g. thread id, faulting address) are included in the request. If the checker verifies that no errors occured in the execution trace up to the point where the access was encountered, the access will be approved. Otherwise, if the access is disapproved because of a driver fault, the interposition layer can initiate recovery using appropriate techniques [48, 6, 26].

tions and were not pursued in this work.

Because the checker's response will typically incur some latency, the interposition layer has at least two options regarding what what to do while waiting for the checker's response. The first is to hold the request in the hypervisor until the response arrives, effectively freezing the virtual CPU. To maintain the responsiveness of the guest OS, if interrupts are generated during this period, they should be delivered to the virtual CPU at the point just before the faulting instruction.[3] For development expediency, we selected a different option: the interposition layer simply returns control to the faulting instruction periodically. In other words, a guest OS thread that accesses device registers will continue executing the access and trapping into the interposition layer, until either the checker verifies the safety of the access or the thread is preempted.

Completing device register access After the checking tool has verified that the intercepted register access is safe, there are two ways of issuing the operation: (i) retry the faulting instruction after temporarily making the device register available to the guest OS [12], and (ii) emulate the faulting instruction in the hypervisor. Since the concurrently executing kernel threads, of commodity OSes, share a single kernel address space, ensuring that the temporarily accessible page is only accessed by the verified operation in the intended thread at an appropriate time, the first option requires great care. Consequently, in our current impolementation, we've chosen the emulation option to avoid potential containment errors, especially in SMP environments.

3 Driver Correctness Tools

Guardrail enables a wide range of driver correctness checking tools. In this work, we focus on tools for memory safety and concurrency, and OS protocol issues, because studies have shown that these account for a significant fraction of production driver faults [9, 19, 33, 42]. This section describes the three instruction-grain dynamic analysis tools that we developed for finding memory faults, data races and violations of OS rules for using DMA in unmodified Linux driver binaries.

3.1 DMCheck: Detecting Memory Faults

Kernel-mode drivers for commodity OSes are prone to type safety issues because they are written in unsafe languages (C and C++). Common memory faults in drivers include accesses to unallocated memory, unsafe use of uninitialized data, and memory leaks. The objective of our analysis is to detect such faults in driver executions. To this end, we adapt the analysis in *Memcheck* [30], a popular tool for finding memory faults in *application* binaries, to kernel-mode drivers. Specifically, we use *Memcheck*'s algorithm for finding memory faults by maintaining *metadata* for each byte of memory indicating whether the byte is currently allocated and, if so, whether it has been initialized. The metadata is updated in response to instructions that initialize data or system calls that allocate or free memory. An error is reported if an instruction accesses unallocated memory or uses uninitialized data in an unsafe way, or a memory leak is detected.

[3] We assume that the faulting instruction will eventually be re-executed, and the matching approval from the checker can then be applied. The VMM may need to monitor the guest to ensure it doesn't make an adjustment to prevent such re-execution (e.g. re-writing the stack). Such adjustments were not encountered in our experiments.

Our tool, *DMCheck*, adapts *Memcheck* to kernel-mode drivers by addressing two issues: (i) recognizing kernel functions for (de)allocating memory, and (ii) dealing with memory objects that are (de)allocated outside the driver. The first issue is trivially handled by recognizing that kernel memory management functions such as `kmalloc()` and `kfree()` are analogous to user-space functions such as `malloc()` and `free()`.

The second issue arises because of the need for drivers to communicate with the kernel in an efficient manner. Sometimes, this means the driver will manipulate memory objects that are allocated by other parts of the kernel. An example can be found in how socket buffers, for storing network packets, are handled in the network stack. The packet transmission path of a network driver receives socket buffers from the network stack and deallocates them after transmission. Conversely, the packet reception path allocates socket buffers, for received packets, and expects the network stack to deallocate them. DMCheck addresses this issue by incorporating the kernel-driver interface module into our analysis, as described in Section 2.1, so that the address range for each such memory object can be captured by the analysis.[4]

3.2 DRCheck: Detecting Data Races

Our second dynamic analysis tool, *DRCheck*, detects data races in kernel-mode drivers. A *data race condition* occurs whenever there are two unserialized accesses to the same shared data with at least one being a write. Race conditions are difficult to avoid during driver development because of the complex concurrency setting in which drivers operate, and difficult to find during pre-release testing because of their non-deterministic nature. Moreover, most drivers are developed by third parties who are not kernel experts [19, 33]. As modern OS kernels and their drivers increasingly exploit parallelism to improve performance, avoiding race conditions becomes all the more challenging, posing a serious threat to system stability.

A recent, related effort, DataCollider [15], used a sampling approach to investigate these issues in kernel space. This tool purposely stalls kernel threads and detects synchronization errors by observing "collisions" between the stalled thread and improperly synchronized threads. A thread "collides" with a purposely stalled thread only if there is nothing preventing them from colliding—the tool need not reason about the particular mechanisms used to serialize threads. However, because such stalling is not suited for threads servicing time-critical interrupts, DataCollider provides only limited coverage of interrupt contexts. This makes DataCollider less effective for drivers, because interrupt contexts represent significant portions of driver executions.

Our approach, *DRCheck*, in contrast, covers interrupt contexts as well as all other contexts. Furthermore, DRCheck can detect not just realized race conditions but also some potential race conditions that may occur in thread execution interleavings other than the one(s) observed.

Our tool is based on the LockSet algorithm explored in Eraser [44]. Unfortunately, this prior work may not be employed directly because it, and related studies [57, 18, 45], focused only on user-space applications, and the concurrency issues of kernel-mode driver execution are signifi-

[4] As in prior work, we trust the kernel-driver interface module. E.g., we assume that pointer and size arguments passed to the driver correspond to a properly allocated memory object for the given address range. The design can be readily extended to correctness check the kernel, but this is beyond the paper's driver-checking scope.

cantly more complex than user-mode execution. In particular, we have identified the following four sources of additional complexity that must be addressed in kernel-mode driver execution:

- concurrent execution of multiple priority levels, so that a thread may race even with itself;
- *ad hoc* mutual exclusion techniques that avoid lock overheads, such as disabling interrupts and preemption;
- deferred execution using softirqs and kernel timers; and
- synchronization invariants based on the context of the driver state.

These issues can lead to excessive false positives and false negatives using existing tools. In this section, we discuss the issues in further detail and describe the modifications to LockSet required to enable *DRCheck* to handle driver execution while minimizing false positives and avoiding false negatives.

3.2.1 Detecting driver concurrency

The most basic source of driver concurrency is multi-threaded execution of driver code and accessing of shared driver data. However, kernel preemption also introduces driver concurrency in a subtle manner using a single thread, as follows. Drivers normally perform tasks of varying importance (priority). In a network driver, for example, servicing an interrupt generated by the network card is of higher priority than responding to user-level requests for network card statistics. However, when kernel preemption switches a thread from a lower to a higher priority task that happens to share driver data, the thread may race with itself due to the interleaving of the two tasks.

Our approach is to exploit thread context to detect concurrency in drivers, by tracking both the identifier and context of kernel threads. In Linux, for example, a kernel thread at any point in time is either in *process* or *interrupt* (*bottom* or *top* half) *context*, which reflects the priority of its current task. (Similar contexts are used for kernel threads in other commodity OS kernels, like Windows.) Basically, just as memory operations of a user-mode thread are considered serialized, we consider the memory accesses of a kernel thread *in a particular context* to be serialized. In other words, except when explicitly synchronized by any of the methods discussed in this section, a kernel thread's memory access in one context is considered to be concurrent with its memory accesses from a different context (as well as memory accesses by other kernel threads).

3.2.2 Detecting mutual exclusion primitives

The kernel provides a variety of synchronization primitives for mutual exclusion: (i) locking primitives such as spinlocks and mutexes, (ii) operations that disable interrupts and preemption, and (iii) hardware atomic instructions such as `test_and_set`. Detecting (and tracking) locking primitives such as spinlocks and mutexes is easy because of their modularized interface (e.g., `spin_lock()/spin_unlock()`). Interrupt enabling/disabling can be detected (and tracked) by observing the specific instructions (e.g., `STI`, `CLI`, `POPF`, `IRET`, etc. in x86) in the execution trace. Hardware atomic instructions like `test_and_set` are more challenging because of

the need to determine whether the instruction guards a critical section and, if so, whether or not it succeeded in entering. *DRCheck* uses pattern matching over a small window of the trace starting with the `test_and_set` instruction (`btsl` in x86) in order to determine whether the sequence matches a known critical section preamble for the specific kernel. If so, it checks the value returned by the `test_and_set` to determine whether it succeeded.

3.2.3 Handling deferred execution

Kernel threads that execute under tight deadlines (e.g., interrupt service routines) are often responsible for important tasks (e.g., copying received packets from the network card) that cannot be completed in a timely manner. Thus, most OS kernels provide mechanisms for postponing work until a more convenient time, such as *softirqs* in Linux, *deferred procedure calls (DPCs)* in Windows, and *software interrupts* in Solaris. *Kernel timers* are also provided for deferring the execution of a function until at least a specified time in future. Common uses of timers include checking that tasks are completed on schedule, or that a device is still functional.

Softirqs are commonly used by interrupt handlers of high performance drivers to defer work to a future context, e.g., to the *bottom half* context. However, the way the interrupt thread deferring work synchronizes with the polling thread that will do the work poses a challenge for data race analysis because these threads do not share any locks. Instead, the interrupt thread enqueues the work, and then calls `raise_softirq` to asynchronously activate the polling thread. The Linux softirq infrastructure guarantees that only one polling thread, on the same processor as the interrupt thread, responds to the call and completes the deferred work. *DRCheck* recognizes the `raise_softirq` call as the serializing operation between the threads.

Kernel timers also pose some challenges to data race detection. For example, although a delay is specified when registering a timer, only the operations that were performed by the thread prior to timer registeration are guaranteed to be serialized with execution (possibly by a different thread) of the deferred function. This is because the thread could be preempted for a period longer than the timer delay. Also, successive executions of the function of a timer are serialized, even though synchronization primitives (e.g., locks) are not used in the function. On the other hand, executions of functions with different timers are not serialized.

These issues with kernel timers are handled in *DRCheck* as follows. First, we associate a virtual state with each timer. A timer is *inactive* before its registeration, and *active* until it executes, after which it becomes *inactive* again. This serializes the execution of the timer to operations preceding its registration. Next, we associate a virtual lock with each timer that is held throughout the execution of the timer function. This serializes the successive executions of the function of the timer.

3.2.4 Tracking state-based synchronizations

Many peripheral devices—e.g., ethernet, scsi, usb, etc.—behave like finite state machines (FSM), and drivers often use their states to protect critical sections. The set of valid operations for a device depends on the state of the device, and so the kernel, in order to prevent device failures, invokes only driver callbacks that are valid for the current device state. In other words, device states act

as the invariants that guard the invocation of certain driver callbacks by the kernel. Thus, any pair of driver callbacks that are never concurrently valid (i.e., they have conflicting invariants) will not execute concurrently, and their critical sections are mutually serialized as a result. For example, consider the FSM snippet in Figure 5 for a Linux network device. It shows that the `pci::probe` and `netdev::open` callbacks of a network driver are valid in different device states, and hence cannot race with each other. Existing race detection tools are oblivious to the invariants (or states) in which driver callbacks are executed, and hence they can incorrectly report races between callbacks with conflicting invariants. Indeed, our experimental study in Section 4 shows that ignoring state-based synchronization results in a high false positive rate. (As noted earlier, DataCollider is an exception to this false positives problem because it manifests only actual races.)

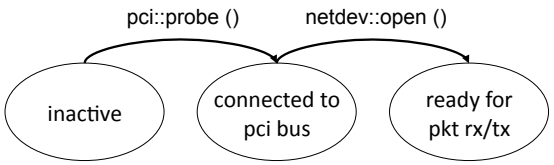

Figure 5: State transitions for a Linux PCI network device, showing that the `probe()` and `open()` functions of the driver are serialized.

So far, our discussion on state-based synchronization has focused on device states that are used by the kernel to control driver execution. Some examples include status of the PCI connection, interrupt request line (IRQL), polling/interrupt handling, etc. However, it is possible for a driver to use other state information internally to manage critical sections. Nevertheless, our focus is on kernel-aware device states, because most OS kernels organize devices into classes (e.g., network, scsi, graphics, usb) and export a standard interface to the drivers of a given class. It is therefore more scalable to design for the kernel interface than for individual drivers.

DRCheck incorporates kernel-aware states that control driver execution by tracking, based on the execution trace, the set of states under which each callback is invoked. Alternatively, one could use specifications obtained from kernel experts [14, 21], perhaps incurring less runtime overhead. We choose our approach because it does not rely on specifications being both correct and representative of the kernel code.

Because drivers routinely change device states, the basic approach of tracking states at driver entry points is not sufficient: Other regions of a callback might execute under a different set of states. As a refinement, *DRCheck* also tracks device states at code points that follow device state changes.

3.2.5 DRCheck Implementation

DRCheck is an extension of the *Lockset* algorithm in Eraser [44]. *Lockset* detects races in multithreaded applications by checking that shared data access is protected by a consistent locking discipline. *Lockset* maintains metadata for each word of shared memory indicating whether the location has been accessed by multiple threads, and if so, the set of locks consistently held by all

threads accessing the location from that point on. If there is no such common lock, *Lockset* reports a potential data race.

DRCheck extends *Lockset* as follows. First, adapting *Lockset* for kernel-mode locking primitives was straightforward for the ones that behave similarly to user-mode primitives (e.g., kernel spinlocks). However, some kernel-mode locking primitives also disable interrupts (e.g. `spin_lock_irq`). Based on previous *Lockset* proposals for supporting interrupts, per-CPU virtual locks are associated with *interrupt* contexts, and are acquired by threads that disable preemption or interrupts. Logical locks are maintained for virtual and real locks, e.g. spinlocks, including *bitlocks* of atomic `test_and_set` instructions. In the evaluation, we call this variant *KLockset*.

Second, we add the mechanisms for handling deferred execution discussed in Section 3.2.3. Finally, we further include state-based synchronization tracking, as follows. For each shared data, in addition to tracking the set of locks held by threads on each access, the set of device states is also tracked. The state variable field in the device class data structure of each driver is used to track device states. When a shared data's set of locks becomes empty at an access, a race is not reported only if the current device state is disjoint with the state set of the data. Instead, the location's metadata is reset to the "exclusive" (i.e., no longer accessed by multiple threads) state.

Note that, as in all our tools (recall Section 2.1), *DRCheck* tracks synchronization in both the driver and kernel-driver interface execution, while reporting races only in the driver execution.

3.3 Direct Memory Acess (DMA) Faults

Direct Memory Access (DMA) is an efficient method for performing bulk I/O data tranfers between system memory and peripheral devices. The main attraction of DMA is that data transfer is performed by device, while (valuable) CPU cycles are conserved. Thus, drivers for high performance devices (e.g., gigabit network cards, and graphics cards) commonly use DMA to efficiently achieve high I/O transfer throughputs. However, incorrect DMA operations are a serious threat to system stability, and thus motivated our third dynamic analysis tool, named *DMACheck*. *DMACheck* performs instruction-grained analysis of driver execution to detect incorrect DMA operations.

DMACheck complements prior IOMMU hardware [2, 1], and software [54] solutions. IOMMU places a restriction on the physical memory locations that a device can DMA into (or from), ensuring that OS (or hypervisor) enforced memory protection is not circumvented by misconfigured DMA transfers. In contrast, *DMACheck* analyzes driver execution to detect misconfiguration of DMA transfers. *DMACheck* and Nexus-RVM [54] are similar in that they both detect DMA errors by checking driver execution, however, they target different types of DMA faults. Nexus-RVM leverages device-specific logic to detect device protocol bugs (e.g., improperly formatted DMA transfer requests). In contrast, and as explained in more details below, *DMACheck* is designed to detect driver violations of the OS protocol regarding DMA operations, and can therefore be used for any device supported by the OS.

To motivate the kind of faults that *DMACheck* was designed to detect, we briefly discuss how DMA is used by drivers. Although the discussion below is based on Linux drivers running on x86 systems, we expect that the issues generally apply to other platforms.

3.3.1 DMA in Linux Drivers

To take advantage of DMA for I/O data transfer, a driver must: (i) map (and pin) physical memory region(s) (a.k.a. *DMA buffer(s)*) to be used as source/destination, into the kernel and I/O address spaces, (ii) inform the device of the DMA buffer(s) (i.e., location in the I/O address space), (iii) signal the device to begin the transfer, and (iv) wait for the transfer to complete.

As with other system resources, the OS kernel controls the management of DMA buffers, and provides functions for mapping DMA buffers into, the kernel address space (i.e., for driver access), and the I/O address space (i.e., for device access), along with the corresponding unmapping functions. Thus DMA buffers are, in general, accessible via two different addresses: (i) virtual addresses, used by drivers, and (ii) device address (I/O bus address), used by devices. The DMA subsystem of the Linux kernel provides a variety of functions (i.e. `dma_map_{single,page,sg}`) for mapping DMA buffers into the I/O address space, and obtaining the corresponding bus addresses.

Before instructing the device to begin transferring data, the driver must supply the device with the bus addresses of the DMA buffers to be used for the transfer. This is done by updating the appropriate set of device registers. The driver's role in setting up DMA is done as soon as it signals the device to commence data transfer. The driver then waits for completion, either by yielding the CPU or performing other important tasks in the meantime. The device signals transfer completion by interrupting the processor. On completion of an incoming transfer, the driver arranges for data to be transferred, up the I/O stack, to the requesting process. For outgoing transfers, the driver optionally releases the source DMA buffers, or recycles them for future use. Nevertheless, a driver must ensure that its DMA buffers are unmapped (i.e. using `dma_unmap_{single,page,sg}`), before it is unloaded.

Because the driver (i.e., via processor) and the device access physical memory (i.e., DMA buffers) through different data buses, the driver is responsible for avoiding coherence problems, i.e., ensure that both work with updated data. To assist drivers achieve this, the kernel provides functions (e.g. `dma_sync_single,sg`) for synchronizing the cache and physical memory copies of DMA buffers. In particular, a driver can use these functions, to fetch DMA buffer(s) from memory into the caches, before accessing incoming I/O data, and flush DMA buffer(s) from the caches, before the device accesses outgoing I/O data.

3.3.2 DMA Buffer Faults

Based on the preceding description of DMA operations by drivers, one might observe a number of ways in which driver defects could cause problems for the I/O subsystem. Specifically, *DMACheck* is designed to check that drivers correctly handle the following DMA buffer issues:

1. **Sharing**: DMA buffers are shared by the driver and device, and so, the driver should avoiding racing the device. In particular, while transfer is in progress, the device should be assumed to have exclusive access to avoid data corruption. For example, driver writes into source DMA buffers could corrupt outgoing I/O data.

2. **Management**: DMA buffers are system resources, and should be carefully managed by drivers. Drivers should avoid leaking (i.e., failing to unmap) DMA buffers, or (un)mapping them multiple times. Leaks wastes the system's DMA resources, while multiple (un)maps could corrupt the DMA subsystem.

3. **Coherence**: device access to DMA buffers bypasses the caches, thus to avoid coherence issues DMA buffers should not share cache line, with other data (including other DMA buffers). One solution is to ensure that DMA buffer size and virtual address are cache line width aligned.

From the 3 issues listed above, we identfied 5 types of DMA buffer faults: (i) data races between driver and device, (ii) leaks, (iii) repeat mapping, (iv) repeat unmapping, and (v) misaligned virtual address. In summary, *DMACheck* is the first use of dynamic analysis to study DMA related problems in drivers. Moreover, we expect that *DRCheck* can be extended for other faults, relating to DMA buffers or DMA in general.

3.3.3 Design

DMACheck detects errors by monitoring how drivers operate on DMA buffers. Linux drivers manipulate DMA buffers using both virtual and bus addresses. For example, the virtual address is used to read/write the DMA buffer while the bus address is used to synchronize the cache and memory copies of a DMA buffer to avoid coherence issues (e.g., `dma_sync_single_for_cpu()`. Thus, *DMACheck* tracks the mapping of a DMA buffer in both the virtual address space and the I/O address space, unlike other driver checking tools (i.e., *DMCheck*, *DRCheck*) which track only kernel address space objects.

The DMA buffer faults that *DMACheck* detects can be grouped into 2 categories, based on the granularity of the detection analysis. Instruction-grained analysis is used to detect races between driver and device—checking if a memory operation by the driver overlaps a DMA buffer that the device is currently accessing. In contrast, the other faults (e.g., misaligned DMA buffers) can be detected by inspecting the arguments of DMA function calls (e.g., `dma_map_single()`) made by the driver.

DMACheck detects races on DMA buffers by checking for unserialized accesses by the driver and device to a DMA buffers. However, doing this precisely is challenging because device access to DMA buffers cannot be (directly) observed by *DMACheck*. Instead, *DMACheck* leverages its ability to observe driver execution to approximate the time intervals when a DMA buffer could be accessed by the device. We identified two pairs of driver operations for approximating this interval for a given DMA buffer: (i) mapping the buffer into the I/O address space, and the corresponding unmapping, and (ii) specifying the buffer as part of a DMA transfer to the device, and the corresponding servicing of the completion interrupt.

Although, at first glance, the second option appears to be a more accurate approximation of when the device actually uses a DMA buffer for transfer, however, the conservative first option turns out to be a more practical approach for a couple of reasons. First, some coherence issues of DMA are addressed when DMA buffer(s) are mapped/unmapped into/from the I/O address space.

For example, the cache lines of a source DMA buffer are flushed when it is mapped for the device to read, and thus later driver updates may not be captured in the transfer. In fact Linux kernel documentation recommends that drivers should not touch DMA buffers that are accessible to the device, without unmapping the buffer or synchronizing the cache and memory copies. Next, the second option introduces the complexity of understanding device-specific logic of how DMA transfers are configured by drivers–an unscalable undertaking, considering the large number of available devices. For these two reasons, *DMACheck* adopts the first option to approximate intervals when the driver should not access a DMA buffer.

3.4 Discussion

As we demonstrate through evaluation with production Linux drivers (Section 4.2), our checking tools can detect errors that are missed by current techniques, which suggests that our techniques can be used to improve driver debugging and testing, or make production systems more resilient to defective drivers. However, in evaluating how to deploy our techniques in these scenarios it is worth considering the practical implications of false negatives and false postives (i.e., *DRCheck*) of the analysis algorithms.

The underlying *Lockset* algorithm of *DRCheck* leads to false data race reports for properly synchronized code that however deviates from the expected locking discipline. This is a serious limitation for production deployments, because halting a system for a false alarm is simply unacceptable. Moreover, the fact that 76%–90% of true races are actually *benign* [29], means that simply avoiding false alarms (e.g., by incorporating a *Happens-Before* approach [57]) is insufficient. However, rather than foregoing race detection entirely on production systems, we believe that this would be an appropriate situation for deploying Guardrail in *triage* mode (Section 2); to automatically classify the alarms raised by *DRCheck* into harmless and harmful races. Furthermore, *DRCheck* could be extended to recognize the synchronization patterns and *benign* data sharing patterns that it had incorrectly flagged in the past to reduce the number of spurious alarms and the need for triaging.

The dynamic nature of our techniques creates the possibility of false negatives,—our tools cannot guarantee driver correctness. Rather, they can only determine whether or not the observed driver executions (i.e., code paths, thread interleavings, and input) are fault-free. For production deployments, this is not a problem since the goal is to keep the system running (i.e., *availability*), until there is a compelling reason to do otherwise (i.e., driver misbehaving). In contrast, for driver debugging or testing, false negatives make it difficult to reproduce bugs or guarantee their absence. Thus, our tools will be more effective for pre-release purposes when combined with techniques for achieving high coverage driver execution [8, 39].

4 Evaluation

We developed a prototype of Guardrail to answer two questions:

1. How effectively do our techniques detect driver faults, particularly when compared with existing techniques?

Class	Driver	Device
Audio	snd_hda_intel	High Def Audio (ICH7)
Network	tg3	BCM5754 1Gpbs NIC
Storage	ahci	ICH7 SATA disk
Video	nvidia	Quadro NVS 285

Table 1: Linux drivers and devices used in commodity hardware studies.

Class	Driver	Device model
Network	e100	I82559 100Mbps NIC
	e1000	I82543gc 1Gbps NIC
	pcnet32	AM79C973 100Mbps NIC
	tg3	BCM5703C 1Gbps NIC
	tulip	DEC21143 100Mbps NIC
Storage	qla1280	ISP1040 SCSI disk
	qla2xxx	ISP2200 SCSI disk
	sym53c8xx	SYM53C875 SCSI disk

Table 2: Linux drivers and devices used in simulated hardware studies.

I/O type	Benchmark	Version	Description
Audio & Video	Mplayer	1.0	Multimedia player
Network	Apache	2.2.6	Webserver
	Memcached	1.2.3	In-memory key value store
	Netperf	2.4.0	Network performance measurement tool
Storage	GNU Make	3.81	Software compilation utility
	Postmark	1.5.1	Filesystem benchmark

Table 3: The I/O intensive benchmarks used for evaluation.

Hardware	Network client						Postmark		
	Apache			Memcached		Netperf			
	Threads	Requests	File size	Threads	Req/thread	Length	Trx.	Files	File size
Real	1–32	16K	40KB	32–256	100K	20 secs.	100K	20K	10KB–20KB
Simulated	16	1600	40KB	16	1K	5 secs.	100K	1K	10KB–20KB

Table 4: Benchmark parameters.

2. What is the impact on the system end-to-end performance, of using Guardrail to protect I/O devices from driver faults, particularly if the monitored device is heavily used?

4.1 Methodology

In our Guardrail prototype, the I/O interposition layer is a modified version of a paravirtualized (PV) Xen-3.3.1, which includes our extensions for containing potential driver faults. The guest OS is based on the Linux 2.6.18 kernel. A variety of devices were evaluted in this software environment, and all of the corresponding drivers were stock, unmodified binaries. The devices were directly assigned [55] to the VM, to enable the execution of non-paravirtualized drivers, and minimize I/O virtualization overheads [13].

Benchmarks We tested our implementation with a variety of device types, and consequently, we used a set of popular I/O benchmarks, listed in Table 3, to understand how our techniques affect the reliability and performance of the corresponding drivers.

We used the open source media player, *Mplayer*, to evaluate the audio and video drivers; and we used it's benchmarking feature to collect our experimental results. We evaluated the network drivers using the *Apache* web server, the *Memcached* in-memory key-value store, and the *Netperf* network performance measurement tool. To generate input loads, we used Apache's benchmarking tool, *ApacheBench*, and memcached's benchmarking tool, *Memslap*. We evaluated storage drivers with the *Postmark* filesystem benchmark and kernel compilation.

Experimental Setup We conducted experiments on both real and simulated multicore x86 hardware, but used the same software stack in both environments.

To evaluate the fault detection of our proposed instruction-grain tools (Section 4.2), we simulated hardware assisted logging [7, 52] to trace driver execution because we did not have access to a software logging mechanism for tracing kernel-mode execution. Aftersight [10] and Retrace [56] are proprietary tools, while [17] was proposed after we had completed our evaluations[5].

Our performance studies focused on the overheads of our virtualization-based interposition (Section 4.3) and of deploying the checking tools in decoupled fashion to protect I/O operations (Section 4.4). We conducted experiments on real hardware platforms to measure interposition performance. For similar reasons as our fault detection study, we simulated hardware-assisted logging to measure the end-to-end performance of using Guardrail to protect I/O operations from driver faults (Section 4.3). This also enabled us to factor out the overheads of software logging and measure only the performance of the tools.

Our evaluations on real hardware focused on 4 classes of drivers in the 2.6.18 kernel, namely audio, network, storage, and video. However, only storage and network drivers were used in our simulation studies because of the difficutly of obtaining audio and video device models for our simulator (Simics [46]). The drivers used for our real hardware studies are presented in Table 1, while those used for simulation studies are listed in Table 2. Table 4 shows parameter settings for the experiments conducted on both real and simulated hardware. Further details of the experimental setups for the performance studies on real and simulated hardware are provided in Sections 4.3 and 4.4 respectively.

[5]Initial consultations with the authors indicate that significant engineering effort would be required to incorporate into our framework.

Tool	e1000	qla2xxx	Total
DMCheck	1	1	2
DDT	0	1	1
KAddrheck	0	1	1
KMemcheck	1	1	2

Table 5: *DMCheck* found 2 memory bugs that are now fixed, and discovered the *qla2xxx* bug. While KMemcheck can find both bugs, the other tools will find only one.

4.2 Fault Detection

We implemented each of the proposed dynamic analysis tools: (i) *DMCheck*, for detecting memory faults (Section 3.1), (ii) *DRCheck*, for detecting data races (Section 3.2), and *DMACheck*, for detecting DMA faults (Section 3.3), in Guardrail to evaluate how they can improve the quality of production drivers. For the evaluation, we applied them to the 8 production Linux drivers (and corresponding I/O devices) listed in Table 2. The drivers were put under normal I/O workloads for this study.

As shown in Tables 5, 6 and 8, our tools found serious errors in these real-world drivers, including previously unknown bugs. We examine the bugs found each tool in more details below, and then compare each tool against competing kernel-mode dynamic correctness checkers, by evaluating whether all the bugs could be detected using competing techniques.

4.2.1 Memory Faults

As shown in Table 5, *DMCheck* found 2 serious memory faults, both of which have been fixed. In particular, the *qla2xxx* memory bug was previously unknown until reported by our tool. Based on our report, the bug was eventually fixed in the 3.2 release of the Linux kernel, 6 years after the 2.6.18 version that we used for our study. Because these bugs involve memory that is exclusively used by the driver, they cannot be detected using fault isolation technqiues that only check driver interraction with the kernel [53, 50, 58, 20, 54, 6]. For example, the *e1000* memory bug is an unsafe use of uninitialized stack data, while the *qla2xxx* memory bug is an out-of-bounds read of memory mapped device registers.

Furthermore, we use Table 5 to compare *DMCheck* against existing kernel-mode memory fault detectors for the Windows (*DDT* [25]), and Linux (*KAdddrcheck* [17], *KMemcheck* [32]). *DDT* and *KAddrcheck*, track memory addressibility, and therefore can only detect the out-of-bounds bug. *KMemcheck*, on the other hand, tracks both memory addressibility and initialization, and should therefore detect both the memory faults.

4.2.2 Data Races

As shown in Table 6, *DRCheck* found 9 serious data races in 5 Linux drivers, 6 of which have either been confirmed or fixed. Also, using this table, we compare *DRCheck* with DataCollider based on the details in [15]. We made a couple of assumptions in our analysis to increase the

Tool	qla1280	qla2xxx	sym53c8xx	tg3	tulip	Confirmed/Fixed	Unconfirmed*	Total
DRCheck	1	3	2*	2	1*	6	3	9
Det-DataCollider	0	1	0	1	0	2	0	2
Ideal-DataCollider	1	2	2*	1	0	4	2	6

Table 6: DRCheck found 9 serious races in Linux drivers, 6 of which were confirmed/fixed. DataCollider will detect 2 races, if it sampled the racy accesses, and 6 races if racy accesses occured in an idealized order for it.

	qla1280	qla2xxx	sym53c8xx	tg3	tulip
KLockset	1	36	13	35	26
DeferExec	1	13	13	22	18
DRCheck	0	0	6	4	1

Table 7: False positives of our different race detection techniques.

Fault Type	Count	Drivers
Data race	7	tulip (7)
Leak	4	sym53c8xx (4)
Repeated map	2	tg3 (1), tulip (1)
Repeated unmap	2	tulip (2)
Misaligned virtual address	10	e100 (1), e1000 (1), pcnet32 (3), tg3 (2), tulip (3)

Table 8: Summary of DMA buffer faults detected by *DMACheck* in Linux drivers. The number of fault instances found in each driver is shown in parenthesis.

chances that DataCollider's sampling will detect the races. First, we assume that the racy accesses, outside of interrupt contexts, are deterministically sampled (*Det-DataCollider*). DataCollider does not sample interrupt context accesses for robustness reasons. Second, for races involving interrupt and non-interrupt contexts, we assume that the non-interrupt context access occured earlier (*Ideal-DataCollider*). With these assumptions, 2 races will be detected by *Det-DataCollider*, and 6 races by *Ideal-DataCollider*.

However, unlike *DataCollider* which has no false positives, *DRCheck* generated a small number of false alarms while detecting these driver races, as shown in Table 7 (*DeferExec* is *DRCheck* without state-based synchronizations (Section 3.2.4)). Although *DDT* [25] detects data races, it was not described in sufficient details to allow comparisons in our context.

4.2.3 DMA Faults

The different DMA buffer faults found by *DMACheck*, in 6 drivers are summarized in Table 8. Races on DMA buffers, which are the most serious of these bugs, affected only the *tulip* network driver. *DMACheck* found 7 unique driver writes (i.e., static instruction address) that could potentially corrupt I/O data that was being read by the network card. DMA buffers with unaligned virtual addresses (assuming 32 byte cache lines) are the most common fault type—affecting 5 drivers

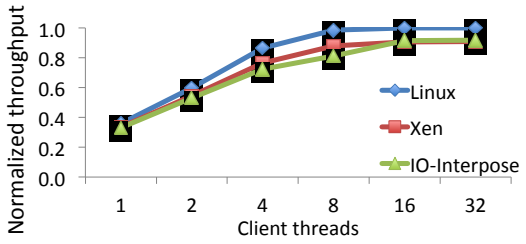

Figure 6: I/O interposition overheads on *Apache* throughput; normalized to peak *Linux* throughput.

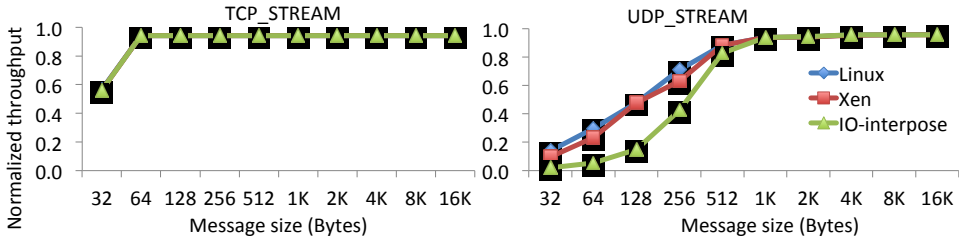

Figure 7: I/O interposition overheads on *Netperf* throughput; normalized to BCM5754's link rate (1Gbps).

(i.e., *e100*, *e1000*, *pcnet32*, *tg3*, *tulip*). *sym5c8xx* was the only driver that leaked DMA buffers (i.e., failed to unmap DMA buffers before unloading), while *tulip* and *tg3* were the only drivers to map previously mapped DMA buffers, or unmap previously unmapped DMA buffers. Although these faults reflect programmer error in managing DMA operations, and should be avoided, however we did not observe any resulting system failures during our experiments.

Because we are unaware of any prior work in detecting DMA faults, we could not compare *DMACheck* against alternative approaches.

4.2.4 Summary

In summary, our evaluation validated our proposal that, instruction-grained dynamic analysis can improve system reliability by detecting real-world bugs in production drivers. Our dynamic analysis tools detect a significant number of bugs in production Linux drivers: memory faults, data races, and DMA faults, that are missed by other tools, including a previously unknown memory faults in the *qla2xxx* storage driver. Moreover, to our knowledge, our study of DMA buffer faults using dynamic analysis is the first of its kind. The superior bug detection quality of our proposed dynamic tools sometimes incurs a small number of false positives, e.g., for data race detection. Also, Guardrail's support for all these tools demonstrate it's value as a general-purpose framework for implementing driver correctness checking tools, in contrast to error-specific tools such as *DataCollider* and *KMemCheck*.

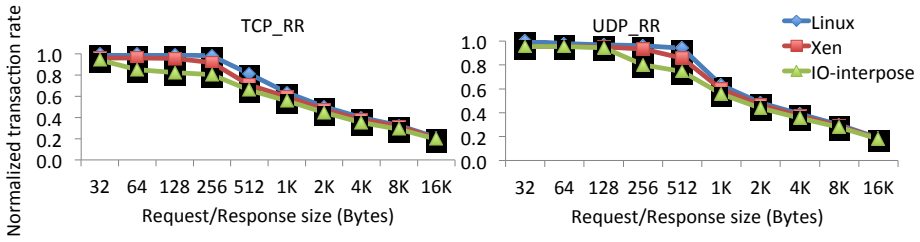

Figure 8: I/O interposition overheads on *Netperf* transfer rate; normalized to peak *Linux* rate.

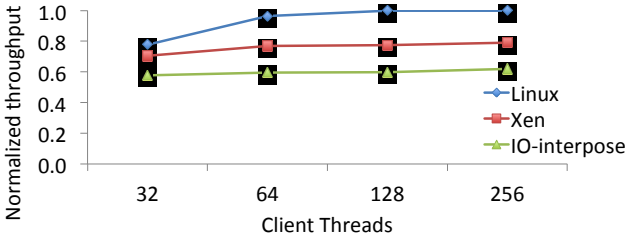

Figure 9: I/O interposition overheads on *Memcached* throughput; normalized to peak *Linux* rate.

4.3 I/O Interposition Performance

We evaluated the performance of our I/O interposion techniques using the drivers in Table 1. We define baseline performance as the driver running in an unvirtualized system with direct access to the device of interest. We use the following naming convention to present results: the baseline (unmodified 2.6.18 kernel) is *Linux*, adding virtualization without I/O interposition (unmodified PV Xen-3.3.1) is *Xen*, and our I/O interpositioning modifications is *IO-Interpose*. The test system used for the evaluation is a dual-core Intel Core 2 system running at 2.66 GHZ with a 2 GB RAM. For convenience, we ran the driver in dom0. For the network performance studies, which involved client-server experiments, we studied the server running in the test system. The client load was supplied by a non-virtualized, 32-bit Ubuntu 10 (2.6.32 kernel) system (quad-core Intel Core 2 system running at 3 GHz with 4GB RAM). All other I/O experiments were conducted on the test system.

4.3.1 Audio & Video Performance

We used *Mplayer's* benchmarking features to measure how interposition affects the generation of audio, and video outputs of multimedia files. The multimedia file used in our experiments, was a 150 seconds long movie trailer recorded in the movie industry standard $1080p24$ *Full HD* format (i.e., 1920 x $1080p$ resolution and 24 frame rate). *Mplayer* was configured to use *ALSA* mode for audio output, and *X11* mode, for video output. The reported results are median of 10 runs.

The movie playback on *IO-Interpose* was smooth, and of similar audio, and video quality to *Linux*, and *Xen*. There were no dropped frames, and virtually identical frame rates were achieved, on the different systems. The reported median frame rates were 23.94 on *Linux*, 23.93 on *Xen*, and 23.91 on *IO-Interpose*. These results show that user experience is barely impacted by I/O

	Audio Output		Video Output		Playback
	Time(s)	Overhead(%)	Time(s)	Overhead(%)	Time(s)
Linux	1.22	-	47.57	-	150.51
Xen	1.28	5	47.73	0	150.52
IO-Interpose	1.27	4	47.77	0	150.52

Table 9: Impact of I/O interpostion on audio and video performance.

interposition on modern multimedia workloads.

Table 9 presents the audio, and video output generation times, as well as the overall playback time (including audio and video decoding). Generating the audio output took 4% longer on *IO-Interpose*, compared to *Linux*, while video output time was not noticeably different. Unsurprisingly, these small slowdowns did not affect the overall quality of the movie. Moreover, *Xen* performed similarly to *IO-Interpose*, which suggests that a significant portion of interposition overheads can be attributed to virtualization.

4.3.2 Network Performance

For the network experiments described below, we report the median of 10 runs in all cases.

Apache We configured *ApacheBench*, as shown in Table 4, to make 16000 requests for a 40 KB static page from the *Apache* server, and measured the transfer rate for a varying number of client threads. As shown in Figure 6, *IO-Interpose* is scalable, as the transfer rate scales with increasing client threads. Although, performance degradation due to *IO-Interpose* varies depending on client thread count, we observed that when the *Linux* server is saturated (64 client threads), *Xen* and *IO-Interpose* are both only 7% worse.

Memcached *Memcached* was configured to run on the server using 2 threads and a 512 MB in-memory object store. *Memslap* is configured to initially load the server with $100K$ objects, and then use a configurable number of threads to make *get* requests, with $100K$ requests per thread (Table 4). The results are shown in Figure 9. We observed that with *IO-Interpose*, server throughput scales, albeit modestly, with increasing client threads. Moreover, when the *Linux* server is saturated (256 client threads), *Xen* and *IO-Interpose* offer 26% and 35% lower throughput, respectively.

Netperf We measured the impact of I/O interposition on network throughput and transaction rate, using the TCP and UDP transport versions of the stream tests (TCP_STREAM and UDP_STREAM), and request/response (TCP_RR and UDP_RR) tests. We ran a single instance of each test from the client for 20 seconds, varying the message sizes, but with default settings of other parameters. As shown in Figure 7, while *IO-Interpose* delivers similar throughput as *Linux* and *Xen* for TCP_STREAM, its UDP_STREAM throughput degrades for small (< 512 bytes) messages. Further investigations showed that for UDP_STREAM, server load increased significantly with small messages, and so, the increased CPU utilization directly exposes the overheads of interposing on

I/O operations. This also explains the poor *Linux* and *Xen* performance with small messages. In contrast, server load is fairly insensitive to the size of TCP_STREAM messages. For transaction rates, *IO-Interpose's* performance is comparable to the others for most message sizes, as shown in Figure 8.

4.3.3 Storage Performance

We now describe our evaluation of the impact of interposition on storage performance, using kernel compilation and *Postmark*.

GNU Make We measured how I/O interpostion affects the time that it takes to compile a Linux 2.16.8 kernel with default configuration, using the GNU *Make* utility. The compilation process was performed using the stock *GCC* 3.4.6 compiler that was distributed with the OS kernel. In addition, we leveraged the parallel compilation feature (`-j`) of *Make*, to evaluate the scalability of our I/O interposition prototype on the Dual-core test system.

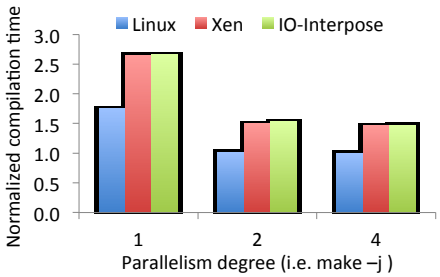

Figure 10: Kernel compilation time; normalized to 4-way parallel compilation time on *Linux*.

In Figure 10, we present the kernel compilation times, measured on *Linux*, *Xen*, and *IO-Interpose* for different degrees of parallelism, and normalized to the best *Linux* result. We observed that on each system, compilation time improved as parallelism increased, and this improvement peaked at 4-way parallel compilation. Some intuition for this improved performance can be gleaned from Figure 11, which shows that *Make* exploits available parallel computing resources. For example, CPU utilization increased from 52% for sequential compilation, to 95% for 4-way parallel compilation, while the increase was 47% to 95%, for *IO-Interpose*. These results show that, similar to its comparision points, *IO-Interpose* allows storage I/O performance to scale with increased parallelism in computing resources.

With 4-way parallelism, *IO-Interpose* increased compilation time by 47%, relative to *Linux*. This represents a modest improvement over the 52% degradation that we observed with sequential compilation, suggesting that *IO-Interpose* benefits relatively more from parallel compilation. However, since similar overheads were observed with *Xen*, regardless of parallelism degree, we suspect that most of the interposition overheads are due to virtualization.

Postmark *Postmark* [24], is a single-threaded benchmark that simulates an Internet e-mail server. We configured *Postmark*, as shown in Table 4, to perform $100K$ file transactions (create, delete,

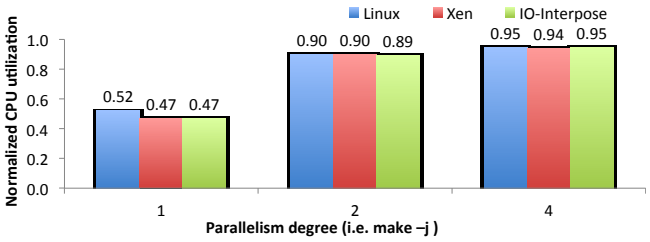

Figure 11: CPU utililzation of kernel compilation for different degrees of parallelism.

read, write) operations on $20K$ files with size range $10KB$–$20KB$. All other parameters were left at default values. The results presented below, are the median of 10 runs.

In Figure 12, we show the measured transaction, read, and write rates, normalized to *Linux*. We observe that relative to *Linux*, *IO-Interpose* degrades each performance metric by about 10%; 9% for transaction rate, and 10% for read and write rates. Just as we observed for kernel compilation, *Xen* does not perform noticeably better than *IO-Interpose*.

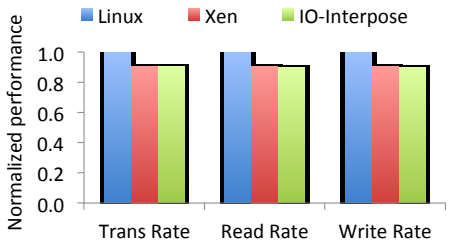

Figure 12: *Postmark* transaction, read, and write rates; normalized to *Linux* rates.

4.3.4 Summary

We used experiments involving I/O intensive benchmarks, to show that modest overheads (at most 10%) are imposed by I/O interposition on I/O performance in many cases. In particular, we observed that the quality of audio and video playback, using *Mplayer*, is virtually unaffected by interpositon, and the frame rate was virtually unchanged during the playback. Other benchmarks, for which we observed similarly low overheads incude: (i) *Apache* (7%), all the *Netperf* tests except UDP_STREAM, and *Postmark*.

On the other hand, higher overheads were incurred by *Memcached* (35%) and kernel compilation (47%). Furthermore, we observed that virtualization was responsible for significant portions of these overheads; about 75% of *Memcached* overheads, and virtually all the overheads experienced for kernel compilation, can be attributed to virtualization.

Moreover, we observed that our interposition prototype offers scalable performance; either scaling up to meet increasing concurrency in client load, or scaling up to better utilize available parallel computing resources.

Simulated H/W	CMP	Dual-Core, x86, 2.6Ghz, 2GB RAM
	Private L1I	16KB, 64B line, 2-way assoc, 1-cycle access lat.
	Private L1D	16KB, 64B line, 2-way assoc, 1-cycle access lat.
	Shared L2	2MB, 64B line, 8-way assoc, 10-cycle access lat, 4 banks
	Main Memory	200-cycle access latency
	Tracing	512KB log buffer
	DriverVM	2 VCPU, 1GB RAM
	Analysis VM	1 VCPU, 512MB RAM
Real Word S/W	VMM	PV Xen-3.3.1
	OS	PV 32-bit Linux kernel (Fedora Core 6)
	Network driver	*tg3*
	Storage driver	*sym53c8xx*

Table 10: Simulated Guardrail system used for measuring performance impact of protecting I/O devices from driver faults.

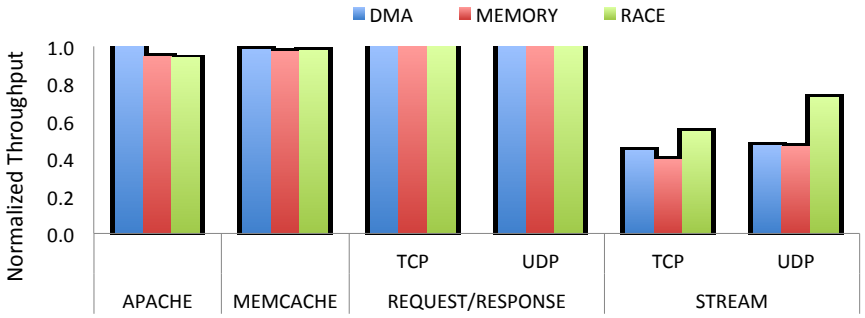

Figure 13: Throughput when protecting *BCM5703C* NIC from *tg3* faults; normalized to no protection.

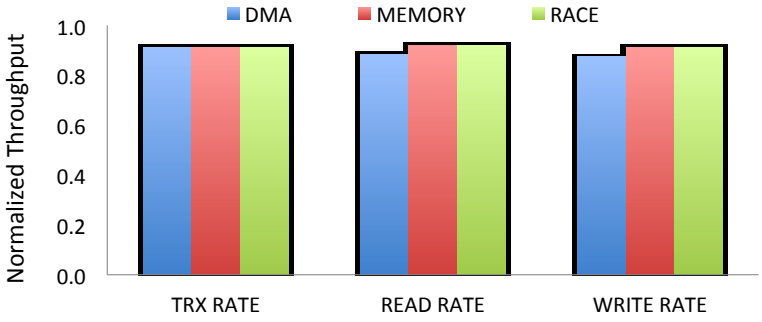

Figure 14: *Postmark* performance when protecting *SYM53C875* SCSI disk from *sym53c8xx* faults; normalized to no protection (i.e., *Linux*).

4.4 End-to-End Performance

To understand the impact of online driver monitoring on end-to-end I/O performance, we simulated Guardrail as a dual-core, 2.6 GHz, x86 CMP system with 2GB memory. For comparison, *Linux*,

Xen and *IO-Interpose* are evaluated using a similar simulated x86 CMP system (minus the decoupling extensions). In the virtualized environments, the driver runs in domU (*DriverVM*), while the analysis runs in dom0 (*AnalysisVM*) when monitoring is enabled. domU is configured with 2 virtual CPUs, and a 1GB memory, 512KB of which is reserved for execution logging (assuming each instruction record can be compressed down to a byte [7]). dom0 is configured with 1 virtual CPU, and a 512MB memory. The details of the simulated environment that was used for this performance study are illustrated in Table 10. The non-virtualized client, for network experiments, is simulated using similar hardware configuration, and runs a stock Fedora Core 6 (2.6.18) kernel.

One drawback of this performance study is that, due to robustness issues of the simulated device models, we were forced to scale down the input parameters (see Table 4). The immediate impact of smaller input parameters was that virtualization and I/O interposition overheads became negligible (i.e., *Linux*, *Xen*, and *IO-Interpose* performed similarly). However, even with this reduced input sizes, only *SYM53C875* SCSI disk and *BCM5703C* network card models were robust enough for our experiments. All experiments were simulated to completion.

4.4.1 Decoupling

First, we evaluate the efficiency of using Guardrail to decouple dynamic analysis from the monitored driver execution. Since hardware-based execution tracing incurs no overheads, we specifically measured whether Guardrail's analysis scheduling technique (Section 2.2) introduces additional monitoring overheads beyond that of the checking tools. We used an idealized checking tool (*NullCheck*) that consumes available log entries in a single cycle to measure any overheads relative to *IO-Interpose*.

We did not notice any I/O performance degradation by *NullCheck*, relative to *Linux* in all cases, showing that Guardrail schedules *NullCheck* to consume execution traces in a timely fashion. Additionally, we observed that compared to Guardrail, polling increased *NullCheck*'s CPU usage significantly (e.g. > 300X for monitoring TCP_RR & UDP_RR).

4.4.2 Protecting I/O Devices

Next, using *Linux* as baseline, we evaluate the impact of using *DMCheck*, *DRCheck* and *DMACheck* to protect the device, from the corresponding driver faults, on end-to-end performance. For this experiment, we did not apply any optimizations on the tools, for better performance, and so these are conservative results.

The impact of protecting the *BCM53703C* network card on the throughput of different network intensive workloads is presented in Figure 13. The figure reports the normalized throughput relative to running without protection. We observed that most of the benchmarks experienced minimal throughput loss, the exception being network streaming using TCP and UDP. In particular, for TCP and UDP streaming respectively, *DMACheck* reduced throughput by 55% and 53%, *DMCheck* by 60% and 53%, and *DRCheck* by 45% and 27%. However, the other benchmarks experienced very little performance impact, and in particular, the worst case performance for each checker was with *Apache*, where *DMACheck* reduced throughput by 1%, *DMCheck* by 5%, and *DRCheck* by 6%.

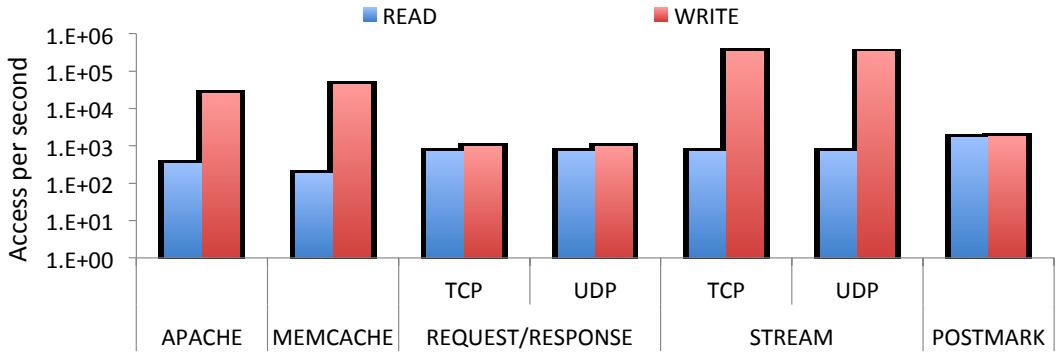

Figure 15: Rates of device register access by network benchmarks on the *BCM5703C* NIC, and a storage benchmark (Postmark) on the *SYM53C875* SCSI disk.

Our investigation into the severe degradation of network streaming performance suggests that the high rate of device register accesses by networking streaming, compared to other workloads, could be the reason for the overheads of driver monitoring. As shown in Figure 15, networking streaming generates device register accesses (especially writes) at a rate that is orders of magnitude higher than other workloads. In particular, we observed over 300K device register writes per second for network streaming compared to about 25K and 40 writes per second for *Apache* and *Memcache* respectively. Since driver execution is stalled at device register accesses, until validation by the (potentially lagging) analysis, it means driver stalling is significantly more frequent for network streaming.

Figure 14 presents the impact of protecting the *SYM53C875* SCSI disk on the performance of the *Postmark* benchmark. The figure reports the normalized read, write, and transaction rates of the benchmark, relative to running without protection. We observed that protecting the disk from faults incurred only modest overheads. In particular, the worst overheads for each tool was experienced for writes, 13% for *DMACheck* and 9% for *DMCheck* and *DRCheck*. This relatively good performance, compared to network streaming, can be explained by Figure 15, which shows that *Postmark* generates about 3K device register accesses per second, and thus leads to less frequent driver stalls.

4.4.3 Summary

Our experiments showed that even with unoptimized sophisticated checking tools, online protection of I/O devices from subtle driver faults (e.g., memory faults, data races) can be achieved with mininal impact on end-to-end performance of most I/O intensive benchmarks. Network streaming was the exception to this, and we observed up to 60% drop in throughput. However, we expect that these overheads can be significantly reduced through existing software [37, 31, 41, 40] and hardware [7, 52] techniques for accelerating dynamic analysis.

Proposal	Detection		Protected component	System support
	Events	Synchronous		
BGI	Interface	Yes	Kernel	
DataCollider	Internal	Yes	Kernel	Breakpoints
KAddrcheck	Internal	Yes	Kernel	
MicroDriver	Interface	Yes	Kernel	Memory protection
Nexus-RVM	Interface	Yes	Device Kernel	Memory protection
Nooks	Interface	Yes	Kernel	Memory protection
SafeDrive	Internal	Yes	Kernel	
SFI	Interface	Yes	Kernel	
SUD	Interface	Yes	Kernel	Memory protection IOMMU
XFI	Interface	Yes	Kernel	
Guardrail	Internal	No	Device	Virtualization Execution tracing

Table 11: Classification of techniques for mitigating driver faults.

5 Related Work

Despite pre-release efforts to avoid and remove faults in drivers, the unfortunate reality is that driver faults still escape into production environments, and lead to a significant fraction of system failures. Therefore, production system reliability depends on the ability to tolerate these faults. Driver faults are a threat to system reliability, because when executed, the driver transitions into an erroneous state (i.e. become *faulty*), which could lead to a system crash, or corruption of the kernel, devices or other drivers. Therefore, fault tolerance requires the protection of trusted system components (e.g., OS kernel and I/O device) from the harmful effects of driver faults; so that the system continues to operate correctly, albeit with degraded I/O functionality. Recovery from driver faults is also desirable for fault tolerance; to reclaim system resources from the failed driver, and to resume disrupted I/O services by restarting the driver in the case of transient faults. Moreover, fault recovery is simplified by restricting the scope of damages to the driver.

Driver faults can be prevented from compromising trusted system component(s) by interposing on the respective driver interface(s) (e.g., to kernel library functions). Current proposals for mitigating driver faults can be classified based on: (i) fault detection approach, (ii) range of protected system components, and (iii) required system support.

5.1 Detection

Checking a driver's execution for correctness violations is an effective technique for detecting faults. Current techniques can be classified based on: (i) whether checking is applied to all of driver execution (*Internal-checking*), or just the interface execution (*Interface-checking*), and (ii) whether

checking (and thus fault detection) is synchronous (*Synch-checking*) or asynchronous (*Asynch-checking*) with driver execution.

Interface-checking imposes lower monitoring overheads than *Internal-checking* because driver interactions with the system typically constitue a relatively small portion of overall execution. However, this benefit comes at price of reduced fault coverage because the information available at the driver's interface (e.g., function call arguments and return values) are often insufficient to determine that a driver is behaving correctly. On the other hand, *Internal-checking*, especially instruction-by-instruction, offers high fidelity fault detection, and typically identifies faults much earlier in the execution, which helps with debugging. Howevever, as shown in Table 11 *Interface-checking* is the more popular approach, perhaps because of the performance advantage. Proposals based on *Interface-checking* include SFI [53], Nooks [50], XFI [58], Microdrivers [20], Nexus-RVM [54], BGI [6], and SUD [5]. In contrast, *Internal-checking* is used by Guardrail, SafeDrive [16], DataCollider [15], and KAddrcheck [17].

Synch-checking preemptively checks driver execution events (e.g., memory access) and therefore detects when a driver is about to become faulty. In contrast, *Asynch-checking* checks a history of driver actions, and thus realizes that a driver is faulty after the fact. Delayed fault detection makes *Asynch-checking* more challenging than *Synch-checking* in two ways. First, protecting the system from faults that manifest in driver execution is relatively easier with *Synch-checking* because unsafe actions are intercepted before completion. In contrast, *Asynch-checking* requires additional efforts to protect the system from the harmful effects of detected faults. For example, Guardrail stalls driver execution at interface events to synchronize with the (potentially) lagging checking tool and prevent faults, that will soon be detected by the tool, from compromising I/O operations. Second, recovery from driver faults is relatively easier with *Synch-checking* since the driver is prevented from executing in a faulty mode. With *Asynch-checking*, recovery must handle the side-effects of allowing drivers to execute in a faulty mode for some period of time.

Conversely, *Asynch-checking* offers a couple of benefits over *Synch-check*. First, because *Synch-checking* overheads are directly incurred on the critical path of driver execution, heavyweight analysis (e.g., data race detection) are impractical for timing-sensitive computations (e.g., interrupt service routines) in the driver. In contrast, by decoupling the analysis from driver execution, *Asynch-checking* can significantly accelerate heavyweight analysis [11, 31, 7, 41, 52, 40] to enable comprehensive driver monitoring. Second, *Synch-checking*'s precision (i.e., no false positives/negatives) depends on the atomic execution of the check and checked event, which can only be guaranteed if the kernel-space is race-free. *Asynch-checking* has no such dependencies, but requires a recording of shared memory dependencies of the driver's execution [52, 35, 36].

Perhaps due to the challenges of delayed fault detection, *Synch-checking* is significantly more popular than *Asynch-checking* amongst current proposals. As shown in Table 11 all but Guardrail employ *Synch-checking*. However, by demonstrating the benefits of *Asych-checking*, we hope that Guardrail could motivate further exploration of this approach.

5.2 Protected Components

Techniques for mitigating driver faults can also be evaluated by their containment guarantees, i.e. which system components they protect. Because the OS kernel and persistent device state are the

most vulnerable system components to faulty drivers, protecting them is the focus of most proposals. However, as shown in Table 11, OS kernel protection is the most popular concern, and is the sole focus of all, but two, of the current techniques. The exceptions are Nexus-RVM, which protects both kernel and device, and Guardrail, which protects only the device. Existing techniques protect the kernel from a variety of driver faults including: wild pointer writes, uninitialized memory use (SafeDrive), buffer overflows (SafeDrive and KAddrcheck), control flow integrity faults (XFI and BGI), type safety (BGI), and data races (DataCollider). Nexus-RVM protects the device from device protocol faults and memory faults, while Guardrail protects the device from memory faults, data races, and DMA buffer errors..

5.3 System Support

Proposals for mitigating driver faults also differ in terms of the required system support, as shown in Table 11. SFI, XFI, SafeDrive, BGI, Nexus-RVM, and KAddrcheck perform fault isolation using software checks, and can therefore be deployed without additional mechanisms on commodity systems. Nooks, Microdrivers, and SUD, complement software checks with commodity hardware page protection. However, SUD requires relatively new IOMMU hardware [2, 1], which is increasingly available on commodity systems, for additional kernel protection. Finally, Guardrail relies on virtualization, and execution tracing of kernel-mode execution, both of which can be obtained on commodity systems. However, our current Guardrail prototype assumes the performance benefits of hardware assisted execution tracing.

5.4 Summary

Mitigating driver faults in computing systems is of significant research interest, due to the high bug rate of production drivers. OS kernel protection has drawn the most attention; 10 of the 11 reviewed proposals, protect the kernel, while only Nexus-RVM and Guardrail provide protect persistent device state. Most techniques treat the driver as a black box; focusing on the correctness of its interactions with the rest of the system, rather than whether it is executing correctly. Although this approach reduces monitoring overheads, it also reduces fault coverage. Also, synchronous checking of driver execution is the most popular approach, with Guardrail the only technique that asynchronously checks driver execution for faults.

6 Conclusions

This paper demontrated the feasibility of applying sophisticated dynamic analyses such as data race detection in the context of unmodified, kernel-level device drivers. Moreover, it described a system that uses dynamic analyses to monitor driver execution, and uses commodity virtualization technology to protect persistent device state from memory errors, data races and DMA buffer errors transparently. The checking tools employed were shown to detect more driver faults than existing fault detection techniques. Our data race tool improves upon prior approaches by minimizing false positives and avoiding false negatives, while handling the complexities of kernel-mode drivers.

Finally, the paper showed that decoupling dynamic analysis from driver execution reduces the performance impact of monitoring in most cases.

References

[1] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3), 2006.

[2] Steve Apiki. I/O Virtualization and AMD's IOMMU. *http://developer.amd.com/documentation/articles/pages/892006101.aspx*, 2006.

[3] Thomas Ball, Ella Buonimova, Byron Cook, Valdimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustunner. Thorough Static Analysis of Device Drivers. In *EuroSys*, 2006.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[5] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *USENIX*, 2010.

[6] Miguel Castro, Manuel Costa, Jean-Phillipe Martin, Marcus Peinado, Periklis Akritidis, Austin Donelly, Paul Barham, and Richard Black. Fast Byte-Granularity Software Fault Isolation. In *SOSP*, 2009.

[7] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible Hardware Acceleration for Instruction-grain Program Monitoring. In *ISCA*, 2008.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *ASPLOS*, 2011.

[9] Andy Chou, Jufeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *SOSP*, 2001.

[10] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX*, 2008.

[11] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *ISCA*, 2003.

[12] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS*, 2008.

[13] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. In *HPCA*, 2010.

[14] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System rules using System-specific, Programmer-written Compiler Extensions. In *OSDI*, 2000.

[15] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, 2010.

[16] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *OSDI*, 2006.

[17] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *ASPLOS*, 2012.

[18] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[19] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP Kernel Crash Analysis. In *LISA*, 2006.

[20] Vinod Ganapathy, Matthew Renzelmann, Arini Balakrishnan, Michael Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *ASPLOS*, 2008.

[21] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs. In *ASPLOS*, 2011.

[22] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating Hardware Device Failures in Software. In *SOSP*, 2009.

[23] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *ASPLOS*, 2012.

[24] Jeffrey Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, October 1997.

[25] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing Closed-Source Binary Device Drivers with DDT. In *USENIX*, 2010.

[26] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery domains: an organizing principle for recoverable operating systems. In *ASPLOS*, 2009.

[27] Ben Leslie, Peter Chubb, Nicholas Fitzroy-dale, Stefan Gtz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20, 2005.

[28] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: an idl for hardware programming. In *OSDI*, 2000.

[29] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007.

[30] Nicholas Nethercote and Julian Seward. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, 2007.

[31] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing Security Checks on Commodity Hardware. In *ASPLOS*, 2008.

[32] Vegard Nossum. Getting started with KMemcheck. *http://www.mjmwired.net/kernel/Documentation/kmemcheck.txt*, 2012.

[33] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *ASPLOS*, 2011.

[34] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, 2010.

[35] Gilles Pokam, Cristiano Pereira, Klaus Danne, Rolf Kassa, and Ali-Reza Adl-Tabatabai. Architecting a Chunk-based Memory Race Recorder in Modern CMPs. In *MICRO*, 2009.

[36] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. Coreracer: a practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2011.

[37] F. Qin, C.Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO-39*, 2006.

[38] Matthew Renzelmann and Michael Swift. Decaf: Moving Device Drivers to a Modern Language. In *USENIX*, 2009.

[39] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: testing drivers without devices. In *OSDI*, 2012.

[40] Olatunji Ruwase, Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Decoupled lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking tools. In *PLDI*, 2010.

[41] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing Dynamic Information Flow Tracking. In *SPAA*, 2008.

[42] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *EuroSys*, 2009.

[43] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic Device Driver Synthesis with Termite. In *SOSP*, 2009.

[44] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Race Detector for Multithreaded Programs. *ACM TOCS*, 15(4), 1997.

[45] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitzer - Data Race Detection in Practice. In *WBIA*, 2009.

[46] Simics. Wind River Simics Full System Simulator. http://www.simics.net/, 2010.

[47] Michael F. Spear, Tom Roeder, Orion Hodson, Galen C. Hunt, and Steven Levi. Solving the starting problem: device drivers as self-describing artifacts. In *Eurosys*, 2006.

[48] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI*, 2004.

[49] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24(4):333–360, November 2006.

[50] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP*, 2003.

[51] Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. Quantifying the potential of program analysis peripherals. In *PACT*, 2009.

[52] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications. In *ASPLOS*, 2010.

[53] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993.

[54] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gun Sirer, and Fred B. Schneider. Device Driver Safety through a Reference Validation Mechanism. In *OSDI*, 2008.

[55] Xen PCI Passthrough. http://wiki.xen.org/wiki/XenPCIpassthrough, 2012.

[56] Min Xu, Vyascheslav Malyugin, Jeffery Sheldon, Ganesh Venkitachalam, and Boris Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Determinstic Replay. In *MoBS*, 2007.

[57] Yaun Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adapative Tracking. In *SOSP*, 2005.

[58] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: Safe and Recoverable Extensions using Language-Based Techniques. In *OSDI*, 2006.