Guardrail: A High Fidelity Approach to Protecting Hardware Devices from Buggy Drivers

Olatunji Ruwase[1] * Michael A. Kozuch[2] Phillip B. Gibbons[2] Todd C. Mowry[3]

[1]Microsoft Research [2]Intel Labs Pittsburgh [3]Carnegie Mellon University

olruwase@microsoft.com, {michael.a.kozuch, phillip.b.gibbons}@intel.com, tcm@cs.cmu.edu

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 evaluation of Guardrail shows that it can find serious data races, memory faults, and DMA faults in native Linux drivers that required fixes, including previously unknown bugs. Also, with hardware logging support, Guardrail can be used for online protection of persistent device state from driver bugs with at most 10% overhead on the end-to-end performance of most standard I/O workloads.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Reliability, availability, and serviceability; D.2.5 [*Software Engineering*]: Testing and Debugging—Monitors, Tracing

Keywords Device Drivers; Dynamic Analysis

* Work done at Carnegie Mellon University

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541970

1. Introduction

Because device drivers have been identified as a critical weak link in overall systems reliability [8, 17, 31, 39, 46], researchers have attempted to improve their robustness through strategies such as *static analysis* [1, 8, 21, 31], *specification* [26, 40, 50], *type safety* [35, 44, 54], *user-level drivers* [4, 18, 25, 50], and *run-time analysis* [4, 5, 14, 18, 45, 50]. While each of these approaches has its merits, we focus on *run-time analysis* in this paper because it is complementary to the other approaches and 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* [4, 5, 14, 18, 45, 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 any side effects outside of the driver (e.g., writing to kernel memory [45]).

1.1 Limitations of Existing Fault Isolation Techniques

While existing fault isolation techniques are useful, they suffer from two key limitations. First, they *only check invariants at the driver's interface*, treating 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.

Second, 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

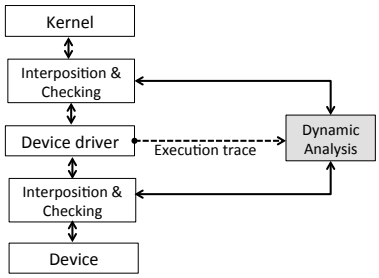


Figure 1: Incorporating decoupled dynamic analysis to protect the system from driver faults.

far more challenging to interpose and successfully check invariants across this latter interface [50].

1.2 Our Approach: Protecting Devices through Decoupled, Instruction-Grain Dynamic Driver Analysis

In this paper, we present a new framework (called *Guardrail*) that focuses on preventing buggy device drivers from corrupting hardware device state. Rather than treating the bulk of the driver execution as a black box, Guardrail'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. Hence Guardrail enables a more comprehensive analysis of whether or not the driver software is behaving *correctly* than what is practical today; for example, handling the case where a buggy driver stores the wrong value in a valid 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 [6, 47] or through binary instrumentation [15, 32]) and stored in a buffer that is consumed asynchronously by a dynamic analysis tool running concurrently in a separate virtual machine. 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). Although the dynamic analysis enabled by Guardrail does increase overall computational load on the system, our decoupled approach allows the bulk of this overhead to be offloaded onto idle CPUs (unless the system is already saturated on CPU throughput). Hence Guardrail is suitable for either a fast debug/testing environment or a production environment that is not already throughput limited.

1.3 Related Work

As described above, our work complements earlier research on *fault isolation* [4, 5, 14, 18, 45, 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. In contrast with the previous proposal for isolating *devices* from driver faults [50], which required modifying the driver and moving it into user-space, our approach is transparent to both the driver and the device, therefore enabling Guardrail to work with arbitrary driver binaries and devices.

Regarding dynamic checking for faults *within* drivers, SafeDrive [54] and KAddrcheck [15] perform run-time checks to detect memory addressability issues in kernel code, including drivers. In contrast with SafeDrive [54], which instruments drivers at compile-time, our approach works directly on binaries and does not require access to driver source code. In contrast with KAddrcheck [15], 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. For example, our *DMACheck* tool dynamically detects DMA bugs in drivers; this runtime approach is complimentary to a static analysis approach [3] detecting similar bugs using separation logic with permissions.

Finally, DataCollider [13] 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 Guardrail-enabled data race detection tool (*DRCheck*, 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 (because *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*.

1.4 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 kernel-mode driver binaries and devices for commodity operating systems.
- Within Guardrail, we demonstrate three new instruction-grain correctness checking tools that detect data races (*DRCheck*), DMA faults (*DMACheck*), and unsafe uses of uninitialized data (*DMCheck*), none of which are supported by existing driver fault isolation techniques. *DRCheck* improves upon prior approaches by minimizing false positives and avoiding false negatives, while handling the complexities of kernel-mode drivers.
- Our experimental results show that Guardrail is more effective at catching driver bugs than previous tools, e.g., by finding a bug in the popular *qla2xxx* SCSI driver that was undetected for years. Moreover, with hardware logging support, Guardrail modestly impacts the end-to-end performance of standard I/O workloads in most cases.

2. System Design

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

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

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

(containment) provide mechanisms for 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

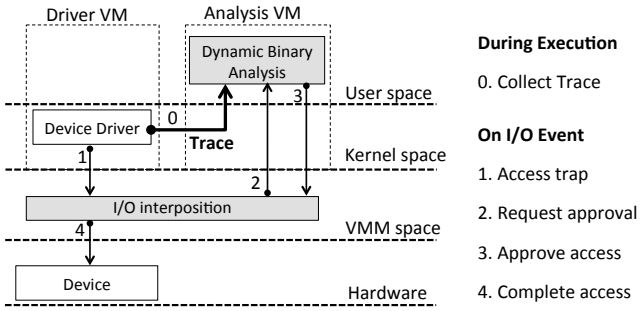


Figure 2: System architecture of Guardrail.

OS (Linux, in our prototype) and related applications, execute 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 propagating 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 a second virtual machine, the "Analysis VM." An instruction-level trace supporting high *detection fidelity* can be captured through one of several mechanisms: binary translation [15, 32] in the Driver VM, VMM-based monitoring [9, 52], or monitoring hardware [6, 47, 48]. Because driver code is potentially executed by an unbounded number of kernel threads, logical logs are maintained per virtual processor in the Driver VM, rather than per kernel thread, 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 (see Section 2.2 for details). 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 particular requirements of the user (e.g., willingness to sacrifice system availability to ensure persistent data integrity). Therefore, to accommodate 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, recording event information for additional off-line post-mortem analysis, taking a system checkpoint, enabling additional online analysis, etc.) Guardrail dispatches the offending I/O operation to the device and resumes normal execution. *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 [22, 27, 34]. 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, this paper 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 implemented through a simple addition to the VMM layer; less than 500 lines of C code were required to retrofit a commodity VMM (Xen [2]) with I/O interpositioning. The complexity of the checking tool may be non-trivial, however, primarily because the system was designed to accommodate arbitrary correctness-checking to cope with the wide variety of bug types that plague drivers [8, 17, 31]. 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 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

```
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 `hard_start_xmit()`.

all pairs of accesses in `hard_start_xmit()` by different threads with at least one writer.

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 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—fortunately, such changes are likely infrequent because they require corresponding modifications to the entire driver code base.

2.2 I/O Interposition Details

Because devices are controlled by reading/writing device registers, the interposition layer prevents driver errors from propagating beyond the VM boundary by: (i) intercepting all[2] device register accesses, (ii) coordinating with a decoupled correctness checker to determine the safety of the accesses, and (iii) ensuring their timely completion as soon as they are deemed safe. Because the Driver VM has direct access to the device [51], the interposition layer is transparent to both the driver and device, and therefore supports arbitrary drivers and devices. Figure 2 depicts the steps associated with the transparent handling of 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, Guardrail intercepts 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 ad-

[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.

[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 operations and were not pursued in this work.

dress. 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 [50], this can be relatively straightforward; however, in our decoupled checking approach, additional coordination is required between the interposition 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 occurred 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 [5, 24, 46].

Because the checker's response will typically incur some latency, the interposition layer has at least two options regarding 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 a device register 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) retrying the faulting instruction after temporarily making the device register available to the guest OS [10], or (ii) emulating the faulting instruction in the hypervisor. Because the concurrently executing kernel threads of commodity OSes share a single kernel address space, the first option requires great care in attempting to ensure that the temporarily accessible page is only accessed by the verified operation in the intended thread at an appropriate time. Consequently, in our current implementation, we chose the emulation option in order to avoid potential containment errors, especially in SMP environments.

[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.

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 [8, 17, 31, 39]. This section describes the three instruction-grain dynamic analysis tools that we developed for finding (i) data races, (ii) violations of OS rules for using DMA, and (iii) memory faults in unmodified Linux driver binaries.

3.1 DRCheck: Detecting Data Races

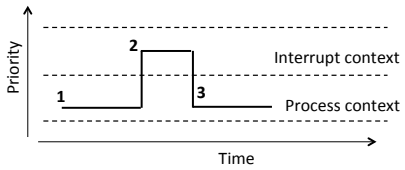
Our first dynamic analysis tool, *DRCheck*, detects data races in kernel-mode drivers. A *data race condition* occurs whenever there are two unserialized accesses to the same shared data with at least one being a write. Race conditions are difficult to avoid during driver development because of the complex concurrency setting in which drivers operate, and difficult to find during pre-release testing because of their nondeterministic nature. Moreover, most drivers are developed by third parties who are unlikely to be kernel experts [17, 31]. 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.

3.1.1 Complexity of concurrency issues in drivers

While there have been many studies on user-mode data race detection [16, 41, 42, 53], existing tools cannot be easily adapted for drivers, because the concurrency issues of kernel-mode execution are more complex than user-mode execution. For example, the LockSet algorithm [41], which associates a *lockset* state with each memory location, assumes that all synchronization occurs through well-defined library primitives. To integrate the LockSet techniques into *DRCheck*, any kernel synchronization mechanisms that do not use library primitives would need to be adapted to the framework. We identified the following four sources of additional complexity that must be addressed in kernel-mode driver execution:

1. a form of concurrency that makes it possible for a *single* thread to make racy accesses to shared data (i.e., to race itself);
2. synchronization invariants based on the context of the device state;
3. synchronization based on deferred execution using softirqs and kernel timers; and
4. *ad hoc* mutual exclusion techniques that avoid lock overheads, such as disabling interrupts and preemption.

These issues can lead to excessive false positives and false negatives for user-mode race detectors, as shown by our experimental study in Section 4. We discuss these concurrency issues in more details below, and then describe how



1. In ***process*** context (e.g. packet transmission)
2. Preempted to ***interrupt context*** to service NIC interrupt (e.g. packet reception)
3. Resume ***process*** context

Figure 4: A kernel thread executing network driver code in different Linux kernel contexts.

DRCheck addresses them to reduce false positives and avoid false negatives.

Sources of concurrency in drivers The most basic source of driver concurrency is multi-threaded execution of driver code that accesses shared data. In addition, a subtle form of concurrency is introduced by the multiple execution contexts of varying priority levels that are provided by commodity preemptive OS kernels, in order to enable scheduling flexibility for time-constrained, privileged work. For example, Linux kernel threads execute either in *process* context (lowest priority) or in *interrupt* context—further divided into *bottom half* and *top half* (highest priority). Kernel-mode execution contexts are critical to driver performance—they enable the prompt completion of higher priority tasks (e.g., interrupt handling) by hijacking a thread that is performing a lower priority task and using it for the higher priority task. As an example, Figure 4 illustrates the time line of a kernel thread executing network driver code in *process* and *interrupt* contexts of the Linux kernel. The thread is initially executing the packet transmission routine of the driver in *process* context, next it is switched to *top half* context to service a network card interrupt using the interrupt handling routine, and afterwards it resumes the packet transmission routine. However, execution contexts complicate concurrency in drivers in the sense that if the high priority code shares data with the suspended low priority code then the kernel thread could race itself.[4] For example, a race would occur in Figure 4 if on resumption of *process* context the thread reads data that was updated in *top half* context. Such a race will be missed by existing race detection tools because only one thread is involved.

Device state-based synchronization Many peripheral devices (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

[4] A restricted form of this issue arises in user-space due to signal handling (i.e., reentrancy), and to our knowledge has been ignored by prior work on user-space data race detection.

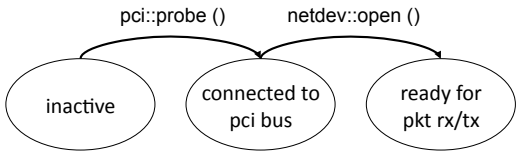


Figure 5: State transitions for a Linux PCI network device, showing that the `probe()` and `open()` functions of the driver are serialized.

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.

Deferred execution Kernel threads that execute under tight deadlines (e.g., interrupt service routines) often have 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 functions, such as checking that tasks are completed on schedule or that a device is still functional, until at least a specified time in future.

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. 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 registration 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.

Kernel-mode 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()`). However, other primitives, such as interrupt enabling/disabling and hardware atomic instructions (e.g., `test_and_set`) require more effort to detect because they are not accessed through modularized interfaces.

3.1.2 Addressing driver concurrency issues

Before describing how *DRCheck* addresses the concurrency issues of kernel-mode drivers, we first discuss how DataCollider [13] (discussed in Section 1.3) addresses these issues.

DataCollider 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.

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.

Detecting driver concurrency *DRCheck* uses thread identifier information to detect and disambiguate the concurrency associated with multi-threaded execution of driver code, similar to user-mode tools. To detect concurrency due to interleaved execution of different execution contexts by a single kernel thread, *DRCheck* also tracks the execution context of each kernel thread. Basically, just as memory operations of a user-mode thread are considered serialized, *DRCheck* considers 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 other contexts.

Detecting synchronization based on device state As discussed in Section 3.1.1, devices can be in certain disjoint states known to the kernel, and the kernel may (implicitly or explicitly) guarantee that certain driver entry points are only invoked in particular device states. Figure 6, for example, shows how the Linux kernel networking stack uses device states to guard the execution of the open() callback of a network driver. Consequently, driver code need not include explicit synchronization between two access if those accesses are known to only occur in particular states.

```
int dev_open (dev) {
  …

if (! test_bit(__LINK_STATE_PRESENT, &dev->state))
     return –ENODEV;
  …
 dev->open (dev)
  …
```

Figure 6: Snippet of Linux network code that uses device state to guard the open() callback of a network driver.

DRCheck detects the use of device state for synchronization by drivers by exploiting the fact that the kernel already leverages device state information to guard driver execution. For each driver callback, *DRCheck* dynamically maintains the set of all the device states in which the callback has been invoked. With this information, *DRCheck* can identify callbacks that are invoked under disjoint sets of device states. Because such callbacks cannot execute concurrently with one another, their accesses to shared data are implicitly serialized and therefore not races. An alternative to dynamically building the set of device states for each callback would be to obtain such information from kernel experts [12, 19], perhaps reducing runtime overhead. However, we adopted our dynamic discovery approach because it does not rely on the availability of correct specifications.

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. So far, we have 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 rather than for individual drivers.

Handling deferred execution Although Linux interrupt threads that defer work using *softirq*s do not share locks with the polling threads that will eventually do the work, they do invoke the polling threads using the `raise_softirq` call. Further, the Linux softirq infrastructure guarantees that only one polling thread, on the same processor as the interrupt thread, will respond to a given call and complete the deferred work. Hence, *DRCheck* recognizes the `raise_softirq` call as the serializing operation between threads.

For kernel timers, *DRCheck* associates a virtual state with each timer. A timer is *inactive* before its registration, and *active* until it executes, after which it becomes *inactive* again.

This serializes the execution of the timer to operations preceding its registration. Additionally, *DRCheck* associates a virtual lock with each timer that is held throughout the execution of the timer function. This serializes successive execution of the timer's function.

Detecting non-lock based mutual exclusion 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.

3.1.3 DRCheck implementation

DRCheck is an extension of the *Lockset* algorithm in Eraser [41]. *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, such as `spin_lock_irq`, also disable interrupts. 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 3.1.2. 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.

3.2 Direct Memory Access (DMA) Faults

Direct Memory Access (DMA) is a common technique for transferring data to and from devices without consuming CPU cycles. To detect incorrect DMA operations that might harm the system, we created a new tool within our Guardrail framework called *DMACheck*, which performs instruction-grain dynamic analysis of drivers to ensure that they correctly address the following issues related to *DMA buffers* (i.e. regions of system memory used in DMA transfers):

Sharing: Because DMA buffers are shared between the driver and the device, there is the potential for data races (e.g., writes by the driver into source DMA buffers could corrupt outgoing I/O data). Hence the driver should assume that the device has exclusive access during a transfer in order to avoid data corruption.

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 redundantly mapping or unmapping them.

Coherence: Because devices access DMA buffers directly (bypassing any caches) on non-coherent systems while drivers' accesses to DMA buffers can be cached, cache lines should never mix DMA buffer data with other types of data (including other DMA buffers). Proper alignment and padding of data can prevent this problem.

DMACheck detects DMA buffer bugs in Linux drivers by monitoring how they manipulate DMA buffers. Because Linux drivers operate on DMA buffers through both *virtual and physical* addresses (e.g., a driver reads or writes a DMA buffer using the virtual address, but synchronizes the cache and memory copies of the buffer using the physical address), *DMACheck* tracks the mapping of a DMA buffer in both the virtual and physical address spaces; this makes it unusual compared with other tools. (*DRCheck* and *DMCheck* (Section 3.3), for example, need to track only virtual addresses.) Although some DMA buffer bugs (e.g., misaligned DMA buffers) can be detected simply by inspecting the arguments that drivers use to make DMA function calls (e.g., `dma_map_single()`), *data races* require instruction-grain dynamic analysis to identify when driver accesses to a DMA buffer overlap with those from the device.

DMACheck detects races on DMA buffers by checking for unserialized accesses by the driver and device to a DMA buffer. One challenge is that device access to DMA buffers cannot be directly observed by *DMACheck*. To overcome this, *DMACheck* defines a time interval that includes the period during which a DMA buffer could be accessed by the device, and checks that no driver accesses are made to the

buffer during that interval. We identified two pairs of driver operations for defining this interval: (i) between mapping the buffer into the I/O address space and the corresponding unmapping, and (ii) between specifying the buffer as part of a DMA transfer to the device and the corresponding servicing of the completion interrupt. While the latter option may appear to provide a smaller correct interval, *DMACheck* uses the former, more conservative, option for two practical 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 be lost 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. Second, the latter option requires understanding the device-specific way that drivers setup DMA transfer—an unscalable undertaking, given the large number of available devices.

3.3 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* [28], 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.

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.[5]

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. This suggests that our techniques can be used to improve driver debugging and testing, and to 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 positives and false negatives that certain analysis tools may incur.

DRCheck is a tool that may incur false positives. The underlying *Lockset* algorithm of *DRCheck* leads to false data race reports for code that while properly synchronized, 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* [27] means that simply avoiding false alarms (e.g., by incorporating a *Happens-Before* approach [53]) is insufficient. However, rather than foregoing race detection entirely on production systems, we believe that Guardrail's *triage* mode (Section 2) can help, when armed with techniques seeking 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 improves its classifier and reduce the number of spurious alarms.

The dynamic nature of our techniques creates the possibility of false negatives—run-time analysis cannot guarantee driver correctness. Rather, our tools can determine only 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 because 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 prerelease purposes when combined with techniques for achieving high coverage driver execution [7, 36].

[5] 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.

I/O type	Benchmark	Description	Workload
Audio & Video	Mplayer	Multimedia player	Full HD movie trailer (i.e., 1920 x 1080p resolution, 24 fps)
Network	Apache	Webserver	16K requests for 40KB static page using 16 concurrent requests
	Memcache	In-memory key value store	256 client threads each making 100K *get* requests
	Netperf	Network perf. meter	20 secs run with 16KB msg for stream and 32B for request/response
Storage	GNU Make	Compilation utility	4-way parallel build of Linux 2.6.18 kernel with default configuration
	Postmark	Filesystem benchmark	100K transactions on 20K files of size range 10KB–20KB

Table 1: I/O intensive benchmarks and workload settings for evaluating Guardrail.

4. Evaluation

We evaluated our Guardrail prototype to answer two questions:

1. How effectively do our techniques detect driver faults, particularly when compared with existing techniques?
2. What is the impact on the system end-to-end performance, particularly if the monitored device is heavily used?

4.1 Methodology

The I/O interposition layer of our Guardrail prototype was implemented by extending paravirtualized Xen-3.3.1 VMM with our techniques for containing potential driver faults. The guest OS in the Driver and Analysis VMs was a 32 bit Fedora Core 6 OS, based on the Linux 2.6.18 kernel. We used a variety of audio, video, network and storage devices in our evaluation, along with the corresponding stock, unmodified Linux driver binaries. In all experiments, the device is directly assigned [51] to the Driver VM to minimize I/O virtualization overheads [11].

Benchmarks We generated driver workloads using a set of popular I/O benchmarks, listed in Table 1. We used the open source media player, *Mplayer*, to evaluate the audio and video drivers. We evaluated the network drivers using the *Apache* web server, the *Memcache* in-memory key-value store, and the *Netperf* network performance measurement tool. Network load for *Apache* and *Memcache* were generated using their respective benchmarking tools: *ApacheBench* and *Memslap*, while *Netperf* load was generated using its *stream* tests (TCP_STREAM & UDP_STREAM) and *request/response* tests (TCP_RR & UDP_RR). The storage drivers were evaluated using the *Postmark* filesystem benchmark and a kernel compilation workload. Table 1 shows the workload settings for the results in this paper.

Class	Driver	Device
Audio	snd_hda_intel	High Definition Audio (ICH7)
Network	tg3	Broadcom 5754 1Gpbs NIC
Storage	ahci	ICH7 SATA disk (200GB)
Video	nvidia	Quadro NVS 285

Table 2: Drivers and devices evaluated in real hardware.

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

Our evaluation of Guardrail on real hardware focused on the performance of virtualization-based I/O interpositioning and software-based instruction tracing of kernel-mode drivers. The devices and corresponding drivers that were used in these experiments are presented in Table 2. The test system for these experiments was a Dell Precision 390 workstation that had Dual-Core Intel Core 2 Duo processors running at 2.66 GHZ and with 2GB of physical memory. For the network experiments, the server ran on the test system while the client ran on a Dell Precision T3400 workstation with Quad-Core Intel Core 2 Extreme processors running at 3 GHz and with a 4GB physical memory. The client system was running 32-bit Ubuntu 10 (2.6.32 kernel) Linux OS. The client and server systems were in the same local area network so that network latency was negligible in our results.

For our Guardrail evaluation on simulated hardware, we used the Simics [43] full system simulator (academic package, version 4.0.63) to model hardware-assisted instruction-level streaming of a driver's execution from the Driver VM to the Analysis VM. We used these experiments to study the bug detection effectiveness of Guardrail and the end-to-end performance of online protection of I/O operations from buggy drivers. Our kernel-mode instruction tracing hardware is based on previous hardware proposals [6, 48] for tracing user-mode execution. We carefully chose the simulation parameters illustrated in Table 3a to derive a similar environment to the real hardware platform. The lack of audio and video device models in our simulation package restricted our evaluation to the network and storage drivers and devices shown in Table 3b.

4.2 Fault Detection

We evaluated how Guardrail improves driver bug detection by using the framework to implement our proposed dynamic binary analysis tools: (i) *DRCheck*, for detecting data races (Section 3.1), (ii) *DMACheck*, for detecting DMA faults (Section 3.2), and (iii) *DMCheck*, for detecting memory faults (Section 3.3). The results show that Guardrail enables better detection of driver bugs than previous approaches.

Parameter	Values used
Processors	Dual-Core, Intel Pentium 4, 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
Driver VM	2 VCPU, 1GB RAM
Analysis VM	1 VCPU, 512MB RAM

(a)

Class	Driver	Device
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

(b)

Table 3: The simulation parameters, and the drivers and devices used for simulation based evaluation.

Tool	Count
DRCheck	9
Det-DataCollider	2
Ideal-DataCollider	6

(a) Data races detected

Tool	Count
DataCollider	0
DRCheck	11
DeferExec	67
KLockset	111

(b) False data race alarms

Table 4: Races and false alarms reported by Guardrail.

Bug type	Count
Data race	7
Leak	4
Repeat map/unmap	4
Misaligned buffer	10

(a) DMA buffer bugs

Tool	Count
DMCheck	2
DDT	1
KAddrcheck	1
KMemcheck	2

(b) Memory bugs

Table 5: DMA buffer bugs detected by Guardrail by type, and memory bugs detected by different tools.

4.2.1 Data races

As shown in Table 4a, *DRCheck* found nine serious data races in five Linux drivers (six of which have either been confirmed or fixed). Also, using this table, we compare *DRCheck* with DataCollider based on the details in [13]. We made two assumptions in our analysis to increase the 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 occurred earlier (*Ideal-DataCollider*). With these assumptions, two races will be detected by *Det-DataCollider*, and six 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 4b. The number of false alarms, though, is an order of magnitude fewer than *KLockset*. *DeferExec*, which is *DRCheck* without state-based synchronizations (Section 3.1.2), falls in between. We were not able to compare with *DDT* [23], an existing data race detector for the Windows kernel, because it was not described in sufficient detail to enable comparisons.

4.2.2 DMA faults

The different DMA buffer faults found by *DMACheck* in six drivers are summarized in Table 5a. Races on DMA buffers, which are the most serious of these bugs, affected only the *tulip* network driver. *DMACheck* found seven unique driver writes (i.e., static instruction addresses) that could potentially corrupt I/O data being read by the network card. DMA buffers with unaligned virtual addresses (assuming 32 byte cache lines) are the most common fault type—affecting five drivers (i.e., *e100*, *e1000*, *pcnet32*, *tg3*, *tulip*). As discussed earlier, this bug is a serious issue in non-coherent systems. The *sym5c8xx* driver was the only one that leaked DMA buffers (i.e., failed to unmap DMA buffers before unloading), whereas *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, we did not observe any resulting system failures during our experiments.

4.2.3 Memory faults

As shown in Table 5b, *DMCheck* found two serious memory faults, 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 six years after the 2.6.18 version we used for our study. Because these bugs involve memory that is exclusively used by the driver, they cannot be detected using fault isolation techniques that only check driver interaction with the kernel [5, 18, 45, 49, 50, 54]. 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 5b to compare *DMCheck* against existing kernel-mode memory fault detectors for the Windows kernel (*DDT* [23]) and the Linux kernel (*KAddrcheck* [15], *KMemcheck* [30]). *DDT* and *KAddrcheck* track

memory addressability, and therefore can only detect the out-of-bounds bug. *KMemcheck*, on the other hand, tracks both memory addressability and initialization, and therefore detected both the memory faults.

4.2.4 Fault detection summary

In summary, our evaluation validated our thesis that instruction-grained dynamic analysis can be used to improve the reliability of device drivers by detecting bugs in their execution. Guardrail's instruction-grained dynamic analysis enables detection of a significant number of hard-to-find bugs in production Linux drivers (i.e., data races, DMA buffer faults, and memory faults) that are missed by other tools, including a previously unknown buffer overflow in the *qla2xxx* storage driver. Guardrail sometimes incurs a small number of false positives, e.g., for data race detection. Guardrail's support for a variety of sophisticated checking tools demonstrates its value as a general-purpose framework, in contrast to fault-specific tools such as *DataCollider*.

4.3 Instruction-Grained Tracing Performance

Guardrail's instruction-grained tracing of driver execution can be implemented using software or hardware techniques. To evaluate the performance of a purely software approach, we obtained an early version of the *Granary* binary instrumentation framework for OS kernels [20] from the authors. *Granary* can instrument individual kernel modules rather than the entire kernel. In the version we used, *Granary* employs a trap-based mechanism to toggle instrumentation as execution switches between the kernel and the instrumented module. We modified *Granary* to stream the execution trace of a module, containing program counter values, instruction opcodes, and effective addresses, into a 256KB per-CPU circular buffer[6]. We refer to our *Granary* modifications as *Granary-Trace*.

We used the *tg3* driver to measure the impact of *Granary-Trace* on server throughput and CPU utilization. We compared against running the server on a native Linux system (i.e., *Linux*), and on a *Granary* system with no instrumentation (i.e., *Granary-Null*). For this experiment, the server system was running a 64 bit Linux 3.8.2 kernel version because *Granary* was implemented in that OS. The results, normalized to *Linux*, are presented in Figure 7. We observed that *Granary-Trace* noticeably impacts both throughput and CPU utilization in most cases. Even *Apache*, which suffers no throughput loss, more than doubles its CPU consumption. Other benchmarks were impacted more significantly. For example, *Memcache*[7] suffered a 60% reduction in throughput, while the *Netperf* consumed 4.6x and over 10x more CPU cycles for TCP and UDP streaming, respectively. Although, *Granary-Null* outperforms *Granary-Trace*, it still performs poorly in some cases, such as a 23% reduction in *Memcache* throughput and a 4.4x increase in CPU utilization for TCP streaming. The results show that while the overheads of a software-based implementation of Guardrail instruction tracing are acceptable in test environments, hardware-assisted instruction tracing is needed for practical deployment of Guardrail in production environments.

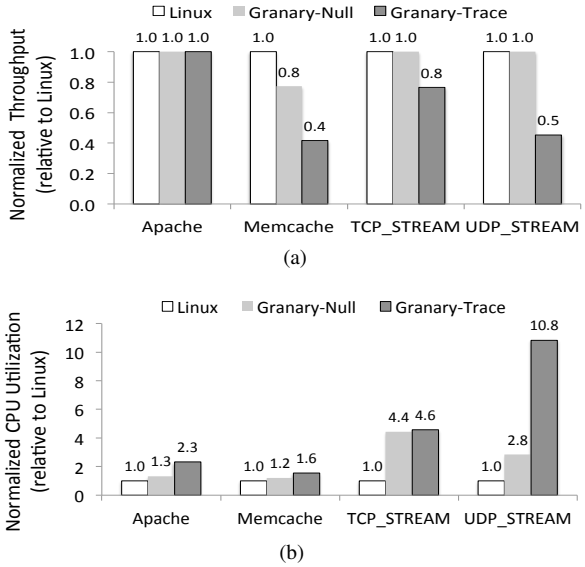


Figure 7: Impact of software-based instruction tracing on (a) server throughput and (b) CPU utilization.

4.4 I/O Interposition Performance

We studied the impact of Guardrail's I/O interposition on audio, video, network and storage I/O performance. For convenience, we use the following naming convention to report the results. *Linux* means a non-virtualized Linux system and is the baseline in all experiments. *Xen* means *Linux* as a guest OS on Xen VMM with directly assigned physical devices (i.e., only CPU and memory are virtualized). *IO-Interpose* is *Xen* with Guardrail's I/O interposition enhancements. Unless stated otherwise, the reported results are the median of 10 runs.

4.4.1 Audio and video performance

We used *Mplayer* to measure the impact of *IO-Interpose* on the audio and video quality of multimedia movie playback using the workload shown in Table 1. *Mplayer* was configured to use the *ALSA* audio output and the *X11* video output modes. The results are summarized in Table 6, and show that the playback was of similar audio and video quality on

	Time (s)	Frame Rate	CPU (%)
Linux	150.51	23.94	33
Xen	150.52	23.93	35
IO-Interpose	150.52	23.91	36

Table 6: Impact of I/O interposition on movie playback.

[6] Trace records were overwritten when the buffer filled up.

[7] 16 client threads were used because of robustness issues in *Granary*.

Linux, *Xen*, and *IO-Interpose*, with virtually the same frame rates achieved on all 3 systems. The CPU utilization of *IO-Interpose* was 3% higher than *Linux* and 1% higher than *Xen*. Overall, these results suggest that I/O interpositioning is unlikely to degrade the user experience of modern multimedia playback in any noticeable way.

4.4.2 Network performance

We used *Apache*, *Memcache*, and *Netperf*, and the workload settings in Table 1 to study the impact of *IO-Interpose* on server throughput or transaction rate (for *request/response*). Figure 8 reports server performance normalized to *Linux*. In most cases, *IO-Interpose* imposes at most 8% overhead on server performance. Since *Xen* performs similarly in these situations, most of the overheads could be attributed to it. The exception is *Memcache*, which loses 38% of its throughput on *IO-Interpose*. *Xen* appears to be responsible for over half of this degradation. CPU utilization for *Apache* (*Memcache*) was 35% (85%) on *Linux*, 52% (89%) on *Xen*, and 58% (92%) on *IO-Interpose*.

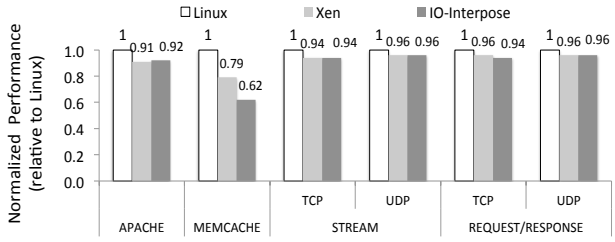


Figure 8: Impact of I/O interposition on network performace.

Memcache's exceptionally poor performance can be attributed to (i) frequent device register accesses and (ii) high CPU utilization in *Linux*. *Memcache*'s device register access rate (184K/sec) is at least an order of magnitude higher than the other benchmarks, which means "trap-and-emulate" overheads are incurred more frequently. Furthermore, high CPU utilization means there are fewer idle CPU cycles available to handle the frequent "trap-and-emulate" operations.

4.4.3 Storage performance

We used the GNU *Make* compilation utility and the *Postmark* benchmark with the workload settings in Table 1 to evaluate how I/O interpositioning affects disk storage performance. The results, in terms of normalized performance relative to *Linux*, are reported in Figure 9. Compilation is 32% slower on *IO-Interpose* compared to *Linux*. Also, *IO-Interpose* consumes about 93% of the CPU, which is about 8% higher than *Linux*. On the other hand, *Xen*'s performance is identical to *IO-Interpose*, which suggests that in this case most of *IO-Interpose* overheads are due to *Xen* (i.e., CPU and memory virtualization). For *Postmark*, the read, write, and transaction rates of *IO-Interpose* are 9–10% lower than *Linux*. Again, the performance of *Xen* is similar to *IO-Interpose*. The CPU utilization of *Linux*, *Xen*, and *IO-Interpose* were 7%, 11%, and 8% respectively.

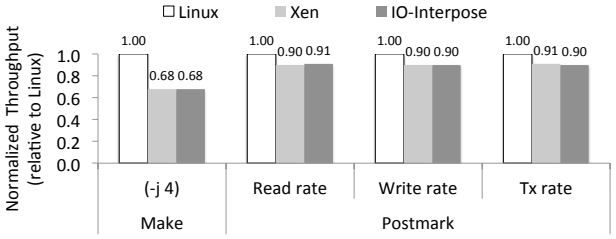


Figure 9: Impact of I/O interposition on storage performance.

4.4.4 I/O interposition performance summary

Our evaluation shows that Guardrail's I/O interposition layer imposed at most 10% overheads on common I/O workloads in most cases, and that code compilation (32%) and *Memcache* (40%) were the exceptions. All of the code compilation overheads and over half of the *Memcache* overheads could be attributed to CPU and memory virtualization. Finally, the combination of *Memcache*'s frequent device register accesses and high CPU utilization in Linux is a worst-case scenario for Guardrail.

4.5 End-to-End Performance

We evaluated the end-to-end performance impact on common I/O workloads of using Guardrail for online protection of persistent device state from defective drivers. We ran Guardrail in *permissive* mode (Section 2) to ensure that the experiments ran to completion without being abruptly halted for detected driver faults. We conducted this experiment in simulation so as to study the benefits of hardware logging support. We specifically measured the performance of I/O workloads while the corresponding driver is being monitored by the Guardrail checking tools—*DRCheck*, *DMACheck*, and *DMCheck*—normalized to the workload's performance without driver monitoring in a non-virtualized Linux system. Robustness issues of the device models forced us to reduce the workload size and study only the *tg3* and *sym53c8xx* drivers.

4.5.1 Network performance

Apache received a total of 1600 requests, *Memcache* was loaded by 16 client threads issuing 1K *get* requests each, and *Netperf* runs for 5 seconds. The results are presented in Figure 10. With the exception of network streaming using TCP and UDP, Guardrail imposed modest overheads on the workloads ($\leq$ 6%). However, the throughput reduction for TCP streaming was up to 60% (*DMCheck*) and for UDP streaming, up to 53% (*DMCheck*). The high rate of device register accesses (up to 300K/sec) of network streaming caused these high overheads, because the driver had to be stalled for the checking tool to catch up.

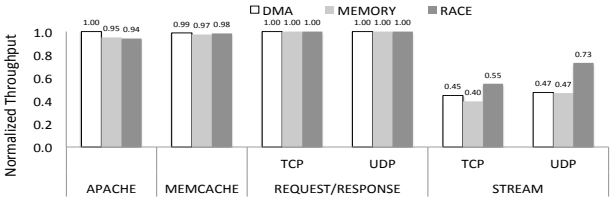


Figure 10: Network performance with Guardrail protection.

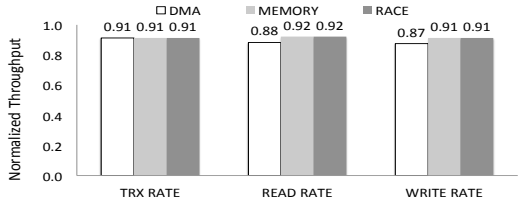


Figure 11: *Postmark* performance with Guardrail protection.

4.5.2 Storage performance

Postmark's workload setting was configured as in Table 1, except with 1K input files. The normalized transaction, read, and write rates of the benchmark with Guardrail monitoring are reported in Figure 11. The overheads were less than 10% in most cases. The sole exceptions were a 12–13% reduction in read and write rates for *DMACheck*. The relatively better performance compared to network streaming is because the device register access rate of *Postmark* is orders of magnitude lower (i.e., 3K/sec.).

4.5.3 End-to-end performance summary

Our experiments showed that online protection of the persistent state of I/O devices from subtle driver bugs (e.g., memory faults, data races) can be achieved with minimal impact on the 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 software [29, 33, 37, 38] and hardware [6, 48] techniques for accelerating dynamic analysis.

5. Conclusion

While device driver code is both (i) critical to proper system operation and (ii) more susceptible to bugs than other system software, relatively little work has been done in the area of online driver correctness monitoring (perhaps due to the performance-sensitive nature of driver software). The results of this paper demonstrate that decoupled correctness-checking together with VM-based I/O interpositioning can provide a high performance driver monitoring framework that achieves each of our system goals: generality, detection fidelity, containment, response flexibility, and trustworthiness. Guardrail represents a promising direction in driver correctness-checking that provides better safety not only for system devices, but also for the user data entrusted to those devices.

Acknowledgments. We are grateful to Peter Goodman, Angela Demke Brown and Ashvin Goel from the University of Toronto for providing us an early copy of Granary for our evaluation. We also thank Michael Swift (University of Wisconsin-Madison) and Brad Chen (Google) for their valuable input on this research. This work is supported in part by a grant from the National Science Foundation and by the Intel Science and Technology Center for Cloud Computing.

References

[1] T. Ball, E. Buonimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustunner. Thorough Static Analysis of Device Drivers. In *EuroSys*, 2006.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and The Art of Virtualization. In *SOSP*, 2003.

[3] M. Botincan, M. Dodds, A. F. Donaldson, and M. J. Parkinson. Safe Asynchronous Multicore Memory Operations. In *ASE*, 2011.

[4] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *USENIX*, 2010.

[5] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donelly, P. Barham, and R. Black. Fast Byte-Granularity Software Fault Isolation. In *SOSP*, 2009.

[6] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-grain Program Monitoring. In *ISCA*, 2008.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *ASPLOS*, 2011.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *SOSP*, 2001.

[9] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX*, 2008.

[10] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *CCS*, 2008.

[11] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High Performance Network Virtualization with SR-IOV. In *HPCA*, 2010.

[12] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules using System-specific, Programmer-written Compiler Extensions. In *OSDI*, 2000.

[13] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race Detection for the Kernel. In *OSDI*, 2010.

[14] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software Guards for System Address Spaces. In *OSDI*, 2006.

[15] P. Feiner, A. D. Brown, and A. Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *ASPLOS*, 2012.

[16] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[17] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP Kernel Crash Analysis. In *LISA*, 2006.

[18] V. Ganapathy, M. Renzelmann, A. Balakrishnan, M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In *ASPLOS*, 2008.

[19] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs. In *ASPLOS*, 2011.

[20] P. Goodman, A. Kumar, A. D. Brown, and A. Goel. Granary: A Sane Framework for Instrumenting an Insane Environment. Manuscript, 2013.

[21] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating Hardware Device Failures in Software. In *SOSP*, 2009.

[22] B. Kasikci, C. Zamfir, and G. Candea. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*, 2012.

[23] V. Kuznetsov, V. Chipounov, and G. Candea. Testing Closed-Source Binary Device Drivers with DDT. In *USENIX*, 2010.

[24] A. Lenharth, V. S. Adve, and S. T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *ASPLOS*, 2009.

[25] B. Leslie, P. Chubb, N. Fitzroy-dale, S. Gtz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level Device Drivers: Achieved Performance. *J. Computer Science and Technology*, 20, 2005.

[26] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *OSDI*, 2000.

[27] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *PLDI*, 2007.

[28] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, 2007.

[29] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing Security Checks on Commodity Hardware. In *ASPLOS*, 2008.

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

[31] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten Years Later. In *ASPLOS*, 2011.

[32] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *CGO*, 2010.

[33] 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.

[34] V. Raychev, M. Vechev, and M. Sridharan. Effective Race Detection for Event-driven Programs. In *OOPSLA*, 2013.

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

[36] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing Drivers without Devices. In *OSDI*, 2012.

[37] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing Dynamic Information Flow Tracking. In *SPAA*, 2008.

[38] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry. Decoupled Lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking Tools. In *PLDI*, 2010.

[39] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming Device Drivers. In *EuroSys*, 2009.

[40] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic Device Driver Synthesis with Termite. In *SOSP*, 2009.

[41] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Race Detector for Multithreaded Programs. *ACM TOCS*, 15(4), 1997.

[42] K. Serebryany and T. Iskhodzhanov. ThreadSanitzer - Data Race Detection in Practice. In *WBIA*, 2009.

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

[44] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the Starting Problem: Device Drivers as Self-describing Artifacts. In *Eurosys*, 2006.

[45] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP*, 2003.

[46] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. *ACM TOCS*, 24(4), 2006.

[47] M. Tiwari, S. Mysore, and T. Sherwood. Quantifying the Potential of Program Analysis Peripherals. In *PACT*, 2009.

[48] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications. In *ASPLOS*, 2010.

[49] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *SOSP*, 1993.

[50] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device Driver Safety through a Reference Validation Mechanism. In *OSDI*, 2008.

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

[52] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Determinstic Replay. In *MoBS*, 2007.

[53] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adapative Tracking. In *SOSP*, 2005.

[54] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *OSDI*, 2006.