Improving Device Driver Reliability through Decoupled Dynamic Binary Analyses

Olatunji O. Ruwase

CMU-CS-13-114

May 2013

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Todd C. Mowry, Chair
David Andersen
Onur Mutlu
Michael Swift, University of Wisconsin
Brad Chen, Google

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by Google, Intel, and the National Science Foundation under grant numbers CNS-0720790, CCF-1116898, CNS-1065112, and CCR-0219931.

Keywords: Device Drivers, I/O, Operating Systems, Virtualization, Dynamic Analysis, Reliability, Computer Architecture

For Molara and Anjola.

Abstract

Device drivers are Operating Systems (OS) extensions that enable the use of I/O devices in computing systems. However, studies have identified drivers as an Achilles' heel of system reliability, their high fault rate accounting for a significant portion of system failures. Consequently, significant effort has been directed towards improving system robustness by protecting system components (e.g., OS kernel, I/O devices, etc.) from the harmful effects of driver faults. In contrast to prior techniques which focused on preventing unsafe driver interactions (e.g., with the OS kernel), my thesis is that checking a driver's execution for correctness violations results in the detection and mitigation of more faults.

To validate this thesis, I present Guardrail, a flexible and powerful framework that enables instruction-grained dynamic analysis (e.g., data race detection) of unmodified kernel-mode driver binaries to safeguard I/O operations and devices from driver faults. Guardrail decouples the analysis tool from driver execution to improve performance, and runs it in user-space to simplify the deployment of new tools. Moreover, Guardrail leverages virtualization to be transparent to both the driver and device, and enable support for arbitrary driver/device combinations.

To demonstrate Guardrail's generality, I implemented three novel dynamic checking tools within the framework for detecting memory faults, data races and DMA faults in drivers. These tools found 25 serious bugs, including previously unknown bugs, in Linux storage and network drivers. Some of the bugs existed in several Linux (and driver) releases, suggesting their elusiveness to existing approaches. Guardrail easily detected these bugs using common driver workloads.

Finally, I present an evaluation of using Guardrail to protect network and storage I/O operations from memory faults, data races and DMA faults in drivers. The results show that with hardware-assisted logging for decoupling the heavyweight analyses from driver execution, standard I/O workloads generally experienced negligible slowdown on their end-to-end performance.

In conclusion, Guardrail's high fidelity fault detection and efficient monitoring performance makes it a promising approach for improving the resilience of computing systems to the wide variety of driver faults.

Acknowledgments

This dissertation could not have been completed without the selfless sacrifices of my family, friends, and academic/research colleagues and mentors who consistently prioritized my success over their personal comforts. To these folks, I owe a debt of gratitude that I can neither fully express nor adequately repay.

Throughout my seven years at Carnegie Mellon, Todd Mowry has been my advisor, advocate, teacher, friend, and strong supporter. His fervent belief in my abilities is perhaps only matched by his willingness to devote tremendous time and effort into mentoring me to become a good researcher. Todd never skipped an opportunity to promote my work to the academic and industry world, and I have benefited greatly as a result.

This thesis document that (hopefully) you are about to read would not have been possible without the willingness of David Andersen, Onur Mutlu, Brad Chen, and Mike Swift to serve on my committee. I am truly honored that these "super stars" were always available and remained highly engaged from start to finish. Beyond their committee responsibilities, they also took great interest in my career development by providing verbal and written recommendations.

My research experience benefited greatly through collaborations with Phil Gibbons, Mike Kozuch, and Shimin Chen at Intel Labs Pittsburgh. From identifying interesting research problems, all the way to the evaluating and submitting proposed solutions for peer review, Phil, Mike, and Shimin enthusiastically combined their exceptional individual research expertise into advancing my research endeavors. But their support went beyond just research, Phil, Mike, and Shimin generously served as fellowship and internship mentors, wrote reference letters, and con-

nected me to the key players in the industry for my career advancement. I was also fortunate to often interact with Michael Kaminsky, Padmanabhan (Babu) Pillai, and Mei Chen at the Intel Labs.

Michelle Goodstein, Theodoros Strigkos, and Evangelos Vlachos were my fellow student-collaborators on the Log Based Architectures (LBA) project. Together we enjoyed (and endured) the unique experience of hacking simulators to demonstrate how much better the world would be if only our "magic" LBA hardware is commercially adopted. Through the often harrowing debugging sessions, the nervous and seemingly endless waits for successful completion of simulation runs, the crushing disappointment of paper rejections, and the absolute euphoria of publications, an incredible bond of friendship, that is truly one of the highlights of my time in Pittsburgh, was forged. Babak Falsafi, Anastasia Ailamaki, Greg Ganger, Vijaya Ramachandran, and Michael Ryan were significant contributors to the LBA project. Vivek Seshadri, Gennady Pekhimenko, and Debby Katz are post-LBA members of my research group and have been a delight to relate and work with.

Beyond my research group, I was fortunate to make wonderful friends at CMU within and beyond the Computer Science department. Amar Phanishayee, Fahad Dogar, and Himabindu Pucha are my close friends and fellow hackers on the extremely fun "Ditto" project. I had a lot of fun playing soccer and winning intramural titles with the SCS and ECE folks. I enjoyed interacting with Michael Papamichael, Nels Beckman, Robert Simmons, Michael Ashley-Rollman, Vijay Vasudevan, Hormoz Zarnani, Robert Rwebangira, Ayorkor Korsah, Israel Owusu, Emmanuel Wornyoh, Tawanna Dillahunt, amongst many others.

Frank Pfenning, Bob Harper, David Eckhard, Roger Dannenberg, Stephen Brooks, Mor Harchol-Balter, and David Brumley were faculty members who I was privileged to interact regularly with, and who positively influenced my graduate school experience. Life in graduate school would have been much more difficult, but for the exceptional CSD staff, including Sharon Burks, Deb Cavlovich, Diana Hyde, Catherine Copetas, Jenn Landefeld, Sophie Parks,

and others. These life savers, having aptly recognized the tendency of overworked PhD students to routinely "drop the ball" on important issues, never tired to proactively rescue me over and over again.

Outside CMU, the Redeemed Christian Church of God, Living Spring International Center, Pittsburgh, under the leadership of Pastors Gboyega and Kemi Esan, has been a constant source of love, warmth, encouragement, and spiritual refreshing. Sele and Caroline Odigie, Deji and Ibukun Alajo, Azeez Otori, Akuoma Alade, and Kechy Eke were ever present sources of friendship and encouragement. The Gbadebos lovingly accepted me into their family as a son and a brother, rather than as an "-in-law". Daddy Gbadebo, I wish you were still here to celebrate, along with Mummy Gbadebo, Kemi, Oyinkan and Olumide, this moment. My parents, Mr. and Mrs. B.P. Ruwase gave me life, and continue to give love, support, and encouragement in all of my endeavors. None of accomplishments could have been possible without them. My sister, Tobi, has always been a steady source of love and support. To my cousins, uncles, aunts, and grandparents, too numerous to name, thanks for your prayers and words of encouragement. A special shout-out to Drs. Toks Fashola and Yemi Adekoya for making me believe that I could grow up to be just like you.

Despite the huge amount of time and energies that went into this thesis, the fact of the matter is that my greatest accomplishment occurred years earlier when Molara agreed to be my wife. Molara, I cannot express how much your love, companionship, devotion, and unyielding belief in me has meant to me through this journey, especially in the darkest moments when I have doubted my ability to conclude my program. You and Anjola have been my most important supporters and motivation. I love you very much.

The length of this acknowledgment section might make it difficult for the reader to believe that it is in fact very terse; significantly lacking in details and names. I apologize to all whose contributions have been diminished by my brevity. Be rest assured that in my heart your contributions remain greatly and forever cherished. God bless you all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Device drivers are critical pieces of system software that manage input/output (I/O) devices in computing systems (e.g., Embedded, Personal Computers, Data Centers, etc.). Drivers enable users to enjoy the rich variety of functionalities that I/O devices offer, including persistent data storage (e.g., optical, magnetic, and flash drives), Internet connectivity (e..g, network cards and web cams) and entertainment (e.g., sound and graphics cards). Due to the privileged role they play, drivers are commonly deployed as extensions of commodity operating systems (e.g., Linux, Mac OS X, Solaris, and Windows) to enable convenient interaction with the OS kernel and the I/O device. However, studies [19, 32, 65, 73] have indicated that defective drivers are the Achilles' heel of system reliability especially in production environments where drivers account for a significant portion of system failures.

Consequently, significant research has been devoted into dynamic approaches for mitigating the risk of using defective drivers in production systems, by preventing the corruption of trusted system components (e.g., OS kernel and I/O devices) [14, 28, 29, 80, 81, 86, 90]. However, existing approaches are limited in two crucial ways. First, most techniques focus only on faults that can be observed during driver interaction with the rest of the system [14, 29, 80, 86]. In

particular, they interpose on the driver's interface to determine the safety of driver operations with external side-effects (e.g., modifications of kernel memory). Such techniques are unable to protect the system from faults that occur within the driver. Second, the vulnerability of persistent I/O operations and devices to driver faults has received little attention [86] compared to the significant number of studies on protecting the OS kernel [14, 28, 29, 80, 81, 90]. This is unfortunate since a faulty driver can permanently damage a device [22, 86], or compromise critical system functions (e.g., virtual memory, filesystems, etc.) that rely on persistent I/O functionality.

This thesis demonstrates that the aforementioned limitations of existing driver fault mitigation techniques can be addressed through *decoupled* instruction-by-instruction correctness checking of driver execution. Instruction-grain dynamic analysis of drivers enables the detection of a wider range of driver bugs than current approaches and provides useful diagnostics information to aid fixes. Decoupling enables this improved fault detection fidelity without incurring high performance overheads, making it possible for production systems to efficiently tolerate driver bugs. We propose a more powerful framework for improving driver reliability, called *Guardrail*, which protects persistent I/O device state and I/O operations from defective drivers through decoupled dynamic analysis of drivers and mediation of their I/O operations.

1.2 Improving Driver Reliability without Dynamic Analysis

The pervasive use of computing systems today, especially for mission-critical tasks (e.g., air traffic control), underscores the importance of system reliability. Unfortunately, since systems software such as device drivers are error-prone, improving the reliability of production systems remains a subject of intense research.

Complementing the focus of this thesis on dynamic approaches are proposals for improving the quality of driver code without executing the driver. These include avoiding driver faults by design, and statically identifying faults in driver code. By not requiring driver execution, runtime overheads, and the burden of constructing the driver's execution environment (i.e., OS kernel and

I/O device) are avoided. Unfortunately, the complexity of real-world drivers [40, 73] (e.g., multi-threading) limits the effectiveness of tackling driver faults without executing the driver, which makes dynamic approaches important for driver reliability. However, by reducing the bug-rate of drivers, avoiding and statically detecting driver bugs offer the following practical benefits. First, system failure rates are reduced to a point where the corresponding device can be reasonably useful to customers. Second, the overheads of dynamic techniques can be avoided for portions of driver code, where faults have been completely avoided or statically identified (and removed). Third, the assumption that faults are rare in production drivers can be exploited to reduce the overheads of dynamic techniques. For example, decoupled dynamic analysis assumes that faults are rare enough that the higher overheads of synchronous dynamic analysis can be safely avoided.

1.2.1 Avoiding Driver Faults by Design

The technical complexity of modern drivers, coupled with the error-prone manner in which they developed, make driver errors inevitable. Studies have shown that the poor code quality of drivers stems from the fact that they are written: (i) manually [49, 64], (ii) by programmers who lack expert knowledge of the OS [9, 19, 65] and/or device [52, 73], and (iii) in unsafe programming languages (C, C++) [73]. These problems have motivated proposals to improve driver quality through type safe programming languages [11, 35, 39, 53, 69], high level specification languages [52, 64, 74], and automatic reverse engineering [17].

Safe Programming Languages Commodity OS drivers are generally written in C and C++, and therefore suffer from type safety issues (e.g., memory leaks, pointer errors, uninitialized memory use, etc.) that are associated with low-level programming. An obvious approach to avoiding this class of errors is to use type safe languages for driver development. However, the main reason for writing drivers in unsafe languages is because they must execute as modules of commodity OS kernels that are themselves written in such languages. Current approaches

for avoiding type errors during driver development include writing the OS in a type safe language [11, 39, 53], and moving drivers into user-space [69].

Proposals for using type safe languages to implement operating systems include the following: (i) SPINOS [11] (written in Modula-3), (ii) JavaOS [53] and JX [35] (written in Java), and (iii) Singularity [39] (written in Shing#). The drivers for these operating systems are free of type errors by construction since they are also written in the type safe language. However, for various reasons, including performance and legacy code compatibility, these ideas have not been adopted for commodity OS development, and thus offer little practical value to the reliability of production systems.

Decaf [69] offers an alternative approach for writing drivers in a safe language without breaking compatibility with commodity operating systems code. Decaf is an extension to the Microdriver [33] approach of partitioning a driver into a *kernel-mode component*, containing the performance-critical and privileged code paths, and a *user-mode component*, containing other parts of the driver. Decaf employs static analysis and multi-lingual programming techniques to enable a safe language implementation of the user-mode component of Microdriver (Java in this case). DriverSlicer static analysis [33] is used to identify driver code portions that can be reliably moved into user-mode, while the Jeannie compiler [38] is used to manage the cross-language interactions between the user-mode Java portions of the driver and the C portions that remain in kernel-mode. Although, Decaf helps to avoid type errors in large portions of drivers, system reliability remains vulnerable to errors in the kernel-mode portions.

Specification Languages Beyond high level programming of drivers are the techniques that employ even higher level specification languages to automate error-prone procedures during driver development. These include tasks that require significant manual effort (e.g., driver maintenance) and expert knowledge of the OS and device (e.g., writing driver interfaces). As discussed below, studies have identified such activities as significant sources of driver faults.

Changes to the internal libraries and APIs of an OS kernel often requires updating client code

in the kernel—including drivers—to maintain compatibility. However, the number of drivers that exist in most commodity operating systems range in the thousands [19, 54, 65] hence updating them is a burdensome task that often results in widespread breakage of driver code, due to either copy-paste errors or failure to update all the required drivers [49, 63]. Cocinelle [64] automatically applies programmer-provided patches across the entire driver code base, thereby relieving the programmer from the manual effort that is ordinarily required. The desired changes are described by the programmer using a specification language—called Semantic Patch Language (SmPL)—provided by Cocinelle. With this specification, Cocinelle systematically identifies driver codes that require updating, and applies the patch in a semantically-aware manner, thus avoiding common patching errors. However, Cocinelle neither guarantees the correctness of the provided patch, nor identifies existing errors in the updated drivers.

Implementing correct interface logic for drivers is notoriously difficult, primarily because of the required OS and device expertise. Devil [52] and Termite [74] automatically synthesize driver interface code from high-level descriptions of the OS or device protocol, thus avoiding errors of manually written code. The underlying idea is that it is much easier for a device vendor to write the correct formal specifications of the device protocol than for an average driver developer to implement a correct interface in C. Hence the burden for interface correctness shifts from the programmer to the specifications provided by OS and device experts, as well as the synthesis tool. This approach is limited by a lack of correct and formal specifications of operating systems and hardware devices. Moreover, other error-prone driver features like multi-threading and memory management cannot be synthesized using these techniques because of their complexity. Consequently, fully synthesized production drivers are not yet available.

Reverse Engineering Porting proprietary binary drivers is important for making devices available to operating systems that are not supported by their hardware vendor. In reality, however, it is a labor-intensive and error-prone exercise that suffers from the lack of device specifications. To mitigate the difficulty of porting drivers, RevNIC [17] reverse-engineers device protocol logic

from proprietary drivers in the supported OS, producing C code that a developer can use in the target driver. Although RevNIC avoids new bugs in the ported code, it does not eliminate existing bugs, which are simply replicated in the target driver. Moreover, portions of the target driver that interact with the OS (e.g., concurrency and memory management) cannot be ported, and are still manually written.

Fault avoidance summary Fault avoidance reflects the belief that programmers are the weak link in driver development, and that automation can improve driver quality by reducing the possibilities of programmer error. Safe languages relieve the programmer from the burden of ensuring type safety in drivers. Specification languages go a step further by relieving programmers from the burden of implementation, leaving them instead with the task of specification. Reverse engineering goes to the extreme by completely removing programmers from the development process by synthesizing driver code entirely from existing implementations. Unfortunately, existing techniques cannot fully automate the development of modern drivers. Hence detection of driver faults is still required for system reliability.

1.2.2 Detecting Driver Faults through Static Analysis

Since current driver development techniques cannot guarantee the absence of faults in driver code, identifying and removing faults is an important technique for improving the quality of production drivers. One approach is to apply static analysis techniques to detect correctness violations in driver code [7, 9, 26, 27, 65]. Although static checking improves the quality of released drivers by preventing many bugs from escaping into production environments, it cannot guarantee driver correctness due to driver complexity. Specifically, static checking attempts to examine all of the possible states of a driver—along all execution paths—for errors. However, the complexity of real-word drivers makes exhaustive state exploration intractable. Therefore, mechanisms for tolerating the remaining faults are required in production systems.

Existing static checkers have identified a variety of faults in driver code including: (i) violations of OS kernel API rules [7, 9, 27, 65], (ii) concurrency management errors [26], and (iii) unsafe assumptions about device behavior [41]. However, the precision of static checking can be impaired by pointer aliasing issues of C/C++, leading to both false positives and false negatives. *Iterative refinement* [21, 45] has been proposed to reduce false positives [7, 9], while incorporation of dynamic checking has been proposed to avoid false negatives [57].

A common strategy for managing the complexity of large systems software (e.g. operating systems) is to define system-specific rules (that go beyond programming language rules) for developers to adhere to. For example, Linux kernel programming forbids the following: floating point operations, blocking (or sleeping) in non-preemptible contexts, using user-space pointers without validating them, etc. Similar rules exists in other commodity OS kernels (e.g., Windows [9]). Unfortunately, these rules are often confusing, poorly documented, and not systematically enforced, which introduces reliability issues for commodity operating systems, since many drivers are developed and distributed independently of the OS vendor [54].

Meta-level compilation (MC) enables systematic enforcement of Linux kernel rules, through the *Metal* language (for specifying rules as compiler analyses) and the *xgcc* framework (for context-sensitive and inter-procedural analysis) [37]. MC extensions found hundreds of errors in kernel code (including drivers) [27]. Static Driver Verifier (SDV) detects similar violations of Windows kernel rules in drivers [9]. SDV uses SLAM [7]—a static analysis engine—to create a boolean-program abstraction of the driver automatically from source code [8] and analyze the abstraction for errors using model checking. Users express the rules to be checked in a specification language called SLIC [6]. SLAM iteratively refines the driver abstraction to reduce false positives. SDV found hundreds of errors in widely used Windows drivers.

RacerX [26] statically detects data race conditions and deadlocks in kernel code (including drivers) using flow-sensitive, context-sensitive and inter-procedural dataflow analysis. RacerX relies on developer annotations to help infer: (i) what locks protect what shared data, what code

contexts are subject to multi-threaded execution, and (iii) the system-specific synchronization functions (including preemption control). False positives are mitigated by using system-specific heuristics to rank detected errors based on severity. RacerX found serious errors in commodity operating systems including Linux, FreeBSD, and an unnamed, large commercial OS.

Although hardware devices sometimes do fail or misbehave, drivers often assume otherwise, which results in system hangs or crashes. Two common examples of such faults in drivers are that they fail to validate inputs from the device, and infinitely wait for a device response. Carburizer [41] is a static analysis tool that checks for instances of such unsafe assumptions in driver source code. Carburizer identified almost a 1000 such assumptions or bugs in Linux drivers, with a false positive rate of less than eight percent.

1.3 Related Work

Techniques for improving system reliability by monitoring driver execution to detect driver faults, and mitigate their impact on the system have been well studied [12, 14, 20, 28, 29, 30, 33, 46, 70, 75, 80, 85, 86, 90]. Dynamic techniques leverage runtime information about driver behavior (e.g., thread interleaving, control flow, etc.) to precisely identify subtle faults (e.g., data races) that are difficult to avoid by design or detect statically. Moreover, dynamic techniques can be effectively used on unmodified driver binaries (i.e., *dynamic binary analysis*) unlike static checking, which requires the availability of the driver source code. Unlike static approaches, dynamic checking: (i) requires executing the driver in its normal environment (i.e., with OS kernel and I/O devices), (ii) incurs performance overheads, and (iii) detects faults only in the executed code paths.

Techniques that dynamically check driver execution can be used (by the developer or user) for testing drivers and making production systems robust to defective drivers. The effectiveness of adopting a dynamic approach in these scenarios is significantly influenced by two factors: (i) whether driver operations (e.g., instructions) are checked for correctness before or after they

complete (i.e. *timeliness*) and (ii) whether checking is applied to all or some portion of driver operations (i.e. *coverage*). These two factors determine the following properties of a driver monitoring technique: (i) the impact on driver performance, (ii) fault detection fidelity, and (iii) fault containment (e.g., OS kernel protection). Another relevant but orthogonal deployment consideration for using dynamic approaches is the ability to handle unmodified driver binaries. We examine these issues in more details below.

The timing of correctness checks during driver execution is an important design consideration for a dynamic technique as it affects both performance and fault containment. One approach is to guard potentially faulty driver operations with the appropriate checking operations, that are performed synchronously (a.k.a. *coupled* checking). Alternatively, checking operations could be decoupled from the driver and performed asynchronously with the driver execution (i.e. decoupled checking). Decoupled checking typically lags driver execution, especially when multiple checking instructions (or cycles) are required for each driver instruction.

The timeliness of performing correctness checks on driver execution directly affects performance because it determines whether or not checking overheads are incurred on the critical execution paths of the driver. The impact of coupled checking on driver execution varies depending on the sophistication of the dynamic analysis—the number of checking instructions required for each driver operation. For lightweight analyses (e.g., for memory safety), which require less than ten instructions to check each driver operation, coupled checking could simply slow down the driver. However, for heavyweight analyses (e.g., data race detection), which require tens of instructions to check each driver operation, coupled checking could actually break the driver due to severe perturbations of timing-sensitive computations (.e.g, interrupt handling) [28]. In contrast, decoupled checking reduces (or eliminates) the impact of dynamic analysis on driver performance to the point where heavyweight correctness checking can be comprehensively performed on driver execution without robustness problems. Moreover, techniques like DISE [23] and Speck [60] can be used to further accelerate decoupled checking.

Timeliness also affects the ability to contain the driver faults that manifest during execution, and prevent them from corrupting the rest of the system. The ability of contain driver faults is critical to helping systems survive driver failures. In particular, driver faults should be prevented from propagating to system components such as the OS kernel and I/O devices which interact frequently with drivers. Coupled checking makes it relatively easy to provide strong fault containment guarantees since faulty driver operations are preemptively identified and prevented from executing. In contrast, decoupled checking makes the containment of driver faults more challenging. This is because due to the lagging checks, faulty driver operations are not detected until much later in the execution. However, in the meantime the driver continues to execute in a faulty mode with the freedom to interact with, and potentially compromise other parts of the system (e.g., the OS kernel).

Thus, coupled and decoupled checking present performance and fault containment tradeoffs to driver monitoring techniques. Coupling enables strong fault containment guarantees at the expense of poor performance, while decoupling enables good performance, but makes fault containment more difficult. As illustrated in Table 1.1, all but one of the existing dynamic techniques adopt a coupled checking approach; Aftersight [20] is the sole exception. Aftersight is also the only technique that does not protect the system from driver faults. The other techniques protect the OS kernel from defective drivers and Nexus-RVM [86] additionally protects persistent I/O state in the system.

Another important design consideration for driver monitoring is that of checking coverage, i.e. what portion of driver execution to check for correctness. This issue impacts both the performance and fault detection fidelity of a dynamic technique. While, checking the entire driver execution guarantees that any fault that is exercised will be detected, it unfortunately incurs the maximum checking overhead. In contrast, checking only portions of driver execution helps to reduce the checking overhead, but does so at the risk of missing bugs that are exercised in the unchecked portions. Moreover, subtle bugs (e.g., unsafe use of uninitialized data) that require

instruction-grain information flow tracking, through registers and memory locations, cannot be reliably detected by partially checking driver execution.

The classification of existing dynamic techniques based on whether they achieve total or partial coverage of driver execution is illustrated in Table 1.1. Only five techniques (Aftersight, DataCollider [28], DDT [46], KAddrCheck [30], and SafeDrive [29] check the entire driver execution for bugs. The remaining eight techniques check only a portion of driver execution, specifically execution of the interfaces to the OS kernel and I/O devices (only Nexus-RVM). Thus, driver bugs that corrupt its internal state such as overflowing or racing on internal buffers can only be detected by those five techniques.

The ability to monitor unmodified driver binaries is important in production environments because driver source code may not be available. Drivers (e.g., graphics drivers) are sometimes distributed in only binary format for proprietary and convenience reasons. Majority of the existing techniques employ *dynamic binary analysis* to monitor driver execution making them effective for unmodified driver binaries. As illustrated in Table 1.1, proposals that work on driver binaries include SFI [85], Nooks [80], XFI [90], Aftersight, DataCollider, DDT, and KAddrCheck. The other dynamic techniques require the availability of driver source code.

Dynamic Analysis Summary Table 1.1 summarizes our discussion on the attributes of existing dynamic driver analysis techniques which are relevant to deployment in test and production environments. An ideal technique for driver testing should analyze the entire execution to increase the chances of detecting faults and be efficient enough to enable heavyweight analysis of interrupt handlers. While the first requirement is met by Aftersight, DataCollider, DDT, KAddrCheck, and SafeDrive, the second is met by only Aftersight through decoupled analysis. However, Aftersight cannot protect the OS kernel or I/O operations from defective drivers which makes it unsuitable for production use and demonstrates the key challenge of decoupled checking, i.e. fault containment. On the other hand, existing techniques which contain driver faults are problematic for deployment in production environments either because of the poor performance

	Yes	No
Couples checking with execution	BGI, DataCollider, DDT, KAddrCheck, Microdrivers, Nexus-RVM, Nooks, SafeDrive, SFI, SUD, SymDrive, XFI	Aftersight
Protects OS kernel	BGI, DataCollider, DDT, KAddrCheck, Microdrivers, Nexus-RVM, Nooks, SafeDrive, SFI, SUD, SymDrive, XFI	Aftersight
Protects persistent I/O state	Nexus-RVM	Aftersight, BGI, DataCollider, DDT, KAddrCheck, Microdrivers, Nooks, SafeDrive, SFI, SUD, SymDrive, XFI
Checks all driver execution	Aftersight, DataCollider, DDT, KAddrCheck, SafeDrive	BGI, Microdrivers, Nexus-RVM, Nooks, SFI, SUD, SymDrive, XFI
Works on binaries	Aftersight, DataCollider, DDT, KAddrCheck, Nooks, SFI, SUD, XFI	BGI, Microdrivers, Nexus-RVM, SafeDrive, SymDrive

Table 1.1: Classification of dynamic techniques for improving driver reliability.

of coupled instruction-grained analysis or the poor fault detection of interface checking.

1.4 Research Goals

The high-level research goal of this thesis is to demonstrate the following statement:

Decoupling can effectively address the limitations of dynamic approaches to driver faults by enabling sophisticated instruction-grain correctness checking without incurring the high performance overheads of current proposals. Also, commodity virtualization can be leveraged to enable fault containment for decoupled monitoring so that sophisticated dynamic analysis can be deployed on production systems to efficiently mitigate the risks of defective drivers.

1.5 Contributions

This thesis 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 the persistent state of hardware devices). In contrast to previous proposals, Guardrail performs instruction-grain correctness checking as the driver executes; it also uses a decoupled VM-based approach to provide efficient and transparent protection from driver faults. Guardrail supports arbitrary kernel-mode driver binaries and devices in commodity operating systems.
- Within our Guardrail framework, we demonstrate instruction-grain correctness checking tools that detect accesses to uninitialized data, data races, and DMA faults (none of which is supported by existing driver fault mitigation 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 proposed correctness-checking tools are more effective at catching driver bugs than previous approaches (e.g., finding a bug in the popular *qla2xxx* SCSI driver that had eluded detection for years). Moreover, our results show that with hardware-assisted execution tracing Guardrail can safeguard network and

storage I/O operations from driver faults with minimal impact on the end-to-end performance of most I/O workloads, with network streaming (up to 60% throughput reduction) as the exception.

1.6 Thesis Organization

This thesis demonstrates how decoupled instruction-grained dynamic binary analysis can improve the detection and mitigation of driver bugs beyond current techniques.

Chapter 2 provides a brief background on device drivers and can be skipped by readers that are familiar with the material. This is followed in Chapter 3 by a high-level description of Guardrail, our proposed framework for using decoupled dynamic analysis to safeguard I/O operations from bugs in kernel-mode drivers. Chapter 4 presents three novel Guardrail-enabled instruction-grained dynamic analyses for finding data races, DMA bugs, and memory bugs in unmodified driver binaries.

Because Guardrail decouples dynamic analysis from driver execution to improve performance, correctness checking can lag the driver execution substantially, especially for heavyweight checkers. Thus, Chapter 5 describes how Guardrail transparently interposes on the I/O operations of the driver to enable validation of those operations by the decoupled checking tool.

Chapter 6 describes how Guardrail decouples dynamic analysis from driver execution for online mitigation of driver bugs, and evaluates the impact of Guardrail on the end-to-end performance of network and storage I/O workloads. Chapter 7 concludes and summarizes the key points of this thesis.

Chapter 2

Background on Device Drivers

Device drivers are a class of system software that manage the peripheral I/O devices in computing systems in order to meet the I/O needs of users. Common functionality that drivers enable include persistent data storage through hard disk drives, Internet connectivity through network cards, and gaming through graphics cards. To provide this functionality, a driver receives I/O requests (e.g., sending a network packet, disk read, etc.) from the operating system kernel and performs the appropriate set of device operations to satisfy the requests. For example, a network driver copies outgoing network packets from system memory onto the network card, and then manipulates the network card to effect actual transmission of the packets across the network. Thus, as illustrated by Figure 2.1, the driver is the interface between the operating system kernel and the device. Due to the privileged nature of driver functionality, commodity operating systems, such as Linux and Windows, typically implement drivers as dynamically loaded modules of the kernel so that the drivers can efficiently and conveniently interact with the operating system and the device.

Commodity operating systems provide device access to user-level software (e.g., applications) through layers of kernel-level software collectively known as an *I/O protocol stack*. Thus, a driver executes as part of the I/O protocol stack of its device. As an example, Figure 2.2 shows the Linux protocol stacks for a network and a storage device. As shown in Figure 2.2, drivers operate as the lowest layer of the protocol stack that interacts directly with the device.

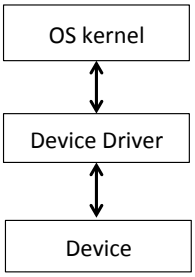


Figure 2.1: A device driver is the interface between the operating system and an I/O device.

2.1 Driver Interaction with Operating System Kernel

A major part of driver computation involves interaction with the operating system kernel [40, 73]. The driver receives I/O requests (e.g., from applications) and sends the corresponding responses through the OS kernel. Moreover, the OS kernel provides critical system resources that the driver needs for computation, such as memory, interrupt lines, and the system bus. Since the driver is a module in the address space of the OS kernel, it interacts with the kernel through CPU registers, physical memory, and functions.

The OS kernel forwards I/O requests to the driver by calling the corresponding driver functions (a.k.a. *callbacks*). To enable this, the driver registers callbacks for performing various device related operations with OS kernel (at load time). For example, a network driver registers callbacks for transmitting packets, retrieving network card statistics, configuring the network card, etc. To reduce the burden of driver support in kernel development, commodity operating systems group drivers into classes (e.g., network, disk, etc.) and export a uniform interface to each class for maintaining state and registering callbacks. For example, the Linux network driver interface includes `struct net_device` data type for state maintenance and `register_netdev()` function for callback registration.

To allow drivers to request and manage system resources (e.g., memory), the OS kernel exports a variety functions, such as `kmalloc()` for allocating memory, `request_irq()` for reserving interrupt lines, etc.

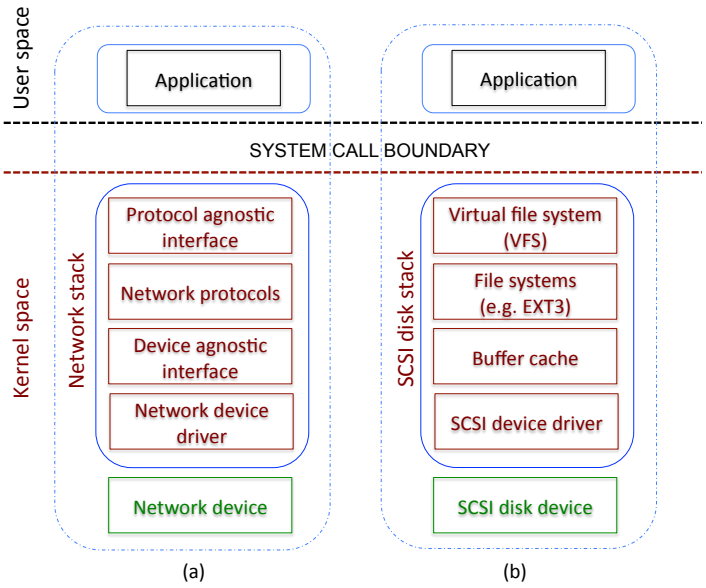


Figure 2.2: Drivers in the Linux I/O protocol stacks for (a) network and (b) SCSI disk devices.

2.2 Driver Interaction with Device

Driver interaction with a device is another significant portion of driver computation [40, 73]. Conceptually, a driver maintains two communication channels with the device, one for *control* and another for *data*. The control channel is used to configure the device, obtain status information, issue I/O commands, and receive device interrupts. The data channel is used to transfer I/O data between device and system memory. However, the manner in which the driver communicates with the device is influenced by the processor architecture, the OS kernel, and the I/O subsystem. Guardrail was designed for Linux-based x86 systems with *PCI*-based I/O system bus and so the following discussion applies to such systems. However, similar concepts apply to other system platforms.

Control communication typically occurs through the device's registers. Device registers are mapped into three distinct x86 address spaces, where they are accessed by the driver. The I/O related address spaces are the I/O port space, memory mapped I/O space (MMIO), and PCI configuration space (`PCI-config`). Typically, there is a single 64KB I/O port space that is

shared among devices in the system and 256 bytes of `PCI-config` space per device. On the other hand, MMIO space is device-specific and of variable size.

Device registers play a critical role in reliable interrupt delivery. Each device is configured to use a specific physical interrupt line for signaling of interrupts. However, because there is a limited number of physical interrupt lines in a system, interrupt lines are sometimes shared by a number of devices. Thus, a device that is using a shared interrupt line is additionally configured to set a designated device register whenever it signals an interrupt. The driver checks the status of the designated device register to confirm the origin of an interrupt received on a shared line and to respond appropriately.

Device registers differ in how they are accessed by software. While I/O ports can only be accessed through privileged `IN` and `OUT` instructions (e.g `inb`, `outb`), MMIO and PCI-config are accessed through regular loads and stores, and thus offer more programming flexibility. Unlike the physical memory space and the processor register space, reading or writing a device register can cause side effects on the device.

The dominant mode of data communication between drivers and devices is through *Direct Memory Access* (a.k.a DMA), where the device transfers I/O data (e.g., network packets, disk blocks, etc.) directly to/from physical memory buffers (a.k.a DMA buffers) without using CPU cycles. DMA is initiated by the driver programming specific device registers with the desired data transfer parameters. The data transfer could involve a single DMA buffer, multiple DMA buffers in physically contiguous memory, or multiple DMA buffers in non-contiguous memory (*scatter-gather* transfer). As part of initiating a DMA operation, the driver configures the device registers with pointer value(s) that correspond to the DMA buffer(s). For scatter-gather transfers, pointers to the DMA buffers could be stored in a region of system memory known as a *DMA descriptor*. IOMMU hardware [1, 4] is increasingly used to tackle the addressing and reliability challenges of DMA.

Chapter 3

Guardrail overview

Guardrail enables online protection of the I/O operations of a system from the harmful effects of driver faults. In this chapter, we provide an overview of Guardrail's approach to mitigating driver faults, while the integral components of Guardrail are described in more detail in subsequent chapters. The following discussion starts with a motivation of Guardrail's decoupled approach to detecting driver faults before presenting a high-level design of the framework.

3.1 Guardrail Approach to Mitigating Driver Faults

Mitigating the impact of driver faults on system integrity generally involves: (i) identifying faulty behavior in driver execution (i.e. *fault detection*), and (ii) executing the driver in a separate fault domain from the protected system components (i.e. *fault containment*). However, as discussed in Section 1.3, current proposals minimize the performance overheads of runtime analysis by checking only portions of driver execution—specifically the driver interface(s)—rather than the entire execution. In particular, driver faults are detected and contained through correctness checks that are interposed on the driver's interfaces to other system components (e.g., OS kernel). As a result, the internal computation of a driver, which accounts for the bulk of its execution, is essentially treated as a black box, as illustrated in Figure 3.1(a), and the bugs which exist there remain

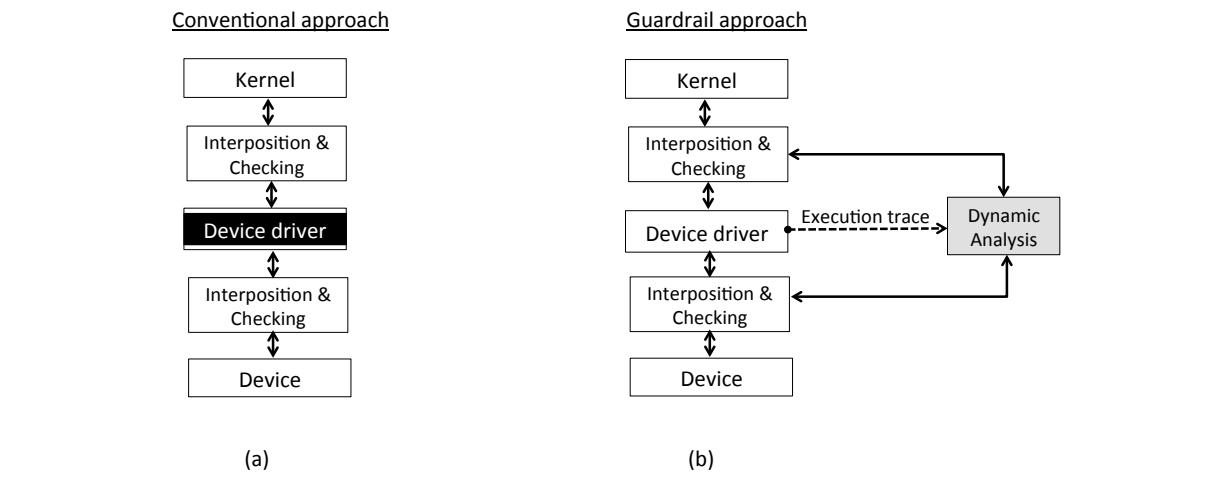


Figure 3.1: Comparing conventional and Guardrail approaches to mitigating driver faults.

a threat to system integrity.

Rather than ignoring the bulk of driver execution for correctness checks, Guardrail proposes a more powerful approach. In this approach, the interposition layer's decision of whether to allow the driver to proceed with a side-effect-causing operation is driven not only by invariant checks at the driver's interface, but also by instruction-grain dynamic analysis of the driver software as it executes, as illustrated in Figure 3.1(b). Indeed, Guardrail typically identifies correctness problems within the driver before they 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 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 a 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 [16, 82] or through binary instrumentation [30, 67]), and stored in a buffer that is consumed asynchronously by a dynamic

analysis tool that runs concurrently on a separate core from the monitored driver. 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 can catch up.

Guardrail effectively achieves a "sweet spot" between *synchronous* instruction-grain analysis (that 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 (that avoids runtime overhead, but occurs too late to prevent faulty drivers from corrupting persistent state).

We now present an overview of Guardrail design.

3.2 System Design

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

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 upon detecting an error (e.g., disable I/O operations from the driver, or simply record information for post-mortem analysis).

Trustworthiness: rely on a minimal trusted computing base for containment.

The system architecture that resulted from these goals is shown in Figure 3.2. To simultaneously satisfy the *containment* and *generality* goals, we adopted a virtual machine-based system.

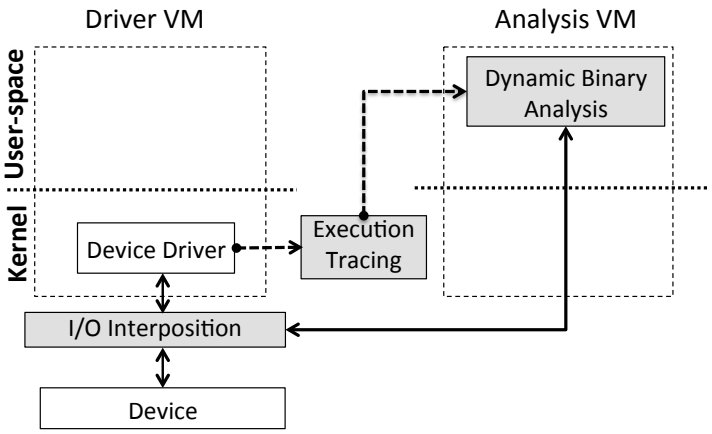


Figure 3.2: The system architecture of Guardrail.

The driver(s) of interest, along with the stock 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 by the interposition layer, and if is 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 the driver's 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 [30, 67] in the driver VM, VMM-based monitoring [20, 88], or monitoring hardware [16, 82, 84]. The execution trace is streamed (possibly with some buffering delay) to the Dynamic Binary Analysis tool, which runs in the 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 preferences of the user (e.g., their willingness to sacrifice system availability to ensure persistent data integrity).

To accommodate their preferences, end users may configure Guardrail to operate in one of three modes: (i) *stringent*, (ii) *permissive*, and (iii) *triage*.

***Stringent*:** In *stringent* mode, Guardrail blocks the intercepted and subsequent I/O operations from the driver, which effectively disables the I/O device.

***Permissive*:** *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 resulting system failures.

***Triage*:** *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 [42, 56]. If the I/O operation is deemed safe, Guardrail behaves as if in *permissive* mode, otherwise it behaves as if it were in *stringent* mode.

Although this flexibility allows Guardrail to be configured in interesting ways for different, real-world, deployment scenarios, this thesis is focused on *stringent* Guardrail because it presents the most challenging performance issues.[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 realized through a simple addition to the VMM layer. Less than 500 lines of C code were required to retrofit a commodity VMM (Xen [10]) with I/O interpositioning.

[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.

3.3 Analysis Scope

An important question that arises in our design is: which instructions in the driver VM should be traced for inspection by the analysis tool? For example, Guardrail could log all the instructions that are executed in the driver VM, only the kernel-level instructions, or just the instructions of the monitored driver. Tracing more instructions than is necessary to identify driver faults is undesirable because it incurs avoidable performance overhead.

Although, logging only the driver's instructions might appear to be sufficient, we soon discovered that this was not the case. This is because drivers execute as part of the protocol stack of I/O devices, as illustrated in Figure 2.2 and I/O protocol stacks provide certain invariants that the driver writer may rely upon. For example, the Linux network stack acquires certain locks before executing driver code to protect shared data accesses within the driver, as illustrated by the code snippet from Linux 2.6.18 in Figure 3.3. As we see in Figure 3.3, 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 because it happens outside the driver context; hence it would incorrectly flag as data races all pairs of accesses in `hard_start_xmit()` by different threads with at least one writer.

Guardrail addresses this issue by tracing the execution of certain kernel functions (e.g., for synchronization) by other parts of the I/O protocol stack, outside the driver. This allows the analysis tool to identify operations outside the driver context that are relevant to its correct behavior. For example, Guardrail logs the synchronization operations (e.g., lock/unlock) which are performed by the *scsi_mod* module in the Linux SCSI I/O protocol stack to identify the critical sections within a SCSI driver for data race detection. While a possible drawback is that interface changes across kernel versions will require corresponding modifications to our checking tools, such changes are unlikely to occur frequently because they often require corresponding modifications to the entire driver code base, and to kernel analysis tools. Nevertheless, this extension is critical for avoiding false data race and memory fault reports in checking tools.

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

Figure 3.3: Serialization of packet transmission by the Linux interface to network drivers.

Chapter 4

Detecting Driver Bugs through Dynamic Binary Analysis

A key feature of Guardrail is that it performs instruction-grain analysis of the driver code as it executes to identify software bugs. In this chapter, we describe three new correctness-checking tools implemented in our framework for detecting: (i) data races, (ii) direct memory access (DMA) faults and (iii) memory faults in unmodified driver binaries.

Our discussion in this chapter is organized as follows. First, we provide some background on using dynamic binary analysis to improve software reliability, and justify our focus on this particular set of correctness issues in drivers. Next, we describe the design and implementation of each tool, starting with the data race checker, then the DMA fault checker, and finally the memory fault checker. We then present our evaluation of the bug detection effectiveness of the checking tools using production Linux drivers. In particular, we compare against state-of-the-art techniques by evaluating whether those techniques can detect similar driver bugs effectively. The performance evaluations of our checking tools in the context of the Guardrail framework are described in Chapter 6. Finally, we discuss some issues to consider when deploying Guardrail in test and production environments, and then examine the bug-detection limitations of Guardrail.

4.1 Background on Dynamic Analysis and Driver Bugs

Researchers have shown that instruction-by-instruction analysis of program execution, for correctness violations, is a powerful approach for identifying hard-to-find software bugs (e.g., race conditions, uninitialized memory use, and security vulnerabilities). Consequently, instruction-grain dynamic analysis has emerged as an important technique for improving software reliability at the user-level. As demonstrated in Chapter 6, a major benefit of Guardrail's decoupled approach is that it enables similarly sophisticated analyses to be efficiently applied on driver execution in the kernel-space.

4.1.1 Dynamic Binary Analysis

Techniques for analyzing unmodified application binaries—dynamic binary analysis (a.k.a. *DBA*)—in particular have gained particular importance as applications are often available (or distributed) only in binary form, for legacy, proprietary, or convenience reasons. As a result, *DBA* techniques have been proposed to detect a variety of correctness issues in unmodified application binaries, including low level issues such as memory errors [58] and security vulnerabilities [59], and high level issues such as concurrency [31, 76, 89] and multi-lingual program interface errors [47]. Spurred by the popularity of *DBA*, researchers have proposed software [13, 51, 58] and hardware-based [23, 24] frameworks to reduce the burden of developing (and deploying) efficient *DBA* tools.

Inspired by the success of *DBA* in user-mode execution, this thesis explored how to improve OS reliability by using *DBA* to improve the mitigation of kernel-mode driver faults. *DBA* is a promising approach for addressing the high bug-rate of production drivers for the following reasons. First, the variety of user-level correctness issues (e.g., memory, security, concurrency, interface) for which *DBA* has been effectively used mirror the major categories of production driver bugs (i.e., memory, concurrency, device interaction, OS interaction). Second, since *DBA* is effective on unmodified software binaries, it is widely applicable even when the driver is

only available in binary form (e.g., proprietary drivers). Finally, when applied at instruction granularity, *DBA* offers sufficient fidelity for precise detection of the most elusive types of bugs in driver execution, and provides useful diagnostics for fixing such errors.

4.1.2 Can User-mode DBA work for Kernel-mode Drivers?

Since application and driver development share similar software correctness concerns (e.g., memory, security, concurrency), it is natural to wonder whether existing *DBA* techniques—that were developed for user-mode execution—could be easily adapted for kernel-mode drivers. Such adaptation would be greatly appealing, if all that is required are modest changes to account for interface differences between the user-level and kernel-level system libraries (e.g., for memory and concurrency management). The potential savings in software engineering effort and time by reusing the many existing user-mode tools for kernel-mode drivers makes this an important question to consider.

Our investigation revealed that the answer to this question depends on whether the relevant correctness property (e.g., memory safety) is treated similarly by user-mode and kernel-mode executions. The three driver correctness issues that we discuss in this chapter represent a broad spectrum of possibilities. For example, applications and drivers manage virtual memory in similar ways; both rely on system library functions to allocate (e.g., `malloc()`/`kmalloc()`) and deallocate (e.g., `free()`/`kfree()`) memory. Therefore, the analyses for detecting memory faults (e.g., unallocated memory access) in applications and drivers are somewhat similar, as discussed in Section 4.4. In contrast, concurrency management is significantly more challenging in the kernel-space, compared to the user-space (e.g., due to interrupts). Therefore, as discussed in Section 4.2, extending existing user-mode data race detectors (e.g., *Lockset* [76], *Happens-Before* [31]) to detect driver races is impractical. Moreover, DMA is a privileged operation, so the correctness issues are only relevant in the kernel-space.

4.1.3 Why The Studied Driver Bugs are Interesting

Although studies have shown that production drivers suffer from a wide range of correctness issues [19, 32, 65, 73], we decided to focus on data races, DMA faults, and memory faults in this thesis for two main reasons. First, this set of bugs reflect the key findings of those studies in terms of the major root causes of driver bugs. The studies broadly classified driver bugs, based on the underlying driver property, into: (i) type safety (e.g., memory faults), (ii) concurrency (e.g., data races), (iii) OS protocol (e.g., DMA faults), and (iv) device protocol issues (e.g., misconfiguring the device). Thus, our study addresses three of the four categories of driver bugs, and would be effective for device protocol bugs if provided the relevant (proprietary) hardware information [75]. Second, our checking tools incorporate a variety of sophisticated techniques, such as *dynamic information flow tracking* [79], and *Lockset* [76], and therefore show that Guardrail can implement similarly sophisticated dynamic analyses (e.g., *taint tracking* [59]). Furthermore, to our knowledge, our third analysis represents the first use of instruction-grained dynamic analysis for DMA faults in drivers.

4.2 Detecting Data Races in Drivers

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 they are difficult to find during pre-release stress testing because of their non-deterministic nature [55, 66, 77]. Moreover, most drivers are developed by third parties who are unlikely to be experts in kernel synchronization mechanisms [32, 65]. As modern OS kernels and their drivers increasingly exploit parallelism to improve performance, avoiding race conditions becomes all the more challenging and poses a serious threat to system reliability.

Our focus is on detecting race conditions on driver data between kernel threads executing driver code. While there have been many studies on user-mode race detectors [31, 76, 78, 89], this prior work cannot be easily adapted for kernel-mode drivers. This is because the concurrency issues of kernel-mode driver execution are 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.

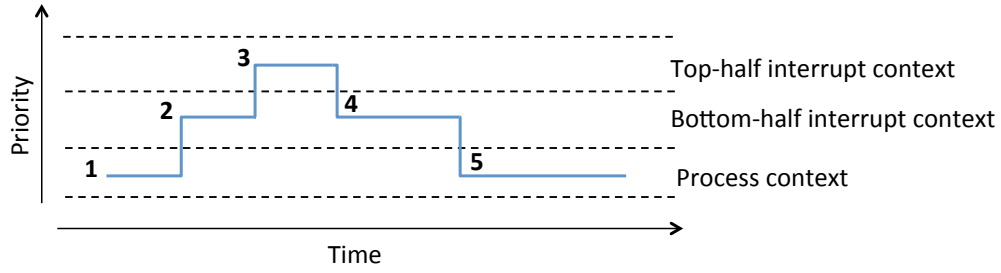
1. Concurrent execution of multiple priority levels, so that a thread may race even with itself.
2. "Ad-hoc" mutual exclusion techniques that avoid lock overheads, such as disabling interrupts and preemption.
3. Deferred execution using `softirqs` and kernel timers.
4. "State-based" synchronization invariants based on 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 show how *DRCheck* handles them, thereby minimizing false positives and avoiding false negatives.

One existing tool that can handle these issues is DataCollider [28]. In DataCollider, 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. In contrast, *DRCheck* covers interrupts and all other contexts; moreover, it can detect not just realized race conditions, but also some potential race conditions that could occur in other event interleavings.

4.2.1 Detecting Concurrency in Driver Execution

There are two basic sources of concurrency in kernel-mode driver execution: (i) multithreading and (ii) "kernel execution contexts" (e.g., Linux kernel threads execute in *interrupt*, or *process* context). High performance drivers (e.g., for network and graphics devices) exploit multithreading to significantly improve I/O performance, especially on CMP systems. Separately, another degree of concurrency (e.g., *reentrancy*) is introduced by the prompt handling of asynchronously occurring, high priority events (e.g, device interrupts) even on uniprocessor systems. Concurrent accesses to shared driver data by different kernel threads (i.e., multithreading) are straightforwardly detected, using standard thread identifier information. In contrast, concurrent accesses from different kernel execution contexts are harder to detect using standard techniques because the accesses originate from a single kernel thread. Before describing how *DRCheck* handles concurrency issues involving a single thread (a.k.a. *intra-thread* concurrency), we provide additional background on kernel execution contexts.



1. In ***process*** context (e.g. packet transmission)
2. Preempted to complete deferred I/O work (e.g. copy received packets from NIC)
3. Preempted to service NIC interrupt (e.g. packet reception)
4. Resume ***bottom-half interrupt*** context
5. Resume ***process*** context

Figure 4.1: The Linux kernel contexts, wherein a thread executes network driver code.

Kernel Execution Contexts

Commodity preemptive OS kernels (e.g., Linux and Windows) provide multiple execution contexts of varying priority levels to enable flexible scheduling of time-constrained, privileged work. This is essential to system responsiveness because it ensures that urgent tasks, such as interrupt handling, can be performed immediately at the highest priority level, and less critical tasks, such as system call handling, can be deferred to a more convenient time. In particular, if a high priority event occurs on a processor (e.g., device interrupt) and the current thread is running in a lower priority context, then the thread is immediately interrupted from its current task and elevated to a higher priority context to process the new event. For example, in the Linux kernel [50], threads execute either in *process* context or the *interrupt* context—which is further divided into the *top-half* and *bottom-half* contexts. The priorities of these Linux kernel contexts in descending order are: (i) *Top-half* (ii) *Bottom-half* and (iii) *process*. In the Windows kernel, execution contexts are called *Interrupt Request Levels* (IRQL), and the kernel offers are 32 IRQLs [62] for scheduling privileged work.

Since drivers handle I/O requests with different timing constraints (e.g., requests from the device are generally more urgent than those from user-space), kernel execution contexts are used to efficiently schedule driver computations. For example, consider Figure 4.1, which illustrates a time line of a Linux kernel thread executing network driver code in the different execution contexts. Initially, the thread is servicing packet transmission requests from user-space in the *process* context. It is then preempted and elevated to the *bottom-half* context to free up memory on the network card by copying previously received packets into system memory. While copying the packets, the thread is again interrupted by the network card, indicating the arrival of new packets from the network that require the driver's attention. The thread then switches to the *top-half* context, where device interrupts are serviced. Since interrupts are disabled on a processor executing in the *top-half* context, interrupt handling must complete quickly. Therefore, the driver simply acknowledges receipt of the interrupt and defers the copying of the received packets to

the *bottom-half* context, where interrupts are enabled. Eventually, after completing the higher priority tasks, the thread switches back to the lower contexts (i.e., *bottom-half*, then *process*) to resume the interrupted, lower priority work.

Intra-Thread Concurrency

Although multiple contexts in kernel-mode execution enable drivers to promptly respond to time critical I/O events (or requests), this benefit unfortunately comes at the risk of introducing subtle variants of common concurrency errors (e.g., a thread racing or deadlocking itself). As an example, consider *intra-thread* concurrency and how it can increase the possibility of reentrant code execution. As shown in Figure 4.1, any data shared by the different driver tasks (e.g., packet transmission and packet reception) must be carefully protected, since these logically independent computations occur asynchronously to one another. The thread will otherwise race itself with on access in the different contexts, and potentially results in data corruption. Worse still, the fact that only a single thread is involved renders standard techniques ineffective at detecting these *self data races*[1]. Also, note that naively introducing mutual exclusion primitives (e.g., locks) to fix this problem could make the thread deadlock itself.

Based on the observation that the execution of a kernel thread could be multiplexed by different kernel contexts, *DRCheck* addresses the *intra-thread* concurrency issues of drivers by tracking the execution context of kernel threads, in addition to the identifier. 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).

[1]Although signal handling introduces similar *intra-thread* concurrency in user-space, to our knowledge, it is basically ignored by prior work.

4.2.2 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` and `CLI` 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 is guarding a critical section. *DRCheck* uses pattern matching over a small window of the trace, starting with the `test_and_set` instruction (`btsl` in x86) 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 the thread succeeded in entering the critical section.

4.2.3 Handling deferred execution

Kernel threads that execute under tight deadlines (e.g., interrupt service routines) are often faced with important tasks (e.g., copying received packets from the network card), which 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 a specified time in the future. Common uses of timers include confirming that tasks are completed on schedule and checking that a device is still functional.

`Softirqs` are commonly used by the interrupt handling routines of high performance drivers to defer work for future processing in a lower priority context (e.g., the *bottom-half* context). However, the way the interrupt-handling thread that defers the work synchronizes with the polling thread, which will do the work, poses a challenge for data race analysis because these

threads may not share any locks. Instead, the interrupt-handling 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-handling 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 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. *DRCheck* addresses these issues relating to kernel timers as follows. First, we associate 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. 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.

4.2.4 Tracking state-based synchronizations

Many peripheral devices—e.g., ethernet, scsi, usb, etc.—behave like finite state machines, 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. Therefore to prevent device failures, the kernel invokes only driver callbacks that are valid for the current state of the device. 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 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 finite state machine snippet in Figure 4.2 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) where driver callbacks are executed, and hence they can incorrectly report races between callbacks with conflicting invariants. Indeed, our experimental study in Section 4.5 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.)

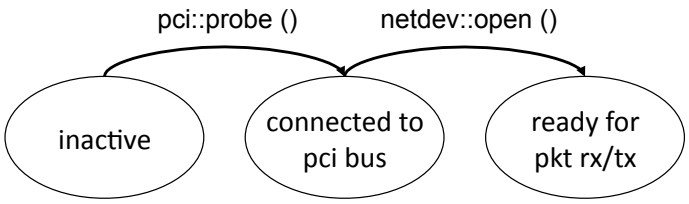


Figure 4.2: State transitions for a Linux PCI network device.

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 [27, 34], perhaps incurring less runtime overhead. We chose 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.

4.2.5 Implementation of *DRCheck*

DRCheck is an extension of the *Lockset* [76] algorithm for detecting data races in applications. *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. The metadata indicates whether the location has been accessed by multiple threads, and if it has, 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* for kernel-mode data race detection as follows. The first set of changes involved adding support for kernel-mode locking primitives, which was straightforward for those (e.g., kernel spinlocks) with behavior similar to user-mode locking primitives. However, kernel-mode locking primitives that also disable interrupts (e.g., `spin_lock_irq()`) were more challenging to support. But, based on previous *Lockset* proposals for supporting interrupts [76], we associate per-CPU virtual locks with interrupt contexts, and these locks are acquired by threads that use interrupt (and preemption) disabling primitives. 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 4.2.3. Finally, we include state-based synchronization tracking as follows. For each shared data the set of device states is also tracked, in addition to tracking the set of locks held by threads on each access. 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

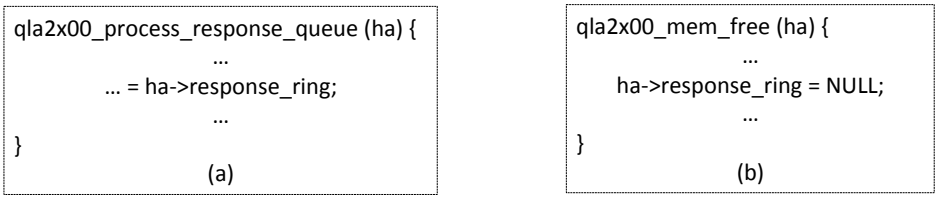


Figure 4.3: Race condition on the response ring pointer in the qla2xxx driver between qla2x00_process_response_queue() (part of interrupt handling) and qla2x00_mem_free() (part of driver unloading).

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 3.3), *DRCheck* tracks synchronization in both the driver and kernel-driver interface execution, while reporting races only in the driver execution.

4.2.6 Case Study: A Data Race Condition in the qla2xxx Driver

To illustrate the key advantages of *DRCheck* over prior work, we will use as a case study a serious race condition found by our techniques in the Linux qla2xxx SCSI host device driver. For the sake of comparison, we consider two alternative dynamic race detectors discussed earlier, DataCollider [28] and *KLockset*.

The race condition, shown in Figure 4.3, is interesting for two reasons. First, if triggered, it could cause the interrupt handler to unsafely compute with a null pointer, potentially leading to hard-to-diagnose system failures. Second, it is difficult to trigger, and it remained undetected for a long time (at least 2 years from the driver version used in our studies) until discovered during unrelated code refactoring by developers. However, the race condition was easily detected using our techniques on a simple execution of loading and unloading the driver.

Our criteria for selecting DataCollider and *KLockset* for evaluation are as follows. We selected DataCollider because it is the state-of-the-art in dynamic race checking for kernel code, and its instrumentation and synchronization-protocol-oblivious approach is complementary to

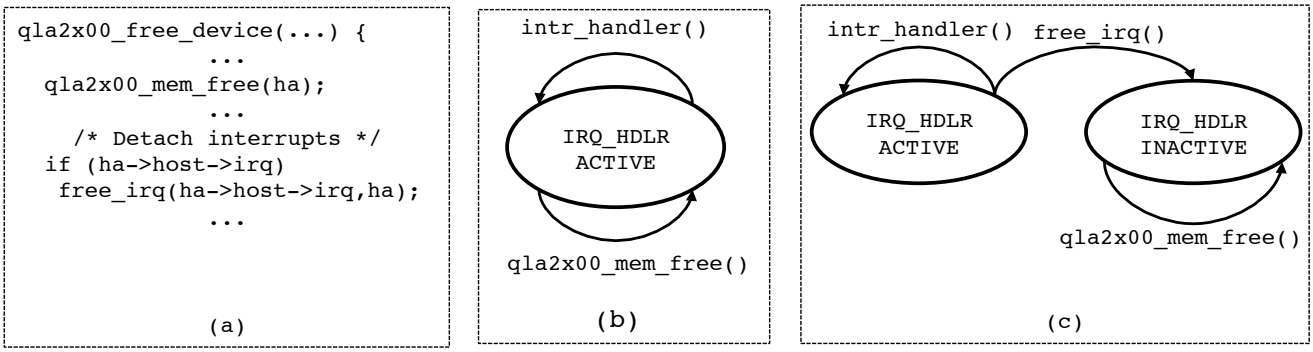


Figure 4.4: (a) During driver shutdown, qla2x00_free_device() calls qla2x00_mem_free() to clear the pointer fields (including response ring) that used for interrupt handling; (b) this occurs in a device state where the interrupt handler could run concurrently; (c) the fix was to disable interrupt handler (via free_irq()) before qla2x00_mem_free().

our decoupled and synchronization-analysis based approach. We selected *KLockset* because it represents the extension of the state-of-the art user-mode Lockset algorithm [76] for kernel-mode execution.

The main conclusion of this study is that driver race detection is problematic for DataCollider because races involving interrupt contexts may be missed, and problematic for *KLockset* because non-lock based synchronization mechanisms lead to false alarms.

We begin our analysis of the data race using the code snippet in Figure 4.4(a). As part of driver unloading, `qla2x00_free_device()` calls `qla2x00_mem_free()` to release memory and clear the associated pointers. The affected memory regions and pointers include those used for interrupt handling, such as `ha->response_ring` (Figure 4.3). Since `qla2x00_mem_free()` is executed before `free_irq()`, it is possible on a multiprocessor system for the interrupt handler to be running on another processor concurrently with driver unloading (Figure 4.4(b)). In such a situation, the interrupt handler could end up using a null pointer (and released memory) in an unsafe manner leading to hard-to-diagnose system failures. However, the interleaving that leads to this failure is hard to produce; in fact, to our knowledge, no Linux bug report of it was

ever submitted, either because the failure never occurred, or because the resulting failures were never properly diagnosed.

Techniques that can detect such subtle driver faults, using workloads as simple as loading and unloading the driver, are useful for improving driver reliability. DataCollider is unlikely to detect this race because it does not work on interrupt handling threads. Remember that to detect a race, DataCollider must stall a thread in its window of race vulnerability as another thread executes to the racing access. Although *KLockset* detected the race, it incorrectly reported that the race remained even after the driver was patched with the developer's fix—reversing the order of `qla2xx00_mem_free()` and `free_irq()` (Figure 4.4(c)). Our investigation showed that the false positive was because the fix did not involve locking. In contrast, *DRCheck* did not generate false positives on the patched driver because it recognized the state-based synchronization used in the fix.

4.3 Detecting Direct Memory Access (DMA) Faults in Drivers

Direct Memory Access (DMA) is an efficient method for performing bulk I/O data transfers 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 throughput. However, incorrect DMA operations are a serious threat to system stability, and so motivated our second dynamic analysis tool named *DMACheck*. *DMACheck* performs instruction-grained analysis of driver execution to detect incorrect DMA operations. 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 will generally apply to other platforms.

4.3.1 DMA in Linux Drivers

The physical memory regions used for DMA are known as *DMA buffers*. To take advantage of DMA for I/O data transfer, a driver must do the following.

1. Map (and pin) the DMA buffer(s) to be used as either source or destination into the kernel and I/O address spaces.
2. Inform the device of the DMA buffer(s) (i.e., their location in the I/O address space).
3. Signal the device to begin the transfer.
4. Wait for the transfer to complete.

As with other system resources, the OS kernel controls the management of DMA buffers. It 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 can be accessed via two different addresses: (i) *virtual addresses* used by drivers, and (ii) *device address* (a.k.a. *I/O bus address*) used by devices. The DMA subsystem of the Linux kernel provides a variety of functions (i.e., `dma_map_single()`, `dma_map_page()`, `dma_map_sg()`) for mapping DMA buffers into the I/O address space, and obtaining the corresponding bus addresses.

Before instructing the device to begin data transfer, the driver has to 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 completed 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. The device signals transfer completion by interrupting the driver. On completion of an incoming transfer, the driver arranges for data to be transferred (up the I/O protocol stack) to the requesting process. For outgoing transfers, the driver optionally releases the source DMA buffers, or it recycles them for future use. Nevertheless, a driver must ensure that its DMA buffers are unmapped (i.e., using `dma_unmap_single()`,

`dma_unmap_page()`, `dma_unmap_sg()`) before it is unloaded.

Because the driver (via processor) and the device access physical memory (DMA buffers) through different data buses, the driver is responsible for avoiding coherence problems; i.e., it ensures that they both work with updated data. To assist drivers in achieving this, the kernel provides functions (e.g. `dma_sync_single() dma_sync_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.

4.3.2 Incorrect Usage of DMA Buffers by Drivers

Based on the preceding description of DMA operations by drivers, one can observe a number of ways that 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 avoid 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 waste 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.

Based on the three issues listed above, we designed *DMACheck* to detect the following five types of DMA buffer faults in drivers: (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 *DMACheck* can be extended for other faults related to DMA buffers or to DMA in general.

4.3.3 *DMACheck* 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 the *DMACheck* detects can be grouped into two categories, based on the granularity of the detection analysis. First, DMA buffer races are detected by using *instruction-grained analysis*, which checks if the memory operations by the driver overlap a DMA buffer the device is currently accessing. The second category of DMA buffer faults (e.g., misaligned DMA buffers) can be detected by inspecting the arguments of the DMA function calls (e.g., `dma_map_single()`) that are made by the driver.

DMACheck detects races on DMA buffers by checking for unserialized accesses by the driver and device to a DMA buffer. However, doing this with precision 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, the second option (ii) might appear to be a more accurate approximation of when the device actually does use a DMA buffer for transfer, however, the conservative first option (i) turns out to be a more practical approach for two 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 (ii) introduces the complexity of understanding device-specific logic of how DMA transfers are configured by drivers–a non-scalable 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.

4.3.4 Other Approaches for Mitigating DMA Faults

We conclude our discussion on DMA faults in drivers by briefly describing alternative hardware [1, 4] and software [86] proposals for addressing this problem.

An I/O Memory Management Unit (IOMMU) [1, 4] is an hardware device for translating DMA bus addresses into the physical memory addresses. In the absence of an IOMMU, unsafe DMA operations by a driver could circumvent the conventional memory protection that is enforced via the processor's memory management unit. This is because the device's accesses to physical memory bypasses the memory management unit. The IOMMU hardware provides an efficient mechanism for addressing this issue by restricting the physical memory regions that are accessible to a device. By preventing the device from accessing physical memory regions that

are not DMA buffers, IOMMUs complement *DMACheck*, which detects conflicting driver and device accesses to DMA buffers. IOMMUs are increasingly available in commodity computing systems.

Nexus-RVM [86] intercepts device register reads/writes by drivers and employs a device-specific reference validation mechanism to ensure that the device is correctly configured for DMA transfers. For example, Nexus-RVM can check that DMA transfer requests are properly formatted (e.g., DMA buffers are specified before a transfer command is issued). Nexus-RVM's focus on correct DMA setup is complementary to *DMACheck*'s focus on DMA buffer races. Moreover, Nexus-RVM's use of device-specific logic makes Nexus-RVM more accurate, but less general than *DMACheck*.

4.4 Detecting Memory Faults in Drivers

Commodity OS drivers are prone to memory safety issues (e.g., buffer overflows) because they are typically written in unsafe languages such as C and C++. Studies have confirmed that memory safety issues are a significant source of bugs [73] and security vulnerabilities [15] in production drivers. This has inspired a number of dynamic binary analysis tools for detecting common memory faults in drivers, including *DDT* [17], *KAddrcheck* [30], and *KMemcheck* [61]. In particular, these tools check driver execution for unsafe memory accesses (e.g., to unallocated memory) and unsafe uses of uninitialized data. Similarly, our third driver checking tool, named *DMCheck*, detects the same kinds of memory bugs in drivers.

The availability of more tools for detecting memory bugs in drivers compared to data races and DMA buffer faults is probably due to the similarities in the memory safety issues for user-level and kernel-level codes. These similarities has significantly influenced the design of these tools (including *DMCheck*) in the sense that they are directly inspired by *Memcheck*; which is a state-of-the-art tool for finding memory faults in application binaries. *Memcheck* tracks the allocation and initialization status of each byte of memory in an application's address space, thus

enabling fine-grained detection of unallocated memory access, memory leaks, and unsafe uses of uninitialized data. Allocation status information is maintained by observing the application's memory management operations (e.g., `malloc()` & `free()`. This enables easy identification of accesses to unallocated memory, and memory bytes that remain allocated after the application exits. On the other hand, uninitialized data errors are more challenging to detect because some uses are considered safe (e.g., copying to pad data structures), while others are unsafe (e.g., accessing memory through an uninitialized pointer). To distinguish between safe and unsafe use of uninitialized data, *Memcheck* employs *dynamic information flow tracking* to propagate initialization status, as data flows through registers and memory locations.

4.4.1 *DMCheck*: Detecting Memory Faults in Kernel-Mode Drivers

To adapt *Memcheck* for kernel-mode drivers, *DMCheck* addresses two issues concerning memory fault detection in kernel-space: (i) recognizing kernel-level memory management, and (ii) dealing with memory objects that are used by the driver, but (de)allocated outside the driver. The first issue is handled by recognizing that kernel memory management functions such as `kmalloc()` and `kfree()`, behave similarly to user-space functions such as `malloc()` and `free()`.

The second issue arises because drivers need 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 used 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 3.3. Consequently, the address range for each such memory object can be captured by the analysis.[2]

[2]As in prior work, we trust the kernel-driver interface module. For example, we assume that pointer and size

4.5 Evaluation of Fault Detection of Driver Checking Tools

We evaluated how Guardrail's instruction-grained monitoring improves driver bug detection by implementing our proposed dynamic binary analysis tools in the Guardrail framework. Specifically, we studied the bug detection effectiveness of the following three tools: (i) *DRCheck*, for detecting data races (Section 4.2), (ii) *DMACheck*, for detecting DMA faults (Section 4.3), and (iii) *DMCheck*, for detecting memory faults (Section 4.4). We applied these tools to detect bugs in eight production Linux network and storage drivers. The results show that Guardrail enables better detection of driver bugs than previous approaches.

First, we describe the experimental setup for this evaluation, including the Linux drivers that were used in the study. Next, we examine the driver bugs that were detected by our checking tools in details. We compare each tool against existing kernel-mode dynamic correctness checkers by evaluating whether the bugs detected using Guardrail could be similarly detected using other techniques.

4.5.1 Experimental Setup

Since Guardrail employs novel hardware-assisted instruction-level tracing of driver execution to decouple and run the checking tool in a separate virtual machine from the monitored driver, we conducted our experiments in a simulation environment so that we could model the proposed hardware tracing support. However, we used real-world device drivers and software stack in our experiments. Unfortunately, the simulation environment restricted our studies to drivers whose device models were readily available in the simulation framework, as further explained below.

Simulated Hardware We used the Simics [87] full system simulator (academic package version 4.0.63) to prototype Guardrail by modifying an x86 chip multiprocessor model with exten-

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 scope of this thesis.

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
DriverVM	2 VCPU, 1GB RAM
Analysis VM	1 VCPU, 512MB RAM

Table 4.1: Configuration of the simulated Guardrail system that was used for evaluation.

sions for streaming the instruction-level trace of multithreaded driver execution to the decoupled checking tool. The design and implementation of Guardrail's hardware-assisted instruction tracing are described in more details in Chapter 6. As illustrated in Table 4.1, we simulated Guardrail as a dual-core, 2.6 GHz, Intel Pentium 4 CMP system with 2GB memory. We configured Guardrail to run two virtual machines: (i) DriverVM for the monitored driver and (ii) AnalysisVM for the checking tool. The DriverVM is configured with two virtual CPUs and 1GB of physical memory. 512KB of the DriverVM's physical memory is reserved for streaming the execution trace of the driver's execution to the checking tool (assuming that each instruction record can be compressed down to one byte [16]). On the other hand, the AnalysisVM is configured with one virtual CPU and 512MB of physical memory.

Real-World Device Drivers Our Simics simulator package included models for network and storage devices, but not models for other important device classes such as graphics and audio devices. Thus, we used only Linux network and storage drivers in our evaluation. In particular, our experiments focused on the five network drivers and three storage drivers that are presented in

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

Table 4.2: Linux drivers for evaluating bug detection effectiveness of Guardrail.

Table 4.2. The drivers ran in a 32-bit Fedora Core 6 OS flavor of the Linux 2.6.18 kernel. Driver workloads were generated using standard I/O intensive benchmarks. Network driver workload was generated using the *Apache* webserver, the *Memcached* in-memory key-value store, and the *Netperf* network measurement utility. The *Postmark* filesystem benchmark was used for the storage drivers.

Tool	Data race count					Confirmed/Fixed		Total
	qla1280	qla2xxx	sym53c8xx	tg3	tulip	Yes	No*	
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 4.3: DRCheck found nine serious races in Linux drivers, six of which have been confirmed/fixed. DataCollider will detect two races, if all the racy accesses were in its sample set, and six races if the racy accesses, also occurred in an idealized order. The number of unconfirmed races are starred in the table.

4.5.2 Data Races

As shown in Table 4.3, *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 [28]. 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 4.4 (*DeferExec* is *DRCheck* without state-based synchronizations (Section 4.2.4)). Although *DDT* [46] detects data races, it was not described in sufficient details to allow comparisons in this context.

	qla1280	qla2xxx	sym53c8xx	tg3	tulip	Total
KLockset	1	36	13	35	26	111
DeferExec	1	13	13	22	18	67
DRCheck	0	0	6	4	1	11

Table 4.4: False positives of our race detection techniques.

4.5.3 DMA Faults

The different DMA buffer faults found by *DMACheck* in six drivers are summarized in Table 4.5. 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 address) that

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

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 five drivers (i.e., *e100*, *e1000*, *pcnet32*, *tg3*, *tulip*). As discussed earlier, this bug is a serious issue in non-coherent (i.e., non-x86) systems. And *sym5c8xx* was the only driver 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.

4.5.4 Memory Faults

As shown in Table 4.6, *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 [14, 33, 80, 85, 86, 90]. 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 4.6 to compare *DMCheck* against existing kernel-mode memory fault detectors for the Windows (*DDT* [46]) and Linux (*KAdddrcheck* [30], *KMemcheck* [61]). *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 should therefore detect both the memory faults.

Driver	Memory faults in drivers			
	DMCheck	DDT	KAddrcheck	KMemcheck
e1000	1	0	0	1
qla2xxx	1	1	1	1
Total	2	1	1	2

Table 4.6: DMCheck found two memory bugs in Linux drivers that are now fixed, including the discovery of the *qla2xxx* bug. While KMemcheck can find both bugs, KAddrcheck and DDT can find only one.

4.5.5 Fault detection summary

In summary, our evaluation validated our proposal that instruction-grained dynamic analysis can be used to improve the reliability of device drivers by detecting bugs in their execution. Our dynamic analysis tools detect a significant number of hard-to-find bugs in production Linux drivers (e.g., memory faults, data races, and DMA faults) which are missed by other tools, including a previously unknown buffer overflow in the *qla2xxx* storage driver. 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 these tools demonstrate its value as a general-purpose framework for implementing sophisticated correctness checking tools for drivers, in contrast to error-specific tools such as *DataCollider*.

4.6 Considerations for Deploying Guardrail

As we have demonstrated through evaluation with production Linux drivers, Guardrail not only detects driver bugs that are missed by other dynamic techniques, but also bugs for which no other tools currently exist. This indicates that Guardrail can improve driver debugging and testing, or make production systems more resilient to the harmful effects of defective drivers. However, in evaluating how to deploy Guardrail in these scenarios, it is worth considering the potential for false negatives and false positives in our checking tools.

The underlying *Lockset* algorithm of *DRCheck* leads to false data race reports for properly synchronized code that deviates from the expected locking discipline. This is a serious limitation for production deployments because halting a system for a false alarm is unacceptable. Moreover, the fact that 76%–90% of true races are actually *benign* [56] means that simply avoiding false alarms (e.g., by incorporating a *Happens-Before* approach [89]) 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 3.2); which automatically classifies the alarms raised by *DRCheck* into harmless or harmful races. Furthermore, *DRCheck* could be extended to recognize synchronization patterns and *benign* data sharing patterns that it had incorrectly flagged in the past, to reduce false alarms and the need for triaging.

The dynamic nature of our techniques creates the possibility of false negatives. In other words, 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 [18, 70].

4.7 Limitations of Guardrail

Although Guardrail improves driver bug detection compared to previous approaches, its effectiveness is limited, however, to particular types of driver bugs. This limitation fundamentally arises from the fact that Guardrail only observes the driver portion of the I/O protocol stack, whereas the other portions such as the states of the application, OS, and the device are hidden. As a result, Guardrail may miss driver bugs whose identification requires examining the execution states of the other parts of the protocol stack. An example of a such bug that could elude Guardrail is incorrect parsing of device state by the driver during interrupt handling that leads to erroneous servicing of device interrupts.

Because Guardrail observes when the driver accesses external state (e.g., reading device registers), it mitigates the issue of limited visibility by exploiting such accesses to approximate otherwise hidden state (e.g., device configuration). However, this approach suffers from relying on the driver to correctly probe relevant external state. Thus, reliable access to the execution state of other parts of the protocol stack [75] during driver execution will greatly improve Guardrail.

4.8 Summary

Inspired by the success of instruction-grain dynamic analyses in improving the reliability of user-space software, this chapter shows that similarly sophisticated dynamic analyses yield similar reliability improvements for kernel-mode drivers. We presented dynamic analysis tools for detecting memory faults (*DMCheck*), data races (*DRCheck*), and DMA faults (*DMACheck*) in Linux kernel drivers. These tools cover a significant portion of the major classes of correctness issues that plague modern drivers. Our evaluations showed that our proposed tools effectively detected driver bugs that are typically missed by existing techniques; suggesting that instruction-grain dynamic analysis could enable the detection of more faults during driver development and the mitigation of more driver faults in production settings. As discussed in Chapter 6, these

driver reliability improvements can be efficiently realized by using Guardrail to decouple dynamic analysis from the monitored driver execution.

Chapter 5

Interposing on I/O Operations of Drivers

A key feature of Guardrail is that it enables powerful correctness checking, such as those described in the previous chapter, to be applied on driver execution in a decoupled fashion to avoid the high performance overheads that are typically incurred by heavyweight analyses. Although, decoupling improves runtime checking performance, it introduces the risk of the system being corrupted by faults in the driver's execution that are yet to be detected by the lagging checker. Guardrail contains such faults by interposing on the driver's interaction with the device (i.e. I/O operations) to prevent the side-effects of the unchecked and potentially faulty driver execution from affecting the device and I/O operations.

Guardrail employs commodity virtualization techniques to realize interposition that is *transparent* (works for arbitrary combinations of driver binaries and I/O devices), *trustworthy* (requires only modest increase of the trusted computing base), and *flexible* (compatible with decoupled correctness checking). Despite its use of virtualization, Guardrail is designed to protect physical devices, rather than virtual devices that are often used in virtualized environments.

In this chapter, we present the design and implementation of Guardrail's I/O interposition component. We describe how this component is combined with a decoupled runtime checker for online protection of persistent I/O state from driver bugs. Finally, we evaluate the impact of I/O interpositioning on the performance of common audio, video, storage and network benchmarks.

5.1 Background

Before describing Guardrail's I/O interposition technique, we briefly review the general use of interposition for mitigating driver bugs, and the specific use of interposition for safeguarding I/O operations.

5.1.1 Interpositioning on Driver Interfaces

Interposing on the interactions between a driver and the rest of the system is a popular technique for mitigating driver faults. This is because faults propagate outside the driver's boundary through interactions that have external side effects (e.g., writing kernel state, configuring the device, etc.). In particular, an interposition layer gives a fault mitigation system the ability to mediate each attempt by the driver to interact with the outside environment. Thus, if the driver is found to be executing in a faulty manner at this point, then the interaction can be blocked to prevent the corruption of trusted system components (e.g., OS kernel and I/O devices).

5.1.2 I/O Interposition in Nexus-RVM

As discussed earlier in Section 1.3, Nexus-RVM is the only prior technique that protects both devices and the OS kernel from driver faults [86]; other proposals focused solely on protecting the OS kernel. Thus, interposing on the device interface of Nexus-RVM drivers is the only comparable proposal to Guardrail.

Nexus-RVM protects the OS by moving drivers out of the kernel space into user space, and it protects the device by using a *reference validation mechanism* (RVM) [5] to mediate the device interactions of the driver. The RVM consists of device-specific *reference monitors*, which run in the kernel (as part of the trusted computing base (TCB)), to ensure that a driver interacts with the device in a manner that conforms to a *device safety specification* (DSS). In particular the RVM prevents unsafe driver actions such as exploiting DMA for illegal memory accesses, priority

escalation, processor starvation, and device-specific attacks. To ensure that the RVM mediates all device interactions, drivers access device registers through system calls which invoke the RVM with the relevant parameters for validation. Nexus-RVM's I/O interposition approach differs from Guardrail in the following two ways.

First, interposition in Nexus-RVM is not transparent to drivers; device register accesses must be identified in driver code and modified to use system calls. In contrast, the use of virtualization by Guardrail makes the interposition of device register accesses completely transparent to drivers. Therefore, while Guardrail can be readily applied to binary drivers, Nexus-RVM requires availability of driver sources, and it imposes additional burden on the developer to modify the driver sources.

Second, interposition and correctness checking (via reference monitors) are coupled in Nexus-RVM, whereas both concerns are decoupled by Guardrail. This design choice impacts *trustworthiness* of fault mitigation because while interpositioning is generally straightforward conceptually, correctness checking on the other hand can be arbitrarily complex depending on the correctness properties of interest (e.g., concurrency management, conformance to the device protocol, etc.). In other words, while Guardrail exploits decoupling to move correctness checking out of the TCB into user space, the non-trivial RVM mechanism of Nexus-RVM remains part of the TCB, raising *trustworthiness* concerns.

5.2 I/O Interposition in Guardrail

Since devices are controlled by reading/writing device registers, Guardrail's interposition layer prevents driver errors from propagating into the device by: (i) intercepting all[1] device register accesses, (ii) coordinating with a decoupled correctness checker to determine the safety of the

[1]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.

accesses (and the driver's execution up till that point), and (iii) ensuring their timely eventual completion as soon as they are deemed safe. Figure 5.1 depicts how the interposition layer leverages virtualization to transparently mediate a device register access that originates from a driver. The actions of the interposition layer are discussed in more details below.

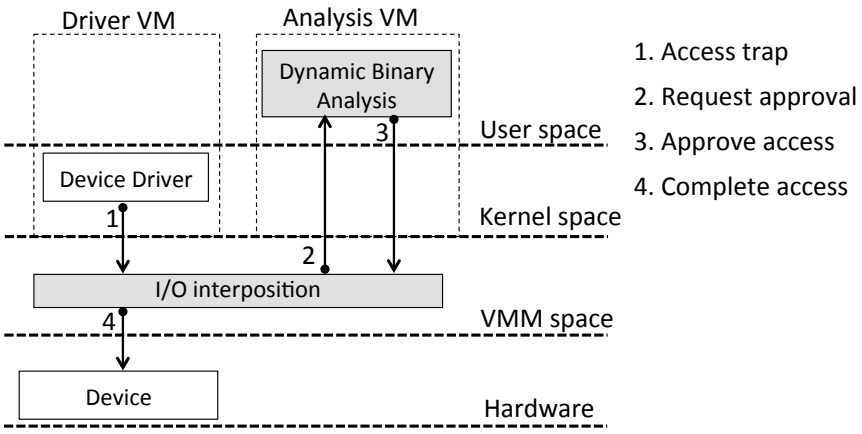


Figure 5.1: Transparent mediation of device register access.

5.2.1 Intercepting device register access

As described earlier, device registers are mapped into the I/O address spaces (I/O ports, MMIO, and PCI-config), and so the interposition layer must identify and intercept memory accesses from the driver's VM (DriverVM) that are destined for those address spaces. One approach is to identify the driver instructions (e.g., using dynamic binary analysis) that access device registers [2]. Although, I/O port accesses are easy to identify with this approach, since they are performed using special instructions, accesses to MMIO and PCI-config locations are difficult to distinguish from regular memory accesses because they are performed using regular load/store instructions.

Instead, Guardrail takes the alternative approach of making the I/O address spaces privileged and thus inaccessible to the DriverVM. Thus, any attempt by the driver to access a device register will fault into the virtual machine monitor (VMM), where it is handled by the interposition layer.

This is a more appealing approach because I/O address spaces are easier to identify compared to the load/store instructions that access them. Moreover, in virtualized x86 environments the I/O port address space is normally regarded as privileged and thus accesses to this space from a guest VM will fault. Because load/store instructions that target the other I/O address spaces are subject to the normal memory management unit (MMU) translation mechanisms, we can intercept them by configuring the DriverVM page tables so that such accesses fault to the VMM. The faulting address can be used to distinguish the page faults resulting caused by interposition from normal MMU page faults. Note that interposition only affects communication originating from the DriverVM; interrupts *to* the DriverVM may be delivered normally.

5.2.2 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 [86], 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 requests approval from the checker to complete the access. The request includes details of the faulting instruction (e.g., thread id, faulting address). The device access will be approved if the checker verifies that the driver's execution is fault-free up to that point. Otherwise, if the access is disapproved because of a driver bug, the interposition layer can initiate recovery using appropriate techniques [14, 48, 81].

Because the checker's response will typically incur some latency, there are a number of options available to the interposition layer regarding how it waits for the response. In addressing this issue we carefully considered the following two options because of their transparency to the thread scheduling policies of the guest OS. The interposition layer could either stall the faulting virtual CPU(vCPU), or it could stall just the faulting thread in the guest VM.

The vCPU can be stalled either by descheduling it, or by holding the request in the VMM. To

maintain the responsiveness of the guest OS, if interrupts are generated during this period, they should be delivered to the vCPU at the point just before the faulting instruction.[2] Furthermore, the interposition layer must promptly reschedule the vCPU once the device access is approved if it was descheduled before expiration of its time quantum.

For development expediency, we selected the alternative option of stalling the guest OS thread: the interposition layer simply returns control to the faulting instruction. In other words, a guest OS thread that accesses a device register will repeatedly trap into the interposition layer until either the checker verifies the safety of the access, or the thread is naturally preempted by the guest OS. Although, repeatedly trapping into the VMM wastes physical CPU cycles, we however expect that the correctness checker will optimized to respond quickly to minimize the waste. Moreover, this approach avoids the complexity of managing vCPU scheduling.

5.2.3 Completing device register access

After the checking tool has verified that the intercepted device register access is safe to perform, there are two ways of issuing the operation to the device: (i) retry the faulting instruction after temporarily making the device register available to the guest OS [25], and (ii) emulate the faulting instruction in the VMM.

Although the first option offers better performance by avoiding emulation overheads it is extremely tricky to implement correctly without breaking bug containment guarantees. This is because commodity operating systems run in a single address space that is shared by concurrently executing kernel threads. Therefore, the interposition layer must ensure that the device register is only accessed by the verified operation in the intended thread, at an appropriate time, and during this window of availability. However, since memory is managed at page granularity it means other device registers in the same page are simultaneously accessible to the guest OS. Therefore,

[2]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.

the interposition layer must also prevent access to the other device registers in the same page.

Consequently, in our current prototype we have chosen the emulation option to avoid potential containment errors, especially in symmetric multiprocessing (SMP) environments. Although this option introduces the additional overheads of emulation, it is however relatively easier to implement correctly. This approach mirrors the *trap-and-emulate* handling of privileged operations in traditional virtualization, and simplifies the incorporation of I/O interpositioning into commodity VMMs.

5.3 Implementation of I/O Interposition

Now that we have presented a high-level description of how Guardrail interposes on the I/O operations of drivers, we now describe how those ideas were implemented in our current Guardrail prototype. We used commodity virtualization—specifically Xen [10]—to implement Guardrail's I/O interposition functionality. The simplicity of our design enabled a lightweight implementation, which required extending the Xen VMM by about 500 lines of C code. Our design does not rely on any Xen-specific features, and could therefore be implemented using other commodity VMMs (e.g., KVM [44]). In the following discussion, we first present some background on Xen before describing the key implementation issues that we addressed in our current prototype.

5.3.1 Xen Background

Xen is a bare metal virtual machine monitor for x86 and ARM processor based systems running commodity guest operating systems like Linux and Microsoft Windows. Guest VMs are known as *domains* in Xen terminology. Xen designates one of the domains as *privileged*, and delegates to it the responsibility of managing the *unprivileged* domains and physical devices in the system. This design choice helps to reduce the complexity of the VMM[3].

[3]In Xen terminology, the privileged domain is called *dom0*, while an unprivileged domain is called *domU*.

Xen offers two virtualization modes: (i) *paravirtualization* (a.k.a Xen-PV) and (ii) *hardware-assisted full virtualization* (a.k.a Xen-HVM). Xen-PV requires guest OS modifications—replacing privileged operations (e.g., page table updates) with *hypercalls* to the VMM—to simplify x86 virtualization and reduce virtualization overheads. In contrast, Xen-HVM leverages virtualization support provided by recent commodity hardware [3, 83] to efficiently run unmodified guest operating systems. Unfortunately, at the time of completing this thesis, hardware virtualization support for Xen-HVM was not readily available in x86 simulators, including the simulator (Simics [87]) that we used to study hardware-assisted tracing of driver execution (Section 6.3). Thus, we used Xen-PV to prototype Guardrail. Although Guardrail runs a monitored driver in an unprivileged domain, the driver is however granted direct access to the corresponding physical device.

5.3.2 Intercepting device register access

Guardrail intercepts the I/O port accesses originating from drivers in a guest VM by exploiting the fact that Xen makes the I/O port space inaccessible to both privileged and unprivileged guests VMs. By default, I/O port accesses by guest VMs in Xen trap into the VMM layer. On the other hand, with *direct device assignment* MMIO and PCI-config are normally accessible to guest VMs, and so we made them inaccessible as described below.

The normal way that a guest VM gains access to an I/O memory region is by mapping the region into its virtual address space. This process involves creating virtual address translations for the I/O memory region in the guest's page tables. However, since page tables in virtualized systems are (ultimately) managed by the VMM, Guardrail is able to intercept requests from the guest operating system to create virtual address translations. At that point, Guardrail marks the corresponding page table entry (PTE) *not-present* to ensure that a page fault is triggered by the MMU on references through the affected virtual addresses. In addition, Guardrail keeps track of interposed virtual addresses by specially marking a currently unused bit in the PTE to distinguish

page faults caused by interposition from normal page faults. Thus, accesses to device registers by guest VMs trap into the VMM layer for appropriate handling by Guardrail.

5.3.3 Validating device register access

The interposition layer requests validation of the intercepted I/O operation from a decoupled correctness checking tool. This validation step involves verifying that the driver's execution is fault-free up till and including the intercepted operation. To facilitate this, Guardrail streams an instruction-grain execution trace of the monitored driver to the checking tool.

As described in more details in Section 6.3, Guardrail's tracing technique involves extending commodity processors with log *producer*/*consumer* components. The *producer* populates the log with the *committed* instructions of the monitored driver, while the *consumer* extracts the log entries for analysis. However, since the intercepted I/O operation faulted (i.e. did not *commit*), an interface is provided for the interposition layer to inject the faulting instruction into the log. A logical log is maintained for each vCPU in the monitored guest VM.

The interposition layer also tracks log production and consumption using the *head* and *tail* indices respectively. Per standard usage, *head* indicates the next free slot, while *tail* indicates the next log record to be consumed. The interposition layer uses this interface to efficiently coordinate validation requests and responses with the checking tool. Recall that in our design the interposition layer busy-waits for the validation request to complete. Therefore, on injecting an I/O operation into the log, the interposition layer also records the current *head* index. Then, it waits for a response by monitoring the *tail* index to detect when the I/O operation has been analyzed[4]. The checking tool communicates its disapproval of the I/O operation to the interposition layer by setting the *tail* to special value. In summary, the interposition layer and the decoupled checker implicitly coordinate the validation of the I/O operations of a monitored using the

[4]Because our design does not affect preemptibility in the faulting vCPU, the *head* of the vCPU's log could change while waiting for validation if the faulting thread is preempted by another driver thread that attempts to perform I/O.

execution logging mechanism.

5.3.4 Emulating device register access

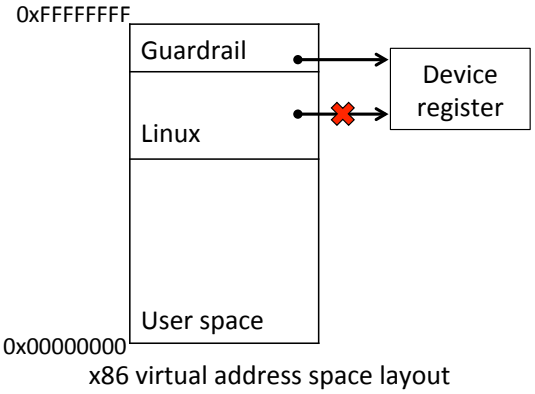


Figure 5.2: Guardrail and Linux pointers that reference the same device register. Guardrail modifies the page tables to ensure that references through the Linux pointer, fault.

As described earlier, after validation by the correctness checker the intercepted I/O operation is completed by emulating the faulting instruction. Normally, the vCPU context at the fault point is used for emulation in the VMM; for example, the vCPU's program counter is used to decode the faulting instruction. However, emulating MMIO and PCI-config access in this manner results in the use of the guest virtual addresses (derived from the vCPU registers) that led to the original page fault, and so the page fault will be repeated during emulation. This problem does not apply to the emulation of I/O port accesses because virtual addresses are not involved.

Guardrail deals with this problem by creating new translations from the VMM's virtual address space to the MMIO or PCI-config address space. These VMM-level virtual addresses are used to access the affected device register(s) during emulation, rather than the guest-level (i.e. kernel) virtual addresses that would have been derived from the vCPU context. Figure 5.2 illustrates a device register that is referenced by VMM-level and guest-level pointers. However, creating (or destroying) VMM-level virtual address mappings to device registers is expensive,

because it involves page table and TLB updates. Therefore, in order to improve performance such mappings should ideally be persistent in the page tables (and TLB) to allow reuse. Unfortunately, achieving this ideal situation is challenging for two reasons: (i) the size of the entire MMIO address space in modern systems, and (ii) the cost of translating the guest-level virtual address (of device registers) to VMM-level virtual address. Guardrail addresses these two challenges as follows.

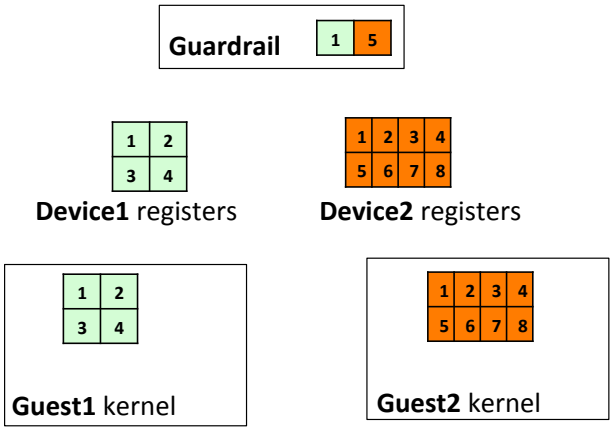


Figure 5.3: The mapping of the registers (at page granularity) of two directly assigned devices (Device1 and Device2) into Guardrail's address space. While each guest kernel persistently maps the entire register space of its assigned device, Guardrail temporarily maps some of the registers to accelerate emulation.

First, the MMIO address space of modern devices is often very large (e.g., the network and video cards in our evaluations had 64MB and 288MB of device registers respectively). Furthermore, modern computing systems are equipped with at least three I/O devices (e.g., display, storage, and network). These two observations—coupled with the relatively small virtual address space of the VMM (e.g., Xen occupies 64MB on x86)—demonstrate the in-feasibility of mapping the entire MMIO address space for every device into the VMM layer. Instead, Guardrail reserves a relatively small region of the VMM's virtual address space for temporarily mapping device registers at page (e.g., 4KB) granularity when needed. The mappings are created by di-

rectly modifying the PTEs in the VMM's page tables to reference the relevant physical pages. In other words, the physical pages of accessed device registers are multiplexed into the available slots in the reserved virtual address space. This region is of configurable size and managed using *least recently used* (LRU) replacement policy. For our evaluations, we reserved 512KB from the 64MB virtual address space of the VMM (i.e. 0.7%) for this purpose. Figure 5.3 illustrates the mapping of the registers of two directly assigned devices into the virtual address spaces of Guardrail and two guest OS kernels. The mappings are shown in the figure at page granularity, i.e. each cell represents a page of device registers. As can be observed in the figure, the entire MMIO address space of each device (Device1, and Device2) is persistently mapped by the respective guest OS kernels (Guest1, and Guest2). In contrast, only temporary mappings of individual device register pages are maintained by Guardrail to emulate device register access.

Second, when a device register access is intercepted, the faulting address that is supplied by the page fault mechanism (i.e. `CR2` register) is the guest-level virtual address. Therefore, in order to emulate the access, Guardrail must translate the guest-level virtual address to the corresponding Guardrail virtual address. One obvious approach, using existing MMU structures, is to search Guardrail's page tables for a virtual address translation that matches the device register. Unfortunately, this incurs undesirably high overheads, even with the relatively small size of Guardrail's virtual address space. Instead, Guardrail uses an associative array to perform the desired translation for each device register, i.e. from guest-level address (*key*) to corresponding Guardrail virtual address (*value*). If the device register is not mapped into Guardrail's virtual address space, a new mapping is created as described earlier, and the array is updated with the appropriate translation. The key-value pairs are maintained at page granularity, and so there are as many pairs as needed for the reserved region in Guardrail's virtual address space (e.g., with 4 KB virtual pages, 128 pairs are needed for 512 KB).

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

Table 5.1: Linux drivers and corresponding I/O devices.

5.4 Evaluation of I/O Interposition

We now present our evaluation of the impact of Guardrail's interposition approach on the performance of I/O intensive workloads. The experiments described here measure only the overheads of intercepting and emulating I/O operations. In other words, these results reflect only part of the overheads of using Guardrail to protect physical I/O devices from driver bugs. In particular, the overheads of using decoupled correctness checking to validate the I/O operations of drivers are not measured, those are reported in Section 6.5. We focused on the following four types of I/O workloads in Linux systems: (i) audio, (ii) video, (iii) network, and (iv) storage. The Linux drivers and corresponding devices that are used in our experiments are presented in Table 5.1.

5.4.1 Methodology

We compare the performance of I/O workloads in a Linux system with Guardrail's I/O interposition against their performance in a non-virtualized, but otherwise similarly configured, Linux system. We also measure the CPU and memory utilization of Guardrail's I/O interposition. Since we employed commodity virtualization (i.e., Xen VMM) to prototype Guardrail, we perform the same experiments on a Linux system running on Xen to separate the overheads of I/O interposition from that of CPU and memory virtualization.

We adopt the following naming convention to report results for the three comparison points.

Linux represents a non-virtualized system running the Linux operating system. *Xen* represents running *Linux* as guest on Xen with directly assigned physical I/O devices (i.e., only CPU and memory resources are virtualized). *IO-Interpose* represents *Xen* with Guardrail's I/O interposition extensions. To mitigate spurious differences in experimental results, the same user-level software stack, Linux kernel, and Xen VMM versions were used in the three configurations, as appropriate. Further details of our experimental setup are presented below.

System software We implemented *IO-Interpose* by extending paravirtualized (PV) Xen-3.3.1 with the changes described in Section 5.3. Recall that *IO-Interpose* reserves a configurable number of VMM-level page table entries to generate virtual address translations for device registers. It also maintains a direct map of the same size between these VMM-level virtual addresses and the corresponding guest-level virtual addresses of device registers. Through experiments, we observed that 128 page table entries was sufficient for achieving negligible miss rates. 128 page table entries map up to 512 KB of device registers into the VMM-level virtual address space (assuming 4KB pages in the guest). The direct map data structure consumes just 1 KB, which is less than 1% of VMM memory. The *IO-Interpose* numbers that we report here were obtained using this configuration.

Fedora Core 6 (2.6.18 Linux kernel) is used as the OS kernel in the non-virtualized environment (*Linux*), and a PV version of the same kernel as the guest OS kernel in the virtualized (*Xen* and *IO-Interpose*) environments. For convenience, the experiments in the virtualized environments were performed in the privileged domain; unprivileged domains were not used in the experiments. Each guest OS driver used for the experiments was a stock unmodified native driver with direct access to the corresponding physical device.

Benchmarks We used a set of popular I/O intensive benchmarks to measure the impact of interposition on I/O performance. We used *Mplayer*, an open source media player, to measure the impact on audio and video performance. We used the *Apache* webserver, the *Memcached*

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 5.2: I/O intensive benchmarks.

Network client						Postmark		
Apache			Memcached		Netperf			
Requests	Concurrency	File size	Threads	Req/thread	Length	Trx.	Files	File size
16K	1–64	40KB	32–256	100K	20 secs.	100K	20K	10KB–20KB

Table 5.3: Network and storage benchmark parameters.

in-memory key-value store, and *Netperf* to evaluate the impact on network performance. And finally, we used the *Postmark* benchmark and the GNU *Make* utility to evaluate the impact on storage performance. These benchmarks are briefly described in Table 5.2. The parameter settings used to evaluate the network and storage benchmarks are presented in Table 5.3.

Hardware A Dell Precision 390 workstation served as the test system for our experiments. This system was equipped with Dual-Core Intel Core 2 Duo processors running at 2.66 GHZ and with 2GB of physical memory. The network performance studies involved client-server experiments. The server ran in the test system, while the client workload was supplied by a non-virtualized system. The client system was 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. All other I/O experiments were conducted on the test system.

5.4.2 Audio & Video performance

We used *Mplayer's* benchmarking features to measure how interposition affects the audio and video I/O performance. We studied the following scenarios in our study: (i) movie playback with both audio and video outputs enabled, (ii) audio output generation with disabled video output, and (iii) video output generation with disabled audio output. 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 the *ALSA* audio output and the *X11* video output modes. The reported results are the median of 10 runs.

	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 5.4: Impact of I/O interposition on movie playback.

The movie playback on *IO-Interpose* was smooth and of similar audio and video quality to *Linux* and *Xen*. The results from this experiment are summarized in Table 5.4. As shown in the table, the playback time and frame rate on *Linux* are only marginally better than on *Xen* and *IO-Interpose*. The movie playback took 150.51 seconds on *Linux* and 150.52 seconds on *Xen* and *IO-Interpose*. The achieved frame rates were 23.94 on *Linux*, 23.93 on *Xen*, and 23.91 on *IO-Interpose*. We also observed that CPU utilization was slightly higher on *IO-Interpose* (36%) compared to *Linux* (33%) and *Xen* (35%). Overall, these results suggest that the user experience

of modern multimedia file playback is not noticeably degraded by I/O interposition.

	Audio Output		Video Output	
	Time(s)	CPU (%)	Time	CPU (%)
Linux	1.22	1	47.57	50
Xen	1.28	1	47.73	50
IO-Interpose	1.27	1	47.77	51

Table 5.5: Impact of I/O interposition on audio and video output generation.

We used the *Mplayer* command line options for controlling output during playback to measure the time and CPU utilization of generating only the audio output (with video output disabled) and the video output (with audio output disabled) of the movie trailer. The results are presented in Table 5.5. As shown in the table, the audio output takes about 4% longer to generate on *IO-interpose* compared to *Linux*, while the video output generation times were identical. We also observed that CPU utilization was identical on all three systems for audio output generation, but was higher in *IO-Interpose* for video output generation. Also, the performance on *Xen* was identical to *IO-Interpose*, which suggests that the observed overheads of *IO-Interpose* are mostly due to CPU and memory virtualization.

5.4.3 Network performance

We used client/server experiments to study how I/O interposition affects network server performance. We conducted these set of experiments using different client workloads to understand the impact of I/O interposition under different conditions, such as when the server is lightly and heavily loaded. For each experiment we report the median of 10 runs.

Apache We used *ApacheBench*, the standard benchmarking tool for the *Apache* webserver, to measure how the web server's transfer rate is affected by I/O interposition. We configured a

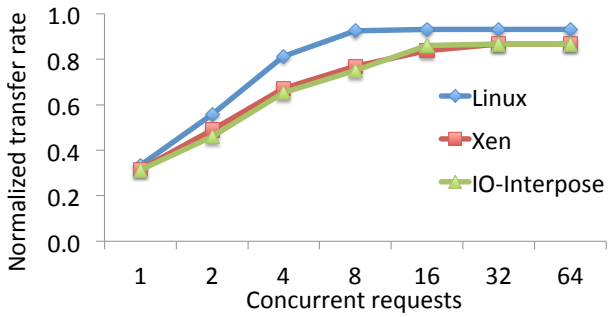


Figure 5.4: Transfer rates of *Apache* server; normalized to network card link rate (1 Gbps).

single *ApacheBench* client to load the *Apache* server with $16,000$ requests for a 40 KB static page while varying the number of concurrent requests. The parameter settings used for the experiments are shown in Table 5.3.

Figure 5.4 presents, for various degrees of client concurrency, the transfer rates achieved by the server normalized to the link rate of the server's network card (i.e., 1 Gbps). Although the peak transfer rate on *Linux* is achieved with 64 concurrent requests, we observed that the server's network card is practically saturated by 16 concurrent requests. At this saturation point the server throughput is about 7% lower on *IO-Interpose* (and *Xen*) compared to *Linux*. We also observed that *IO-Interpose* performs similarly to *Xen* for different concurrency levels of client requests. Finally, we observed that *IO-Interpose* does not hinder the server's ability to scale with increased client load, since transfer rate improves with increasing concurrency of client requests.

Figure 5.5 reports the CPU utilization of the *Apache* server for these experiments. As can be seen in Figure 5.5, *IO-Interpose* consumes noticeably more CPU cycles than *Linux* in general. For example, the *Linux* server is consuming 35% of the CPU by the time it's network card is saturated (i.e., with 16 concurrent requests). In comparison, the CPU utilization of *IO-Interpose* server is 58% at this saturation point. As expected, we observed that CPU utilization on *Xen* is higher than *Linux* but lower than *IO-Interpose*. Moreover, we also observed *Xen* accounts for more than half of the increased CPU consumption of *IO-Interpose* relative to *Linux*.

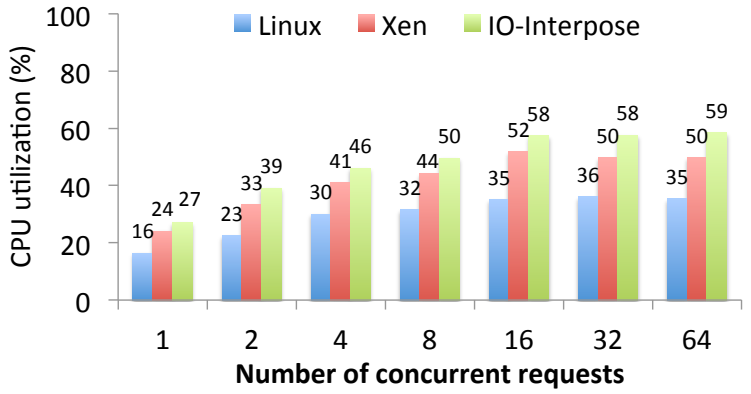


Figure 5.5: CPU utilization of *Apache* server.

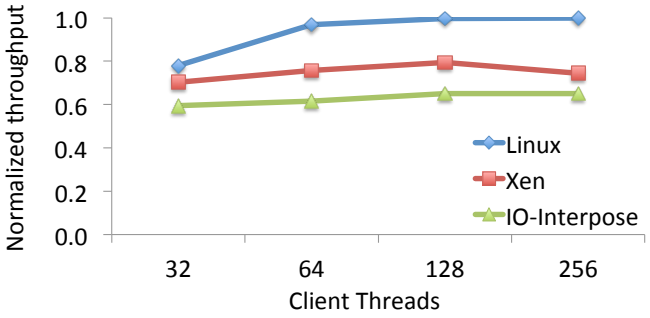


Figure 5.6: Throughput of *Memcached* server; normalized to peak throughput on *Linux*.

Memcached We evaluated the impact of I/O interposition on *Memcached's* throughput using *Memslap* (version 1.0), the server's standard benchmarking tool. We ran a single instance of *Memcached* server on the test machine with 512 MB of physical memory reserved for object storage. The client load was generated by configuring *Memslap* with the parameter settings shown in Table 5.3. *Memslap* initially loads the object storage with $100K$ objects, and then uses a configurable number of threads to make *get* requests, with each thread issuing $100K$ requests. The key and value sizes were hard-coded in this *Memslap* distribution to be 100 and 400 bytes respectively.

Figure 5.6 reports the server throughput for different number of client threads and normalized to the peak *Linux* throughput. The peak *Linux* throughput was achieved with 256 client threads,

at which point the server got saturated[5]. We observed that on *IO-Interpose*, server throughput improves, albeit modestly, with increasing client threads. With 256 client threads, *IO-Interpose* achieves 35% lower throughput compared to *Linux*, while *Xen* achieves 26% lower throughput[6]. Thus, about 75% of *IO-Interpose* can be attributed to *Xen*.

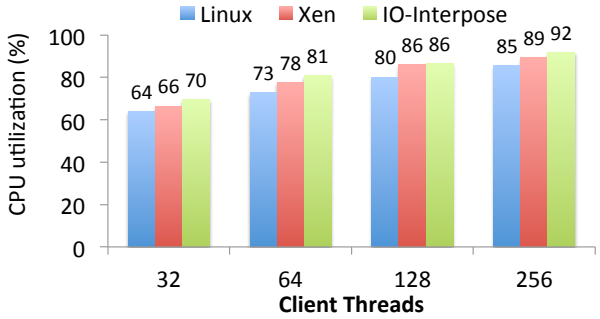


Figure 5.7: CPU utilization of *Memcached* server.

Figure 5.7 presents the CPU utilization of *Memcached* on the three server systems during these experiments. As shown in the figure, CPU utilization is higher on *IO-Interpose* compared to *Linux* in all cases, while CPU utilization on *Xen* roughly falls midway between the two. For example, at the saturation point (i.e., 256 client threads), *IO-Interpose* is using 92% of the CPU, while *Linux* is using 85% and *Xen* is using 89%.

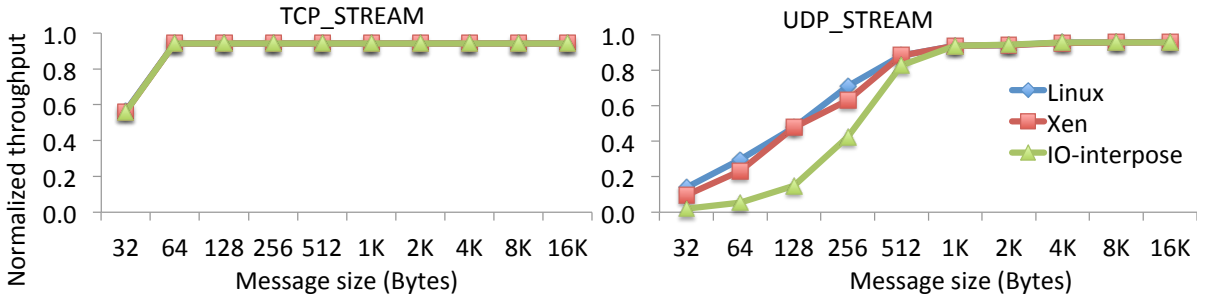


Figure 5.8: *Netperf* streaming throughput; normalized to network card link rate (1 Gbps).

[5]At the saturation point, *Memcached* was using only 50% of the network card's link rate (1Gbps).

[6]*Xen* peak throughput, which is 21% lower than *Linux*, is achieved with 128 client threads.

Netperf *Netperf* is a benchmark for measuring different aspects of network performance under different transport protocols (e.g., TCP and UDP). We used it to measure how server throughput and transaction rate are affected by I/O interposition. *Netperf* provides *stream* tests (TCP_STREAM and UDP_STREAM) for measuring throughput and *request/response* tests (TCP_RR and UDP_RR) for measuring transaction rates. In each experiment, we ran a single instance of each test from the client for 20 seconds with various message sizes (32B–16KB) and using default values for other parameters.

Figure 5.8 reports the observed server throughput during the experiments normalized to the link rate of the network card. As we can see in the figure, I/O interposition (and virtualization) had negligible impact on TCP_STREAM since *IO-Interpose* (and *Xen*) achieve similar throughput as *Linux* for different message sizes. In contrast, we observed I/O interposition overheads are noticeable with UDP_STREAM for messages that are smaller than 512 bytes (up to 33% relative to *Linux* for 64 byte messages). UDP_STREAM is CPU intensive and as depicted in Figure 5.9 doubles CPU utilization for small messages compared to larger ones. Moreover, CPU utilization for TCP_STREAM is about half that of UDP_STREAM for small messages. As a result, *Linux* and *Xen* also suffer throughput degradation for small messages with UDP_STREAM,although to a lesser degree compared to *IO-Interpose*.

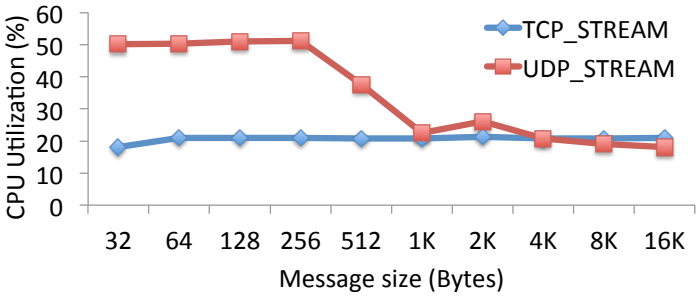


Figure 5.9: *Netperf* CPU utilization on a *Linux* server.

The results from the request/response experiments are presented in Figure 5.10. The figure shows the achieved transfer rates normalized to the best *Linux* throughput for different sizes of

the request. For TCP_RR and UDP_RR, *IO-Interpose* and *Xen* achieve similar transfer rates for most message sizes. Thus, we can conclude that interposition (and virtualization for that matter) have minimal impact on the request/response transfer rates.

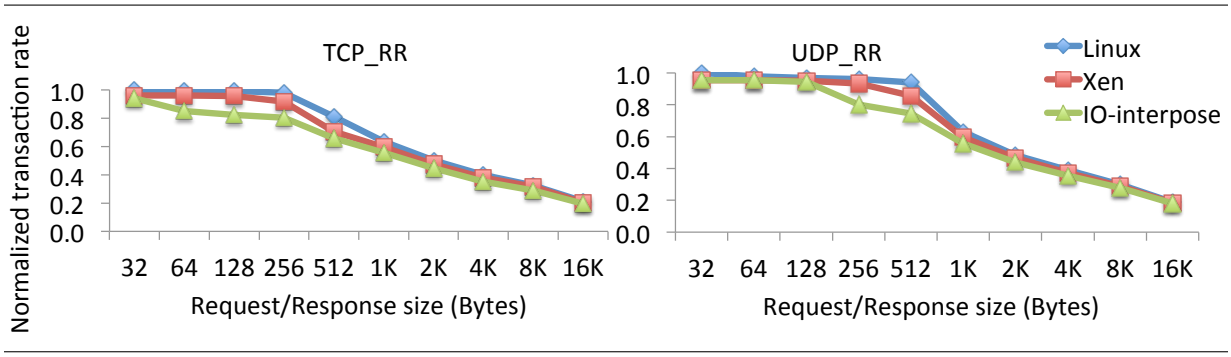


Figure 5.10: *Netperf* request/response transfer rate; normalized to peak transfer rate on *Linux*

5.4.4 Storage performance

We now describe our evaluation of the impact of interposition on disk storage performance using a kernel compilation workload and the *Postmark* benchmark.

GNU Make The disk I/O workload for this experiment was generated by using the GNU *Make* utility to compile a stock Linux 2.16.8 kernel with default configuration options. We measured how I/O interposition affects compilation time and CPU utilization. The compiler used for compiling the kernel in this experiment was a stock *GCC* 3.4.6 compiler that was distributed with Fedora Core 6 OS on the test system. We leveraged the parallel compilation feature (`-j`) of *Make* to evaluate the scalability of I/O interposition on the dual-processor test system.

In Figure 5.11, we present the kernel compilation times for *Linux*, *Xen*, and *IO-Interpose* with different degrees of parallelism and normalized to the best *Linux* result. We observed for each system that compilation time improved with increased parallelism and that this improvement peaked at 4-way parallel compilation. Some intuition for this improved performance can

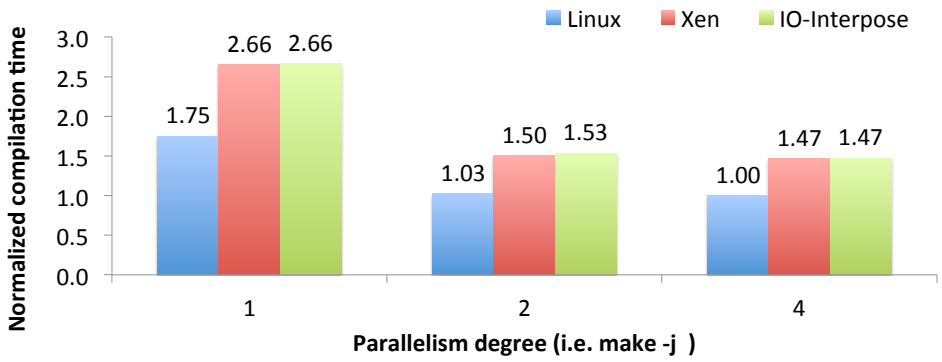


Figure 5.11: Kernel compilation time; normalized to best time on *Linux*.

be gleaned from Figure 5.12, which shows that *Make* exploits available parallel computing resources. For example, CPU utilization on *Linux* increased from 51% for sequential compilation to 86% for 4-way parallel compilation, while the increase was 53% to 93% for *IO-Interpose*. These results show that similar to *Linux* and *Xen*, *IO-Interpose* enables the compilation workload to improve performance through increased parallelism in computing resources.

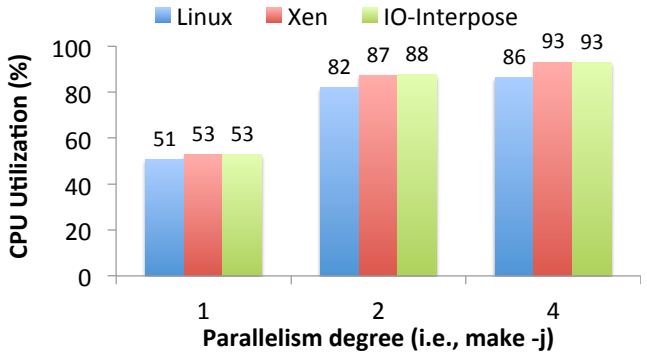


Figure 5.12: CPU utilization of kernel compilation.

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

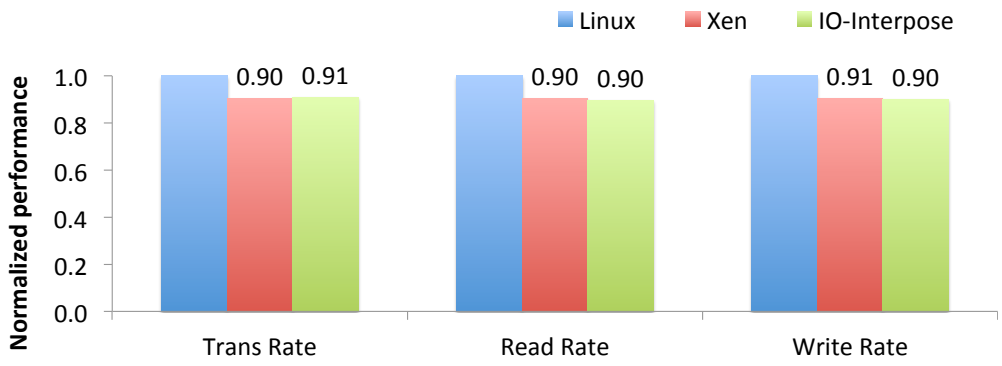


Figure 5.13: *Postmark* transaction, read, and write rates; normalized to rates on *Linux*.

Postmark *Postmark* [43] is a single-threaded benchmark that simulates the behavior of an Internet e-mail server. As shown in Table 5.3, we configured *Postmark* to perform $100K$ file transaction operations (create, delete, read, write) on $20K$ files whose sizes ranged from $10KB$ to $20KB$. All other parameters were set to their default values. We report the median of 10 runs.

In Figure 5.13 we report the measured transaction, read, and write rates normalized to *Linux*. The figure shows that *IO-Interpose* degrades the transaction rate by nine percent and the read and write rates by 10% relative to *Linux*. Similar to kernel compilation, *Postmark* does not perform noticeably better on *Xen* compared to *IO-Interpose*. The CPU utilization of *Postmark* on *Linux*, *Xen*, and *IO-Interpose* were 7%, 11%, and 8% respectively.

5.4.5 I/O Interposition performance summary

We used different types of I/O intensive workloads to measure the performance impact of Guardrail's I/O interposition layer, and observed that the overheads were modest (at most 10%) in most cases. In particular, we observed that the quality of audio and video playback using *Mplayer* is virtually unaffected by I/O interposition. Other benchmarks for which we observed similarly low overheads include: (i) *Apache*, (ii) all the *Netperf* tests except UDP_STREAM with small messages, and (iii) *Postmark*. Moreover, Guardrail's I/O interposition did not limit the ability of I/O intensive programs to leverage the available parallelism in a system to improve performance.

On the other hand, *Memcached* and kernel compilation incurred significant overheads with I/O interposition, 35% and 47% respectively.

Driver	Read slowdown	Write slowdown
ahci	3.7	34.47
nvidia	3.75	37.77
snd_hda_intel	5.66	48.98
tg3	1.79	35.48

Table 5.6: The overheads of trapping and emulating device register access.

Explaining I/O Interposition Overheads We observed that the CPU and memory virtualization of the underlying Xen VMM accounted for a significant portion of our Guardrail prototype overheads: about 75% of *Memcached* overheads and basically the entire compilation overheads. We further observed that impact of I/O interposition on a workload's performance depends on the rate of device register access and on the workload's CPU utilization in the non-virtualized system.

Frequent device register accesses reduce Guardrail performance because such accesses are handled through expensive trap and emulation operations. Table 5.6 reports the average slowdowns of device register accesses when they are trapped and emulated by Guardrail. As can be seen in Table 5.6, device register reads are 1.79X to 5.66X slower, while device register writes are 34.47X to 48.98X slower. The overhead of interposing on device register accesses is exacerbated when the CPU utilization of the workload is high (over 50%) in the non-virtualized environment. This is because there are fewer idle CPU cycles that can be used to perform the required traps and emulations without degrading overall performance.

The rates of device register reads and writes performed by drivers for different benchmarks are shown in Figure 5.14. The reported numbers are from the best performing *IO-Interpose*

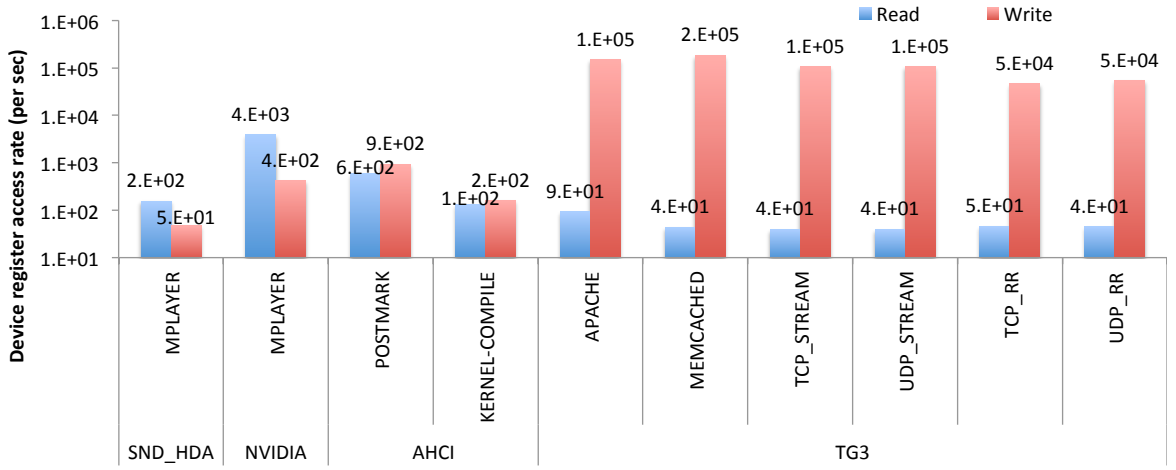


Figure 5.14: Rate of device register accesses by device drivers for different I/O workloads.

configuration for each benchmark (e.g., *Memcached* with 256 client threads). Figure 5.14 shows that *Memcached* generates the highest rate of device register accesses, especially writes (about 184K writes per second). This high rate of device register writes and the high CPU utilization of *Memcached* (i.e., 85% on a *Linux* server with 256 client threads) explain the relatively high overheads of the workload on *IO-Interpose*. In contrast, *Apache* and the *Netperf* streaming workloads which generate comparably frequent device register writes experience relatively lower overheads on *IO-Interpose* because of their low CPU utilization in *Linux*. *Apache* performs 148K device register writes per second and consumes 35% of the CPU, while *Netperf* streaming performs about 106K device register writes and consumes 20% of the CPU. Thus, I/O workloads that exhibit a combination of frequent device register accesses and high CPU utilization in the non-virtualized environment are prone to high overheads on *IO-Interpose*. In contrast, workloads that infrequently access device registers (e.g., storage and multimedia workloads) or that incur low CPU utilization (e.g., *Apache* server) should have good performance on *IO-Interpose*.

5.5 Summary

Interposing on the I/O operations of kernel-mode drivers is an effective technique for protecting persistent I/O state from driver bugs. This chapter described how we used commodity virtualization technology to design and implement a *transparent* and *trustworthy* I/O interposition layer in Guardrail. Guardrail's I/O interposition is designed to *flexibly* coordinate with decoupled correctness checking of driver execution to mitigate driver bugs on-the-fly. We used I/O intensive benchmarks to show that the performance of common I/O workloads (audio, video, network, and storage) is modestly impacted by interposition in most cases. We observed that the exceptional cases of high performance overheads (e.g. *Memcached*) were due to I/O workloads that result in frequent device register accesses and have high CPU utilization in the non-virtualized environment. Fortunately, our evaluation showed that this behavior is not exhibited by most popular I/O workloads, therefore Guardrail's I/O interposition will typically offer good performance.

Chapter 6

Mitigating Driver Bugs through Decoupled Dynamic Analysis

A key advantage of Guardrail over binary instrumentation is that it reduces the overhead of fine-grained runtime detection of driver bugs by *decoupling* correctness checking from driver execution. This chapter presents the design, implementation and evaluation of Guardrail's decoupling of dynamic analysis of drivers. We evaluate the performance implications of using Guardrail for the online protection of persistent I/O state from corruption by defective drivers.

Guardrail exploits hardware extensions to efficiently stream the execution trace of a driver to the decoupled checking tool. Since driver execution occurs intermittently, Guardrail leverages commodity interrupt mechanisms for optimal scheduling of the checking tool to ensure efficient monitoring without wasting computing resources (e.g., CPU cycles). For evaluation purposes, we prototyped Guardrail using a simulated x86 chip multiprocessor (CMP) system that we extended with the proposed hardware tracing components. Our evaluation using production Linux network and storage drivers showed that Guardrail's decoupling does not introduce additional overheads besides that of the analysis. Furthermore, we observed that Guardrail safeguards the integrity of network and storage I/O operations from corruption by memory faults, data races and *DMA* faults in drivers with minimal overheads on common I/O-intensive workloads in most cases.

6.1 Decoupled Program Monitoring

The growing popularity of CMP systems, with ever increasing processor count, has made decoupling a promising technique for accelerating online correctness checking of programs. Traditional program monitoring involved (binary/source) instrumentation of untrusted code [13, 51, 58] such that monitored and monitoring codes alternatively execute in the same thread context (a.k.a. *coupled* dynamic analysis). A key benefit of coupled dynamic analysis is that faulty instructions can be identified before they execute, which makes it easier to contain any harmful side-effects. However, coupled dynamic analysis leads to the multiplexing of processor resources (e.g., cycles, registers, caches, etc.) by the monitoring and monitored execution, which results in (sometimes significant) degradation of monitored program performance. The amount of slowdown experienced by the monitored program depends on the granularity and sophistication of the analysis, e.g. instruction-grained dynamic analysis typically incurs orders of magnitude program slowdown. In contrast, a decoupled approach allows both program and analysis to run simultaneously using separate processing resources (e.g., CPUs, registers, caches etc.), thus reducing monitoring overhead [20, 23, 60, 82]. Specifically, decoupling removes analysis computation from the monitored program's execution paths, which leads to significant reductions in program slowdown, especially for heavyweight analysis like data race detection.

However, the performance benefit of decoupling comes at the cost of increased complexity for the following program monitoring issues: (i) timeliness of fault detection, (ii) observing the monitored execution, and (iii) scheduling the execution of analysis code. We examine these issues in further details below.

Delayed Fault Detection. A fundamental complexity of decoupled monitoring arises from the delay between when an instruction in the monitored program (e.g., a memory access instruction) executes, and when it is checked for errors by the checking tool. As a result, faults in the monitored execution are detected after they occur. For heavyweight analysis tools such delays could

run into thousands of cycles because these tools execute many analysis instructions per monitored program instruction. During this window of vulnerability, the monitored program continues to execute in a buggy state with the potential of causing irreparable damage. Therefore, decoupling increases the difficulty of containing the harmful effects of software faults, compared to coupled analysis, where faulty instructions are identified (and squashed) before they execute.

Observing Monitored Execution. To detect specific kinds of program errors, dynamic analysis needs to examine the monitored execution state at an appropriate granularity (e.g., instruction-by-instruction). In the coupled dynamic analysis approach, analysis code executes in the same thread context as the monitored code, and can obtain the required information (e.g., register values) directly by reading physical registers and memory locations. Moreover, commodity instrumentation frameworks like *DynamoRio* [13], *Pin* [51], and *Valgrind* [58], often provide this information for the convenience of checking tools. However, since decoupling executes the monitoring code in a separate process context, direct access to the monitored execution state is lost. Rather, the standard approach is to use software [20, 60] or hardware [82] logging to stream the monitored execution state to the checking code. Logging application execution, especially for instruction-grained analysis, introduces the challenge of efficiently managing the bandwidth and storage requirements of the execution trace.

Scheduling Analysis Execution. Instrumentation frameworks carefully place checking code to ensure that monitoring overheads are incurred only for specific operations (e.g., memory access) and in specific execution modes (e.g., user-mode). However, with decoupled monitoring the decision of when to invoke correctness checking depends on the arrival rate of execution events at the consumer end of the log queue. The arrival rate depends on a number of factors, including buffering delays in the log and whether the monitored execution is CPU or I/O bound. Nevertheless, to avoid significant slowdowns, correctness checking must be carefully scheduled to minimize the latency of log consumption without wasting system resources. For example, if

the log is rarely empty (e.g., a CPU-bound execution), *polling* the log for new events is an efficient way to minimize log consumption latency. In contrast, if the log is often empty (e.g., an I/O bound execution), *polling* significantly wastes CPU cycles without much performance advantage compared to *sleeping* when the log is empty.

6.2 Decoupled Mitigation of Driver Bugs

The previous section discussed the performance benefits of decoupled dynamic analysis, as well as the three challenges that it introduces to program monitoring. Since Guardrail employs decoupled dynamic analysis to mitigate kernel-mode driver bugs, it needs to address those challenges in the context of kernel-mode execution. First, Guardrail must handle the inherent risks of permitting privileged code, such as a driver, to continue running for a while after executing a bug. Second, Guardrail must ensure that the decoupled checking tool can observe the execution state of the monitored driver for correctness checking. Finally, Guardrail must decide the appropriate points in time to execute the checking tool given the intermittent nature of driver execution. Figure 6.1 illustrates the current Guardrail prototype, and highlights the main system components that are used to address these three challenges.

With regards to the delayed detection of driver bugs, Guardrail ensures that persistent I/O state and the dynamic analysis tool cannot be corrupted by the harmful effects of driver bugs. As described earlier in Chapter 5, this is achieved by transparently interposing on the I/O operations of the driver, and running the dynamic analysis tool in a separate virtual machine (a.k.a. AnalysisVM) from the driver's virtual machine (a.k.a. DriverVM).

Guardrail enables the decoupled analysis tool to observe the execution state of a monitored driver—for correctness checking— by streaming a trace of the driver's execution to the analysis tool. To avoid the overhead of software-only tracing, our current prototype assumes hardware assistance for tracing driver execution. The trace is consumed on-the-fly by the analysis tool, which uses dynamic binary analysis to identify bugs in the driver's execution.

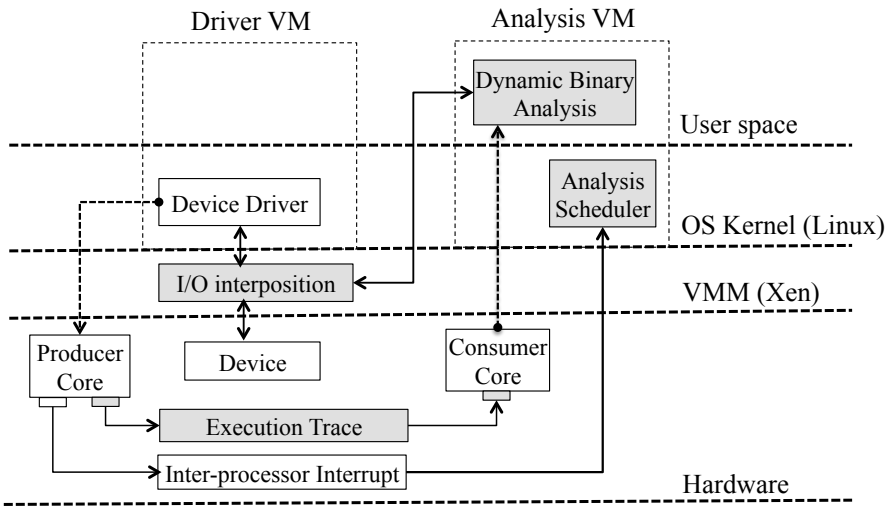


Figure 6.1: Guardrail prototype.

Because driver execution is typically sporadic in nature (e.g., responding to device interrupts), the analysis tool might sometimes be idle, and waiting for driver execution to check. Guardrail exploits standard interrupt mechanisms to avoid running the analysis tool during such idle periods, while ensuring that the execution trace is consumed with minimum latency. This objective is realized through a kernel module called *Analysis scheduler* for scheduling the execution of analysis threads in the AnalysisVM.

Since Guardrail's I/O interposition technique was described and evaluated in considerable detail in Chapter 5, we will not discuss it any further in this chapter. Instead, we will focus on the other two challenges in the the rest of this chapter.

6.3 Tracing Driver Execution for High-Fidelity Bug Detection

Guardrail enables fine-grained correctness checking of multithreaded driver execution by streaming a detailed trace of the driver's execution, containing instruction-level events (e.g., memory accesses) and shared memory dependencies, to the decoupled dynamic analysis tool. Providing this functionality requires Guardrail to do the following: (i) identify driver execution within

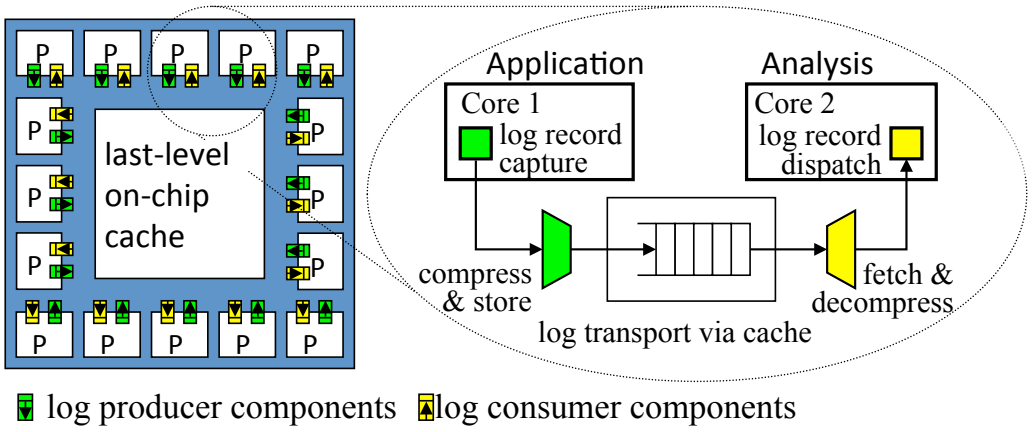


Figure 6.2: LBA on a chip multiprocessor system.

kernel-mode execution for tracing, (ii) identify concurrency within driver execution, and (iii) assign buffers for streaming parallel traces of driver execution. To help Guardrail efficiently meet these requirements, we proposed hardware support for execution tracing as a set of hardware extensions to CMP systems called *Log Based Architectures* (a.k.a. LBA) [16, 84].

In the rest of this section, we provide further details on hardware-assisted tracing provided by LBA and the corresponding software support in Guardrail. Then, we described how Guardrail satisfies the three aforementioned requirements for tracing driver execution.

6.3.1 Log Based Architectures: Hardware-Assisted Execution Tracing

Log Based Architectures (LBA) extends processor cores for efficient streaming of the execution trace of a monitored thread, running on a *producer* core, to an analysis thread, running on a *consumer* core [16]. An illustration of the hardware extensions that LBA proposed for CMP systems is presented in Figure 6.2. The LBA extensions include *log producer component*, for capturing instruction-level execution events of the monitored thread, and *log consumer component*, for dispatching the event handling routines of the analysis. To enable flexible scheduling of monitored and monitoring threads, as opposed to pinning the threads to specific cores, each processor core in the system is extended with log producer and consumer components. A circular

log buffer in the last level on-chip cache is used to stream the compressed trace to the consumer core and reduce producer-consumer synchronization overhead. We further extended LBA through *Paralog* [84] for efficient monitoring of multithreaded execution. Paralog introduces an *order capturing component* for capturing the shared memory dependencies of the monitored threads, and an *order consuming component* for ensuring that those dependencies are respected by the checking tool. The order capturing component captures shared memory dependencies by tracking context switch and cache coherent events.

6.3.2 Software Support for LBA

For monitoring flexibility, LBA hardware for execution tracing should be managed by software. Examples of critical tracing related tasks that would benefit from software control include: (i) reserving (and protecting) physical memory for use as log buffers, (ii) enabling the log production and consumption components when needed, and connecting them to the correct log buffers, and (iii) optimal scheduling of monitored and monitoring threads. Since Guardrail monitors kernel-mode drivers the guest OS kernel in the DriverVM cannot be trusted with this responsibility since its integrity can be compromised by defective drivers. Instead, Guardrail relies on the virtualization layer, which executes in a more privileged context, to provide the required software support. We implemented the desired software support in our current Guardrail prototype by extending the Xen virtual machine monitor. In particular, the virtual machine monitor reserves log buffers from the physical memory pool of the DriverVM, and protects them using the page protection mechanisms available in commodity processors. It also configures the logging components of the physical CPUs appropriately when scheduling the DriverVM and AnalysisVM.

6.3.3 Identifying Driver Execution within Kernel-Mode Execution

Two important but conflicting issues in decoupled correctness checking are the precision of the checking tool (i.e., soundness and completeness) and the storage and bandwidth costs of execu-

tion tracing. Avoiding precision problems often requires fine-grained (instruction-level) tracing of the monitored code, which incurs high execution tracing costs. Thus, for efficient decoupled monitoring it is important to strike a good balance between both concerns by tracing only execution events (e.g., memory accesses, synchronization operations) that are relevant to bug detection. For example, monitoring an application for memory access violations basically requires tracing memory loads, stores, and (de)allocations by the application's code—system libraries and the OS kernel are typically trusted in such a scenario. Similarly, to detect memory access violations in kernel-mode drivers, Guardrail traces the corresponding execution events (e.g., memory loads) by the monitored driver's code to the exclusion of other kernel code (and user-level codes)[1].

This comprehensive yet efficient tracing approach requires identifying the execution events of monitored code. For application monitoring this relatively easy because standard process (and thread) identifier information can used to perform the needed identification during user-mode execution. However, since a kernel-mode driver shares the kernel process with other modules (e.g., other drivers) and core kernel code(s) (e.g., the thread scheduler), its instructions cannot be distinguished from other kernel-mode instructions in this manner. Instead, Guardrail uses the entry and exit points in a driver's code to identify its instruction stream for tracing during kernel-mode execution. We have implemented this in our current prototype through special registers for recording the address region into which the monitored driver code is loaded, and by checking the targets of control transfer operations in kernel-mode execution against this region. The potential downside of checking each kernel-mode control transfer operation against an arbitrarily long list of address ranges can be mitigated in practice in a couple of ways. The first is to run and thus monitor only a few untrusted drivers in each DriverVM. Alternatively, the special registers could be used to track the core kernel code region(s) so that any execution outside that region is monitored.

[1] As described in Section 3.3, the interface between the kernel and the driver is also logged.

6.3.4 Detecting Concurrency in Driver Execution

As described in Section 4.2.1, there are two basic sources of concurrency in kernel drivers: (i) multithreading, and (ii) kernel execution contexts (e.g., Linux *interrupt* and *process* contexts). To ensure that driver bugs are efficiently detected with precision the shared memory dependencies resulting from the concurrency in driver execution must be tracked accurately.

Guardrail leverages Paralog mechanisms to handle concurrency that is introduced by multithreaded execution of driver code. In particular, the order consuming component captures the shared memory dependencies of the different kernel threads that execute driver code. Also, the checking tools in Guardrail can rely on the order enforcing component to ensure that the execution traces of the driver are analyzed in the right order. On the other hand, the order consuming component is ill-suited for shared memory dependencies that originate from kernel execution contexts. This is because such dependencies may not be accompanied by context switch or cache coherence events, since only one kernel thread is involved. However, since this issue appears to only affect the precision of concurrency error checking, Guardrail does not explicitly handle this problem, but rather it expects each checking tool to address the issue as the tool deems fit (e.g., Section 4.2.1). However, Guardrail provides some assistance for tools that need to address this problem by tagging the logged execution events with kernel execution context information.

6.3.5 Assigning Log Buffers for Tracing Parallel Driver Execution

Another important design consideration is the assignment of log buffers for streaming the execution trace of the monitored driver. The design choices that we considered were to assign a distinct buffer to each monitored kernel thread, to the monitored kernel process, or to the producer CPU. Due to the characteristics of kernel-mode execution, we recognized the last option to be the most practical driver monitoring solution for two reasons. First, a single log buffer for the kernel process would make it difficult to realize parallel log generation (and consumption) when there is an abundance of CPUs to support concurrent monitored threads and concurrent

monitoring threads. Second, maintaining a log buffer per kernel thread is not scalable because there could be an unbounded number of such threads, and is potentially wasteful because driver execution accounts for only a small fraction of the lifetime of such threads.

In contrast, per-CPU log buffers is not the optimal choice for monitoring multithreaded applications. This is because multithreaded application execution involves a relatively small (at most 100s) and fixed set of threads whose lifetimes closely match that of the application, and which spend most of their time executing monitored (i.e., application) code. Since supporting a few hundred log buffers in order to enjoy the improved monitoring performance of parallel log production and consumption is conceivable, per thread buffer assignment would be the most optimal choice.

In summary, Guardrail reserves a log buffer per virtual CPU in the DriverVM to trace parallel execution of the monitored driver by kernel threads. This enables Guardrail to enjoy the performance benefits of parallel log generation and consumption, while avoiding the scalability issues of driver execution by an unbounded number of kernel threads.

6.4 Scheduling Decoupled Dynamic Analysis

Dynamic analysis code is often structured as a collection of routines for handling various execution events (e.g., memory access, control flow transfer, etc.). Thus, the computation pattern of a decoupled analysis thread on a consumer core is a cycle of *decoding* log events, and *dispatching* the appropriate handler routines. LBA provides hardware for performing the *decode-dispatch* sequence to avoid the overheads of a software approach. However, if the analysis thread is not scheduled (i.e., running) at this point in time the performance benefits could be overshadowed by context switching overhead. On the other hand, running an analysis thread when there is no event to process in the near future is a waste of system resources (including energy). Therefore, analysis threads should be scheduled only when new log events are or would soon be available for processing, and be descheduled otherwise (i.e., when the log buffer is empty).

6.4.1 Idleness in Analysis Threads

A key consideration in scheduling decoupled analysis threads is estimating how long a log buffer will be empty. An accurate estimate will help determine when to deschedule and later reschedule the thread. Unfortunately, estimating future events (e.g. monitored execution behavior) accurately is generally impossible. However, on closer examination, it becomes apparent that this problem could be framed as an instance of asynchronous event notification (e.g., arrival of network packets on a network card). That is, an analysis thread in the AnalysisVM should be notified when the monitored driver resumes execution in the DriverVM. Thus, standard techniques such as *polling* and *interrupt* are promising approaches for this problem.

6.4.2 Polling

With *polling*, analysis threads *busy-wait* for new log entries to process rather than yield the CPU when the log queue becomes empty. By avoiding thread scheduling overheads, *polling* ensures that log entries are processed as soon as possible. However, this benefit results in the waste of CPU cycles, which could be quite severe if the log is frequently empty for long periods. Driver execution typically exhibits this behavior because drivers account for a small portion of the execution paths of I/O-bound applications; user-mode execution is commonly the most dominant portion followed by the higher layers of the kernel-mode portion of the I/O stack. Thus, a purely *polling* approach will result in significant waste of system resources.

6.4.3 Yielding

One way to avoid such waste is for an analysis thread to yield the CPU once it's log queue becomes empty and wait for rescheduling when there is work to do. However, this raises the question of when to reschedule the analysis thread to ensure a timely processing of new log events. Note that timely rescheduling is also a relevant issue for *polling* since analysis threads are eventually descheduled on expiration of their time quantum. One possible approach using

standard OS scheduling mechanisms is for the OS scheduler to check the occupancy of the log buffer during context switching, and reschedule the analysis thread as appropriate. Unfortunately, even if the analysis threads are assigned the highest scheduling priority, the common context switch intervals (i.e., time quantum) of modern operating systems are too large (i.e., tens or hundreds of milliseconds) to guarantee timely rescheduling. In other words, a log event that arrives in a previously empty log queue could wait an entire time quantum before being noticed by the OS scheduler.

6.4.4 Interrupts

Guardrail signals the availability of new entries in a log queue by *interrupting* the AnalysisVM and scheduling (via the *Analysis scheduler*) the corresponding analysis thread to execute as soon as possible. This technique works as follows. Whenever the log producing component hardware captures an execution event it also signals a "log event" interrupt to alert the *Analysis scheduler* of the impending arrival of a log event. This interrupt can be delivered to the AnalysisVM through standard inter-processor interrupt (IPI) mechanisms that are available on commodity hardware. In response, the *Analysis scheduler* schedules the analysis thread, and masks the "log event" interrupt line to prevent further delivery of the interrupt. Notice that due to log buffering and transfer delays, signaling the interrupt when the log event is produced as opposed to when it arrives at the consumer core (partially) hides the overhead of rescheduling the analysis thread. A performance improvement which we have not evaluated in our current Guardrail prototype is to reserve distinct "log event" interrupt lines for each producer core. This enables selective (un)masking of "log event" interrupts on producer cores (e.g., while the corresponding the analysis thread is (de)scheduled).

6.5 Evaluation

We evaluated the impact of using Guardrail to monitor driver execution on the end-to-end performance of common I/O workloads. Our evaluation focused on measuring the overheads of using Guardrail for online protection of persistent device state from corruption by the following correctness issues in drivers: (i) memory faults, (ii) data races, and (iii) DMA faults. We ran Guardrail in *permissive* mode (Section 3.2) to ensure that the experiments ran to completion without being abruptly halted for detected driver faults. Recall that *permissive* Guardrail stalls the I/O operations of the monitored driver until the decoupled analysis catches up, and then resumes execution even when it is potentially unsafe to do so (i.e., checking tool detected a fault).

Our experiments measured the performance of an I/O intensive workload with the driver being monitored by each of the three proposed Guardrail tools—*DMCheck*, *DRCheck*, and *DMACheck*—normalized to the workload's performance without driver monitoring in a non-virtualized Linux system (a.k.a. *Linux*).

6.5.1 Methodology and Experimental Setup

We employed simulation techniques to model Guardrail's proposed hardware support for instruction-grained tracing of multithreaded kernel-mode driver execution (Section 6.3). As described earlier in Section 4.5.1, we used the Simics [87] full system simulator to prototype our Guardrail design. The baseline *Linux* system and the Guardrail system were configured for the experiments using the simulation parameters that were earlier presented in Section 4.5.1 (Table 4.1). The non-virtualized client system for the network experiments was configured using the same simulation parameters. All the experiments were simulated to completion.

The simulation approach limited the our performance studies in two ways. First, we were restricted to studying only network and storage I/O performance, because models for other interesting device classes (e.g., audio, graphics, etc.) were not readily available in our simulator version. Next, we had to scale down the workload sizes in the experiments, because the simulated

I/O class	Benchmark	Version	Description
Network	Apache	2.2.6	Webserver
	Memcached	1.2.3	In-memory key value store
	Netperf	2.4.0	Network performance measurement tool
Storage	Postmark	1.5.1	Filesystem benchmark

Table 6.1: The I/O intensive benchmarks used for performance evaluation.

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

Table 6.2: Network and storage benchmark parameters.

device models were not robust enough to handle real-world input sizes. An immediate impact of reduced input parameters was that virtualization and I/O interposition overheads were negligible, thus the overheads observed in the experiments can be solely attributed to the checking tools.

Software Environment We made a best-effort attempt to reproduce the software environment that we earlier used for evaluating the performance of virtualization-based I/O interposition in commodity systems (Section 5.4), in the simulated systems for this performance study. In particular, the same software stack was used for the experiments in both the real and simulated hardware environments. The network and storage intensive benchmarks used in the experiments are illustrated in Table 6.1, while the input parameters, which we reduced for simulation environment, are presented in Table 6.2.

Drivers & Devices Even with the reduced benchmark input parameters, the only simulated device models that proved robust enough for our experiments were the Broadcom *BCM5703C*

Class	Driver	Device model
Network	tg3	BCM5703C 1Gbps NIC
Storage	sym53c8xx	SYM53C875 SCSI disk

Table 6.3: Linux drivers and simulated device models used for performance evaluation.

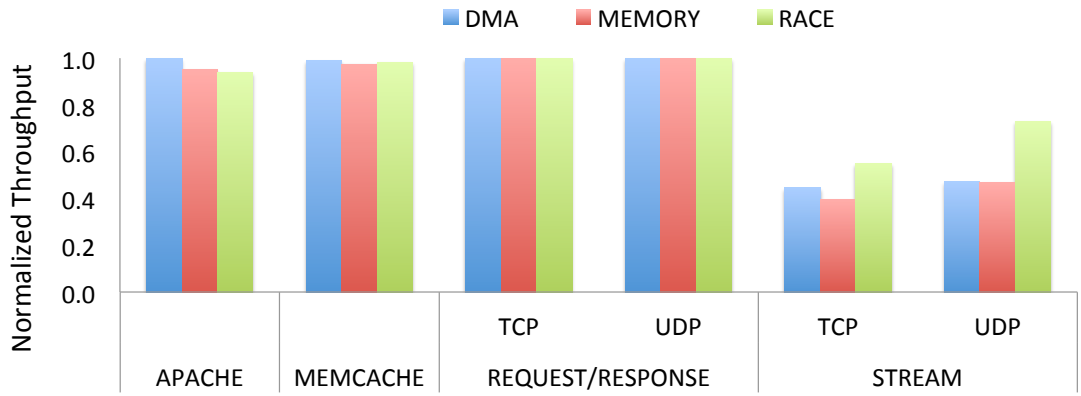


Figure 6.3: Throughput when protecting I/O operations of *BCM5703C* NIC from *tg3* faults; normalized to no protection.

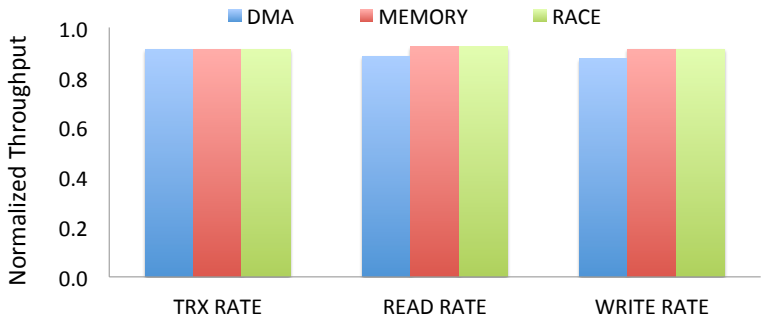


Figure 6.4: *Postmark* performance when protecting I/O operations of *SYM53C875* SCSI disk controller from *sym53c8xx* faults; normalized to no protection.

network card, and the Symbios *SYM53C875* SCSI disk controller. Thus, our performance measurements focused on using Guardrail to monitor the corresponding drivers, *tg3* network driver and *sym53c8xx* SCSI driver, as shown in Table 6.3.

6.5.2 End-to-end Performance

Using the experimental setup described above, we measured the impact on the end-to-end performance of I/O intensive workloads of using *DMCheck*, *DRCheck* and *DMACheck* to prevent driver bugs from corrupting persistent network and storage device state. For these experiments, we did not attempt to optimize the checking tools and so the following results are in fact conservative.

The impact of safeguarding the *BCM53703C* network card operations, performed by the *tg3* driver, on the throughput of different network intensive workloads is presented in Figure 6.3. The figure shows 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. 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%.

Our investigation into the significant degradation of network streaming performance suggests that the high rate of device register accesses by networking streaming, compared to other workloads, was the reason for the overheads of driver monitoring. As shown in Figure 6.5, 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 40K writes per second for *Apache* and *Memcached* 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 6.4 illustrates how *Postmark* performance is impacted by using Guardrail to protect the *SYM53C875* SCSI disk controller from *sym53c8xx* driver bugs. The figure reports the normalized read, write, and transaction rates of the benchmark, relative to running without protection. We

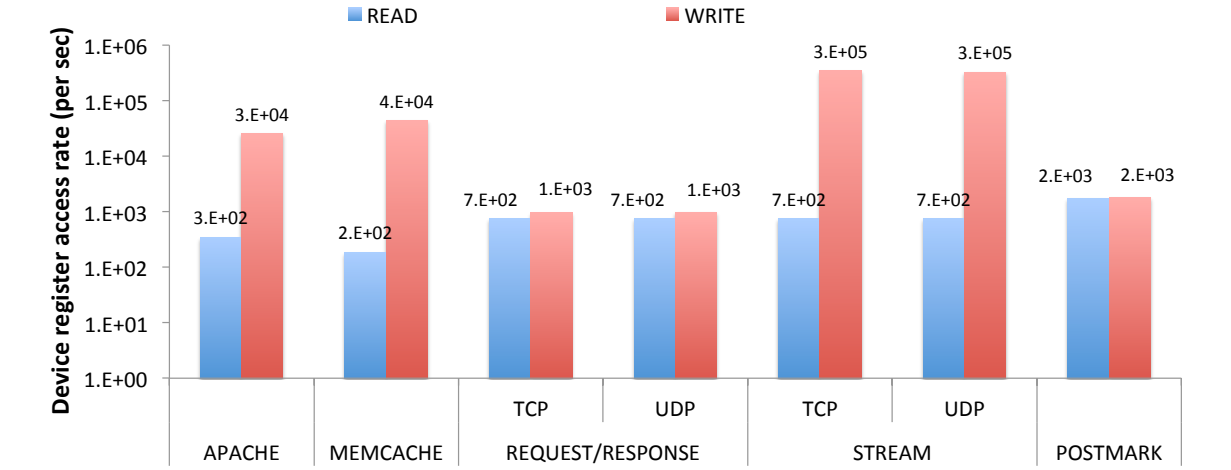


Figure 6.5: Rates of device register access performed by *tg3* driver on *BCM5703C* NIC (network benchmarks), and by *sym53c8xx* driver on *SYM53C875* SCSI disk controller (Postmark).

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 6.5, which shows that *Postmark* generates about 3K device register accesses per second, and thus leads to less frequent driver stalls.

6.5.3 Fault mitigation performance summary

Our experiments showed that online protection of the persistent state of I/O devices from subtle driver faults (e.g., memory faults, data races) can be achieved with minimal 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 [60, 68, 71, 72] and hardware [16, 84] techniques for accelerating dynamic analysis.

6.6 Summary

Guardrail mitigates the risks posed by defective drivers to I/O devices by using sophisticated correctness checking (e.g., data race detection) to detect faults in driver execution, and virtualization-based interposition to contain detected faults within the driver. Guardrail minimizes the overheads of heavyweight driver monitoring using the following two techniques. First, Guardrail leverages multi-core systems and novel hardware support for tracing driver execution to decouple and run the checking tool on a separate CPU from the monitored driver. Second, Guardrail leverages commodity interrupt mechanisms to efficiently schedule the execution of the decoupled checker. Evaluations using Linux network and storage drivers showed that Guardrail can protect persistent device state from corruption by memory faults, data races, and DMA faults in drivers with at most 10% overhead for most I/O intensive workloads. However, Guardrail monitoring reduces network streaming throughput by up to 60%.

Chapter 7

Conclusions

Device drivers are critical system software that enable users to enjoy the wide variety of functionality (e.g., persistent storage, network connectivity, entertainment, etc.) of peripheral I/O devices. Unfortunately, the high bug rate of production drivers accounts for a disproportionately high fraction of system failures, making drivers an Achilles heel of system reliability.

This thesis presented Guardrail, a framework that makes systems more resilient to defective drivers through instruction-grained correctness checking of driver execution to prevent persistent device state corruption. Guardrail leverages multi-core systems and hardware-assisted instruction-grained execution tracing to decouple correctness checking from driver execution and reduce monitoring overhead. Also, commodity virtualization is used to transparently stall I/O operations until validation by the (potentially lagging) decoupled checking tool.

Guardrail's general-purpose support for sophisticated dynamic analyses was demonstrated using three novel tools for detecting: (i) data races (i.e., *DRCheck*), (ii) DMA bugs (i.e., *DMACheck*), and (iii) memory bugs, including unsafe uses of uninitialized data (i.e., *DMCheck*) in unmodified driver binaries. Experimental results showed that these tools detect bugs better than prior work—25 bugs were detected in eight Linux network and storage drivers, including previously unknown bugs. Also, Guardrail overheads are modest (i.e., $\leq 10\%$) for most I/O workloads, with the exception of workloads that perform I/O operations at unusually high rate (i.e., $\geq 100K/s$).

A number of interesting research questions emerge from the results of this thesis. First, how can Guardrail performance be improved, especially for workloads with frequent I/O operations? Possible directions include: (i) leveraging acceleration techniques for decoupled dynamic analysis [16, 60, 68, 71, 72, 84] for faster validation of I/O operations, (ii) leveraging software [2, 36] and hardware [3, 83] virtualization support to reduce the overheads of interposing on I/O operations. Second, can Guardrail enable high-fidelity protection of the OS kernel from driver bugs using decoupled instruction-grained correctness checking? Also, can such protection be achieved efficiently given that a driver interacts significantly more frequently with the OS kernel than it does with a device? Third, can the precision of Guardrail's bug detection and mitigation be improved by monitoring the I/O protocol stack in a more comprehensive fashion (i.e., beyond the driver execution)? For example, are there invariants in the higher layers of the protocol stack or in the device state that Guardrail checking tools can exploit to avoid false positives and false negatives? These and other related research questions are the directions that we plan to pursue in our future study of how Guardrail can improve device driver reliability.

Bibliography

[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. 2.2, 4.3.4

[2] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software Techniques for Avoiding Hardware Virtualization Exits. In *Proc. of the 2012 USENIX conference on USENIX Annual Technical Conference*, 2012. 5.2.1, 7

[3] AMD. AMD Secure Virtual Machine Architecture Reference Manual, 2005. 5.3.1, 7

[4] AMD. AMD I/O Virtualization Technology (IOMMU) Specification, 2009. 2.2, 4.3.4

[5] James P. Anderson. Computer Security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972. URL `http://csrc.nist.gov/publications/history/ande72.pdf`. 5.1.2

[6] Thomas Ball and Sriram K. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, 2001. 1.2.2

[7] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging System Software via Static Analysis. In *29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, 2002. 1.2.2

[8] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 conference on*

Programming language design and implementation, Proc. ACM SIGPLAN 01 Conference on Programming Language Design and Implementation, 2001. 1.2.2

[9] 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 *1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006. 1.2.1, 1.2.2

[10] 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 *19th ACM Symposium on Operating Systems Principles*, 2003. 3.2, 5.3

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Cha mbers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *15th ACM Symposium on Operating Systems Principles*, 1995. 1.2.1, 1.2.1

[12] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proc. of the 2010 USENIX conference on USENIX Annual Technical Conference*, 2010. 1.3

[13] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004. 4.1.1, 6.1, 6.1

[14] 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 *22nd ACM Symposium on Operating Systems Principles*, 2009. 1.1, 1.3, 4.5.4, 5.2.2

[15] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011. 4.4

[16] 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 *Proceedings of the 35th ISCA*, 2008. 3.1, 3.2, 4.5.1, 6.3, 6.3.1, 6.5.3, 7

[17] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with revnic. In *5th ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2010. 1.2.1, 1.2.1, 4.4

[18] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th ASPLOS*, 2011. 4.6

[19] Andy Chou, Jufeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *18th ACM Symposium on Operating Systems Principles*, 2001. 1.1, 1.2.1, 1.2.1, 4.1.3

[20] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proc. of the 2008 USENIX conference on USENIX Annual Technical Conference*, 2008. 1.3, 3.2, 6.1, 6.1

[21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *IN COMPUTER AIDED VERIFICATION*, pages 154–169. Springer-Verlag, 2000. 1.2.2

[22] Jonathan Corbet. The source of the e1000e corruption bug. *lwn.net/Articles/304105/*, 2008. 1.1

[23] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *Proceedings of the 30th ISCA*, 2003. 1.3, 4.1.1, 6.1

[24] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a Flexible Information Flow Architecture for Software Security. In *ISCA*, 2007. 4.1.1

[25] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, 2008. 5.2.3

[26] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *19th ACM Symposium on Operating Systems Principles*, 2003. 1.2.2

[27] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System rules using System-specific, Programmer-written Compiler Extensions. In *4th USENIX Symposium on Operating Systems Design and Implementation*, 2000. 1.2.2, 4.2.4

[28] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *9th USENIX Symposium on Operating Systems Design and Implementation*, 2010. 1.1, 1.3, 4.2, 4.2.6, 4.5.2

[29] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *7th USENIX Symposium on Operating Systems Design and Implementation*, 2006. 1.1, 1.3

[30] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *Proceedings of the 17th ASPLOS*, 2012. 1.3, 3.1, 3.2, 4.4, 4.5.4

[31] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. ACM SIGPLAN 09 Conference on Programming Language Design and Implementation*, 2009. 4.1.1, 4.1.2, 4.2

[32] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP Kernel Crash Analysis. In *Proceedings of the 20th conference on Large Installation System Administration*, 2006. 1.1, 4.1.3, 4.2

[33] Vinod Ganapathy, Matthew Renzelmann, Arini Balakrishnan, Michael Swift, and Somesh Jha. The Design and Implementation of Microdrivers. In *Proceedings of the 13th ASPLOS*,

2008. 1.2.1, 1.3, 4.5.4

[34] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs. In *Proceedings of the 16th ASPLOS*, 2011. 4.2.4

[35] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The jx operating system. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, Proc. of the 2002 USENIX conference on USENIX Annual Technical Conference, 2002. 1.2.1, 1.2.1

[36] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. In *Proceedings of the 17th ASPLOS*, 2012. 7

[37] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, Proc. ACM SIGPLAN 02 Conference on Programming Language Design and Implementation, 2002. 1.2.2

[38] Martin Hirzel and Robert Grimm. Jeannie: granting java native interface developers their wishes. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, Proc. ACM SIGPLAN 07 Conference on Object-oriented Programming Systems and Applications, 2007. 1.2.1

[39] Galen C. Hunt and James R. Laurus. Singularity: Rethinking the Software Stack. In *SIGOPS ORS*, 2007. 1.2.1, 1.2.1

[40] Asim Kadav and Michael M. Swift. Understanding modern device drivers. In *Proceedings of the 17th ASPLOS*, 2012. 1.2, 2.1, 2.2

[41] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *22nd ACM Symposium on Operating Systems Principles*, 2009.

1.2.2

[42] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *ASPLOS*, 2012. 3.2

[43] Jeffrey Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, October 1997. 5.4.4

[44] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguouri. KVM: the Linux Virtual Machine Monitor. In *Ottawa Linux Symposium*, 2007. 5.3

[45] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, Princeton, NJ, USA, 1994. ISBN 0-691-03436-2. 1.2.2

[46] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing Closed-Source Binary Device Drivers with DDT. In *Proc. of the 2010 USENIX conference on USENIX Annual Technical Conference*, 2010. 1.3, 4.5.2, 4.5.4

[47] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *Proc. ACM SIGPLAN 10 Conference on Programming Language Design and Implementation*, 2010. 4.1.1

[48] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery domains: an organizing principle for recoverable operating systems. In *Proceedings of the 14th ASPLOS*, 2009. 5.2.2

[49] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, 6th USENIX Symposium on Operating Systems Design and Implementation, 2004. 1.2.1, 1.2.1

[50] Robert Love. *Linux Kernel Development (3rd Edition)*. Addison-Wesley Professional,

2010. 4.2.1

[51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klause r, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. ACM SIGPLAN 05 Conference on Programming Language Design and Implementation*, 2005. 4.1.1, 6.1, 6.1

[52] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gi lles Muller. Devil: an idl for hardware programming. In *4th USENIX Symposium on Operating Systems Design and Implementation*, 2000. 1.2.1, 1.2.1

[53] James G. Mitchell. JavaOS: Back to the future. In *1st USENIX Symposium on Operating Systems Design and Implementation*, 1996. 1.2.1, 1.2.1

[54] Brendan Murphy. Automating Software Failure Reporting. *Queue*, 2004. 1.2.1, 1.2.2

[55] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008. 4.2

[56] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, 2007. 3.2, 4.6

[57] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language, 2002. 1.2.2

[58] Nicholas Nethercote and Julian Seward. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. ACM SIGPLAN 07 Conference on Programming Language Design and Implementation*, 2007. 4.1.1, 6.1, 6.1

[59] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Anal-

ysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium*, 2005. 4.1.1, 4.1.3

[60] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th ASPLOS*, 2008. 1.3, 6.1, 6.1, 6.5.3, 7

[61] Vegard Nossum. Getting started with KMemcheck. *http://www.mjmwired.net/kernel/Documentation/kmemcheck.txt*, 2012. 4.4, 4.5.4

[62] Penny Orwick. Scheduling, thread context, and irql. *http://www.microsoft.com/whdc/driver/kernel/irql.mspx*, 2012. 4.2.1

[63] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. In *1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006. 1.2.1

[64] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008. 1.2.1, 1.2.1

[65] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *Proceedings of the 16th ASPLOS*, 2011. 1.1, 1.2.1, 1.2.1, 1.2.2, 4.1.3, 4.2

[66] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th ASPLOS*, 2009. 4.2

[67] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010. 3.1, 3.2

[68] 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 *Proceedings of Micro-36*, 2006. 6.5.3, 7

[69] Matthew Renzelmann and Michael Swift. Decaf: Moving Device Drivers to a Modern Language. In *Proc. of the 2009 USENIX conference on USENIX Annual Technical Conference*, 2009. 1.2.1, 1.2.1

[70] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: testing drivers without devices. In *OSDI*, 2012. 1.3, 4.6

[71] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing Dynamic Information Flow Tracking. In *Symposium on Parallelism in Algorithms and Architectures*, 2008. 6.5.3, 7

[72] Olatunji Ruwase, Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Decoupled lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking tools. In *Proc. ACM SIGPLAN 10 Conference on Programming Language Design and Implementation*, 2010. 6.5.3, 7

[73] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *4th ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2009. 1.1, 1.2, 1.2.1, 2.1, 2.2, 4.1.3, 4.4

[74] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic Device Driver Synthesis with Termite. In *22nd ACM Symposium on Operating Systems Principles*, 2009. 1.2.1, 1.2.1

[75] Leonid Ryzhyk, John Keys, Balachandra Mirla, Arun Raghunath, Mona Vij, and Gernot Heiser. Improved device driver reliability through hardware verification reuse. In *Proceedings of the 16th ASPLOS*, 2011. 1.3, 4.1.3, 4.7

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

4.1.1, 4.1.2, 4.1.3, 4.2, 4.2.5, 4.2.6

[77] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *Proc. ACM SIGPLAN 08 Conference on Programming Language Design and Implementation*, 2008. 4.2

[78] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitzer - Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009. 4.2

[79] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th ASPLOS*, 2004. 4.1.3

[80] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *19th ACM Symposium on Operating Systems Principles*, 2003. 1.1, 1.3, 4.5.4

[81] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *6th USENIX Symposium on Operating Systems Design and Implementation*, 2004. 1.1, 5.2.2

[82] Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. Quantifying the potential of program analysis peripherals. In *Proceedings of PACT '09*, 2009. 3.1, 3.2, 6.1, 6.1

[83] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5), May 2005. ISSN 0018-9162. 5.3.1, 7

[84] 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 *Proceedings of the 15th ASPLOS*, 2010. 3.2, 6.3, 6.3.1, 6.5.3, 7

[85] 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*, 14th ACM Symposium on Operating Systems Principles, 1993. 1.3, 4.5.4

[86] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gun Sirer, and Fred B. Schneider. Device Driver Safety through a Reference Validation Mechanism. In *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008. 1.1, 1.3, 4.3.4, 4.5.4, 5.1.2, 5.2.2

[87] Wind River Simics. http://www.simics.net/. 4.5.1, 5.3.1, 6.5.1

[88] Min Xu, Vyascheslav Malyugin, Jeffery Sheldon, Ganesh Venkitachalam, and Boris Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Determinstic Replay. In *Third Annual Workshop on Modeling, Benchmarking and Simulation*, 2007. 3.2

[89] Yaun Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adapative Tracking. In *20th ACM Symposium on Operating Systems Principles*, 2005. 4.1.1, 4.2, 4.6

[90] 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 *7th USENIX Symposium on Operating Systems Design and Implementation*, 2006. 1.1, 1.3, 4.5.4