Thesis Proposal

Log Based Dynamic Binary Analysis for Detecting Device Driver Defects

Olatunji Ruwase
Carnegie Mellon University
oor@cs.cmu.edu

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

1 Introduction

The user experience of modern computing systems is greatly enriched by the availability of hardware I/O devices that provide a rich variety of functions including data storage (e.g disk and flash drives), connection to the Internet (e.g network cards, webcams), and entertainment (e.g. speakers, GPUs). I/O devices are quite popular in today's computing environments (Handheld devices, PCs, Cloud Computing Servers) because they are quite easy to setup for use. In most cases, this involves no more than installing a *device driver*—a piece of software that is often implemented as an Operating System (OS) kernel module—to enable the OS manage the device.

However, defects in device drivers are a major source of OS failures, either by crashing the driver (and consequently the whole system) or by corrupting the kernel or applications. In the Linux kernel, the error rates of drivers are several times that of any other subsystem [12], while up to 85% of Windows XP failures [48, 24] are caused by driver errors. The relatively poor quality of driver code compared to the rest of the kernel could be attributed to a couple of factors. First, most driver developers are not kernel experts, yet they are relied on to write code that will execute correctly in modern OS kernels that rapidly grow in complexity. Second, the large number of drivers (tens of thousands) running at customer sites [32] makes it impractical for OS vendors to test all of them thoroughly.

Dynamic binary analysis has been effectively used by dynamic correctness checkers (a.k.a. lifeguards) for detecting errors in the execution of unmodified application binaries and thus improving the reliability and security of computing systems. Instruction grain lifeguards can detect the most subtle program faults including low level issues such as memory [34] and security [16, 35] errors, as well as high level ones such as concurrency errors [46, 53, 23] and interface errors in multilingual programs [30]. Lifeguards are deployed using either instrumentation or logging of the monitored application. Dynamic binary instrumentation (DBI) approaches inject lifeguard code into the application using software [7, 10, 34], or hardware [15], while logging based approaches capture an instruction grained execution trace of the application using software [13, 36] or hardware [9] for online lifeguard analysis. Dynamic binary analyses seem to be well suited for use on production drivers which are often distributed in binary form for both convenience and proprietary reasons. This thesis work explores the effectiveness of using dynamic binary analysis for online detection of production driver faults, both of the low level nature such as buffer overflows, as well as high level ones like OS protocol violations.

However, interrupts and preemption in modern OSes running on chip multiprocessor (CMP) systems pose a major challenge to the precision of instrumentation based instruction grained lifeguards since the atomic execution of an application instruction and the corresponding lifeguard code is no longer guaranteed. Signal handlers pose a similar problem for instruction grained application lifeguards. Current solutions for enforcing atomic execution of application and lifeguard code using transactional memory [14], or disabling preemption [34, 33], are not applicable for monitoring drivers, and kernel mode execution in general, since they conflict with OS preemptibility. On the other hand, this is not an issue for log based approaches which can leverage solutions for obtaining totally ordered multi-threaded execution traces on CMPs [40, 51, 26] to order lifeguard actions. However, log based approaches require error containment mechanisms due to their delayed correctness checking.

My thesis is that log based lifeguards can use dynamic binary analysis to detect defects in unmodified device driver binaries, and therefore be used for isolating applications and OS kernels from defective drivers.

In this thesis, driver faults are assumed to be developer errors and not malicious attacks, in other words the proposed lifeguards can be used to protect the OS kernel from defective drivers but not from malicious ones. The rest of this

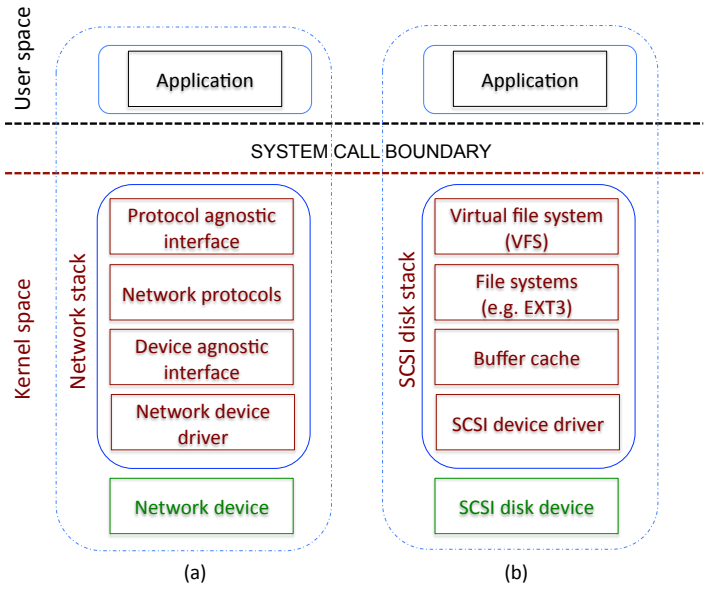

Figure 1. Device protocol stacks in Linux for (a) network devices and, (b) SCSI disk devices. The drivers execute at the bottom of the protocol stacks and interface with the devices.

proposal is organized as follows. Section 2 presents an overview of the proposed log based device driver lifeguards. Section 3 describes related work. Section 4 describes my preliminary work on logging driver execution on CMPs, and showing that application lifeguard analysis are imprecise for drivers and report false alarms. My proposed work on (i) designing lifeguards for detecting faults in driver execution, and (ii) improving lifeguard performance for online monitoring is described in Section 5. Finally, Section 6 presents my proposed schedule for completing this thesis.

2 A Log Based Approach To Isolating OS Kernels From Device Driver Defects

The main objective of this thesis work is to demonstrate that log based lifeguards can effectively detect driver faults, and can thus be used for isolating applications and OS kernels from defective device drivers. This is an important application area for log based lifeguards since driver faults severely compromise system reliability. Moreover, despite the high bug density of drivers, they (and kernel in general) have received little attention in prior log based lifeguard studies, which have almost exclusively focused on application monitoring, Aftersight [13] being the only exception that we are aware of. Before going into details about our log based fault isolation proposal for drivers, first we provide some background on device drivers to motivate why they are hard to write correctly.

2.1 Background on Device Drivers

Device drivers manage hardware devices for the rest of the system and are often implemented as loadable kernel modules. Within the kernel, drivers are organized into "device classes" based on the type of device they manage, and drivers in the same class export a standard interface to the kernel. Examples of device classes in Linux include character (e.g keyboard and mouse), block, (e.g disk), network, sound, and graphics. Drivers implement two special functions: (i) for performing initializations at load time (module_init), and (ii) for cleaning up during unloading (module_cleanup). Some defining features of driver execution, including: (i) as a device protocol stack layer, (ii) concurrency, and (iii) reentrancy, are discussed below.

2.1.1 Device Protocol Stack Layer

A device driver executes in a device protocol stack and implements the interface between the device and the rest of the kernel. It does this by exporting functions that the upper layers of the stack use to make I/O requests, and interacting directly with the device to service those requests. For example, a network driver exports functions that enable upper layers of the networking stack to use the network card for sending and receiving packets. An illustration of drivers within the protocol stacks of a network and disk device in the Linux kernel are shown in Figures [1] 1(a) and

[1] Adapted from http://www.ibm.com/developerworks/linux

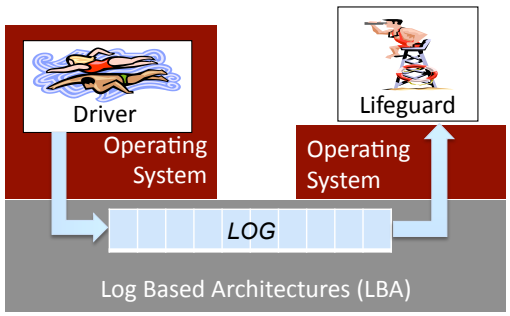

Figure 2. High level view of the proposed log based fault detection for device drivers.

1(b) respectively. Drivers, are thus similar to system libraries that execute below application code in a software stack to provide access to system resources like memory and the filesystem.

2.1.2 Multi-threaded Execution

Since a device is often a shared resource for applications and kernel, multiple independent requests can be sent to it at once. For example, all applications that are transmitting and receiving network packets use the network card. Therefore, by being multi-threaded, drivers can exploit the available parallelism in the device requests to improve system performance. Drivers for high performance devices like network and graphics cards are multi-threaded to exploit available compute resources on CMPs. Even on uniprocessor systems, multi-threaded execution of driver code is possible in modern OSes due to preemption and device interrupts. For example, a network card generates interrupts to indicate packet reception which leads to invocation of the interrupt handler exported by its driver, to transfer the packet from the device to the upper layers of the network stack. Thus driver execution by kernel threads, can be initiated both by kernel subsystems above it (due to application requests), and by the device, resulting in concurrent driver code execution on both uniprocessor systems and CMPs. For example, functions for packet transmission, interrupt handling, and statistics reporting, can run concurrently in a network card driver.

2.1.3 Reentrant Execution

The asynchronous nature of interrupts and thus interrupt handler invocations pose reentrancy complexities for the kernel in general and drivers in particular. When a device generates an interrupt, the thread currently executing on the interrupted processor is delegated to execute the interrupt handler, its execution context is suspended and resumed when the interrupt handler completes. Now, if the interrupted thread was a kernel thread executing its driver's code, then it would be possible for the thread to race with itself on shared data while executing the device's interrupt handler. These kind of data races are more subtle compared to the more ones involving different threads. Signal generation and handling pose similar concurrency problems for the application stack, for example non reentrant functions (e.g malloc) cannot be safely used in signal handlers. Consequently, reentrancy must be carefully handled by drivers for correctness.

2.2 Driver Monitoring using Log Based Lifeguards

Recent studies [24, 44], found drivers to be highly prone to a wide variety of faults including, low level issues like heap overflows, high level issues like concurrency faults, and kernel specific issues like violations of OS and device protocols. Fortunately, lifeguards have been used for detecting a similar variety of faults in unmodified application binaries, heap overflows and uninitialized data uses [34], security vulnerabilities [16, 35], data races [46, 53, 23], and interface violations in multilingual programs [30]. However, instrumentation based lifeguards are not suitable for monitoring kernel mode drivers because OS preemption prevents the guaranteed atomic execution of corresponding lifeguard and driver codes. Log based lifeguards are therefore worthwhile to consider for detecting driver faults.

As illustrated in Figure 2, our proposal is to use log based lifeguards to detect faults in driver execution. As shown in the diagram, our design incorporates mechanisms for logging driver execution for lifeguard analysis, and for protecting the lifeguard from driver faults. The proposed driver lifeguards will run in user space rather than kernel space for the following reasons. First, to avoid further increasing kernel complexity by adding non trivial amounts of lifeguard code. Next, the kernel execution environment is too constrained for sophisticated lifeguards use large amounts of complex data structures e.g locksets. Furthermore, it provides some flexibility in how to protect the lifeguard from being

corrupted by driver faults that it will later detect, for example the lifeguard could run in a separate virtual machine (VM), our current strategy as shown in Figure 2. Finally, it allows control on the amount of driver (or kernel) data that is exposed to the lifeguard towards addressing the confidentiality concerns of lifeguard users. Thus similar to application lifeguards, driver lifeguards can only access execution state that is captured in the log.

Although logging and containing faults in the monitored execution has been studied for application monitoring [36, 9], the proposed solutions cannot be used for driver monitoring because driver execution is quite different from application execution. Consequently, in addition to new lifeguard analyses for detecting driver faults, logging and fault containment issues will also addressed in this thesis, and are further discussed below. Lifeguard efficiency is another critical consideration, since lifeguards that are significantly slower than the monitored execution, or consume too many resources cannot be used in production systems. However, lifeguard efficiency can not be treated in isolation, since it is affected by other aspects of log based fault isolation, i.e logging, analysis, and fault containment, rather it will considered while addressing those issues.

The main contribution of this thesis will be efficient lifeguards for detecting a wide variety of driver faults, including memory faults, concurrency faults, and violations of driver interface with the kernel and device. The main idea for achieving this goal is to incorporate relevant knowledge about the execution semantics of drivers into our designs. Since the kernel interface to drivers differs across device-classes, device-class lifeguards seem to be the best approach for detecting such interface violations. In other words, different lifeguards will be used for find kernel interface violations in block and network drivers. Similarly, lifeguards for detecting interface violations with the device are likely to be device specific. Consequently, driver lifeguards will be developed for different device-classes and devices as appropriate.

2.3 Logging Driver Execution For Lifeguard Analysis

Log based lifeguards detect faults by analyzing instruction grained execution traces, and therefore rely on an underlying software [13, 36] or hardware [9] logging mechanism to obtain such traces. As shown in Figure 2, the hardware logging proposal of Log Based Architectures (LBA) [9], will be extended in this work for logging driver execution. To reduce logging bandwidth and storage overheads, and lifeguard overheads, only driver execution will be logged out of the kernel space portions of I/O stacks (Figure 1). A variety of software faults arise from dependent events either being incorrectly ordered, or not ordered at all, e.g data corruption from writing to freed memory, data races from unserialized accesses to shared data, deadlocks from acquiring locks in a random order. Therefore to precisely detect such faults in drivers, it is essential that the lifeguard can obtain a total ordering of concurrent execution of driver code on different CPUS in a CMP system, existing hardware proposals [40, 51] will be adapted to capture this ordering information in the traces. Addressing the unique challenges of logging driver execution is discussed in Section 4.3 as part of the completed preliminary research .

2.4 Containing Driver Faults Until Lifeguard Detection

The delayed nature of log-based correctness checking means that a non negligible amount of time (e.g thousands of cycles) could elapse between the occurrence of a fault in the execution, and the detection by a lifeguard. Thus a comprehensive strategy for safely and efficiently containing yet-to-be detected faults is required for log-based lifeguards, primarily to prevent corruption of the lifeguard and the rest of the system (outside the monitored process boundary), and also to avoid irrecoverable damages within the monitored process boundary execution (e.g corrupting other non faulting threads).

A common approach used in application monitoring is to contain faults within the process boundary of the monitored application, by executing the lifeguard as a separate process, blocking at system calls [9], and (or) speculatively executing system calls [36]. Since drivers execute in the kernel, well below the system call boundary, execution blocking at system calls is not a viable fault containment approach, although speculative execution could be useful. Moreover, the driver executes within the process boundary of the kernel, and those techniques can neither prevent driver faults from corrupting other parts of the kernel, or even the lifeguard process. Since the primary focus of this thesis is to detect driver faults, our current fault containment strategy is to protect the lifeguard by running it in a separate VM from the monitored driver, similar to Aftersight [13]. However, the OS and applications running in the driver VM remain vulnerable to corruption by driver faults.

3 Related Work

Protecting commodity OS kernels from driver faults has attracted significant research attention in recent times, with proposals for either eliminating faults during driver development or isolating the kernel from driver faults at runtime. Proposals for eliminating driver faults during the development cycle advocate the use of static analysis [3, 20, 19] and testing [29], and synthesis of driver code from formal specifications of device and OS interfaces [45]. Unfortunately, not all faults can be eliminated during development due to the size and complexity of real world drivers, necessitating runtime isolation of faults in production drivers. Existing techniques for isolating driver faults are either hardware based [48, 25, 6, 21], or software based [54, 22, 8]. The section describes how this thesis work relates to these prior works. Proposals to develop drivers in type-safe languages [5, 31, 27] are not discussed since they are not applicable to popular commodity Oses such Windows and Linux which are written in C and C++.

3.1 Eliminating Driver Faults During Development

Engler et al. [20] observed that most system software rules specify legal orderings of operations and the contexts in which operations are permitted. Example rules in the Linux kernel include, re-enabling interrupts after disabling them, not using floating point, and checking user pointers before use. They therefore proposed Meta-level compilation (MC), which enables developers to easily write compiler extensions for checking system-specific rule violations in their code. MC extensions found hundreds of rule violations in the Linux kernel code (including drivers), and other complex system software. RacerX [19] uses flow-sensitive and interprocedural analysis to detect data races and deadlocks in operating system kernels and exploits system features to reduce false positives. It relies on developer annotations to infer which locks protect shared data, which code contexts are multi-threaded, and the system-specific locking functions. RacerX found serious errors in Linux, FreeBSD and a large commercial operating system. Static Driver Verifier (SDV) [3] used the SLAM static analysis engine [4] to find violations of the Windows kernel API usage rules in driver sources. SLAM analysis is based on model checking and symbolic execution, and uses iterative refinement to reduce false positives. SDV is distributed as an automatic tool which includes models of the OS and the execution environment of drivers, and kernel API usage rules to be checked. Our approach of using dynamic analysis to detect production driver faults is complementary to static checking techniques, since the size and complexity of real world drivers makes it difficult for all faults to be detected before release. Moreover static analysis can be used to reduce the overheads of dynamic analysis, by allowing runtime checks to be safely skipped for operations whose correctness can be statically checked.

Device Driver Testing (DDT) [29] is a system for testing device driver binaries that uses selective symbolic execution [11] and symbolic hardware to achieve good coverage of driver execution paths. Selective symbolic execution enables symbolic execution of the driver in its native unmodified OS environment, while the rest of the software stack above the driver run concretely. DDT found 14 serious new bugs in 6 device drivers that had already passed the Microsoft certification process. Similar to our approach of using lifeguards, DDT works on driver binaries and can therefore be employed at customer sites to test drivers before they are loaded. Moreover, DDT uses lifeguards for detecting driver faults during testing. However, DDT does not guarantee that all the execution paths of the driver will be explored during the test runs, therefore lifeguard checking is still needed for the production runs.

Termite [45] automatically synthesizes driver code from formal specifications of device and OS interfaces, making drivers free of interface violation faults by construction. Termite significantly reduces the burden of writing and porting drivers and was used to synthesize two non trivial Linux drivers. Similar to Termite's reliance on the availability of interface specifications, the proposed driver lifeguards will rely on interface knowledge to precisely detect violations in driver execution. However, Termite cannot synthesize multi-threaded drivers, and can therefore not be used for high performance network and graphics drivers. On the other hand driver lifeguards will work on multi-threaded drivers. Moreover, synthesized drivers still rely on handwritten memory management C code, and thus require lifeguard analysis to isolate memory errors at runtime.

3.2 Hardware Assisted Driver Fault Isolation

Nooks [48] used page table permissions to protect the kernel from a defective driver. Direct corruption of kernel data structures by driver code is prevented by restricting direct memory by the driver to regions of the kernel address space, and using introspection to validate modifications of kernel data structures. Nooks also tracks kernel resource management by the driver to enable recovery on driver failure. Nooks requires significant redesign of the OS kernel. Microdrivers [25] isolate the kernel from driver faults by moving most driver execution into user space, while leaving the performance critical portions in kernel space. The kernel is therefore immune to failures in the user space portion

of the driver. Microdrivers require significant redesign of OS and driver architecture. SUD [6] runs drivers in user space for fault isolation. However, unlike Microdrivers, SUD runs unmodified Linux drivers in user space using kernel environment emulation [18], and by mapping device IO registers into the driver's address space. SUD introduced two new Linux kernel modules that: (1) interfaces between the kernel and user space driver, and (2) allows user space drivers to safely manage hardware devices. Additionally, SUD relies on IOMMU [2, 1], PCI express bridges [38], and messaged-signaled interrupts [39] to prevent the device from being used by a malicious driver to corrupt the system.

In contrast to the lifeguard approach of detecting driver faults during execution, hardware based techniques that run the driver in a separate protection domain from the kernel, only insulate the kernel from direct effects of driver faults, such as memory corruption. However, defective drivers can still compromise the system in other ways, such as by corrupting data that it can legitimately access (e.g a network driver randomly flipping bits in a network packet). Therefore these approaches can be complemented with lifeguards to prevent such damages to the system. Moreover, the performance critical parts of Microdrivers, and non preemptible functions of SUD drivers remain in the kernel, and their faults could be detected using lifeguards. While SUD can protect the kernel even from malicious drivers, the proposed lifeguards do not provide such guarantees since driver faults are assumed to innocent mistakes.

DataCollider [21] detects data races in kernel code (including drivers) using hardware breakpoints of modern processors. To detect a race, DataCollider delays a thread when accessing shared data and puts a breakpoint on the shared data. If another access to shared data happens while the thread is delayed and at least one of the accesses is a write, it means the accesses are not synchronized and are racing. By catching races when they occur, DataCollider greatly simplifies bug fixing. In contrast to lifeguards, DataCollider is oblivious to the synchronization protocols used in the program, this is important for kernel code (including drivers) which use a variety of synchronization protocols e.g interrupts, spinlocks e.t.c. However, since it is impractical to delay all accesses to shared data in driver execution, DataCollider will miss real races, unlike lifeguards that detect all races.

3.3 Software-Based Driver Fault Isolation

Our proposed lifeguards fall into the class of techniques that use software to isolate driver faults. Existing techniques in this class are briefly compared to the lifeguard approach below.

SafeDrive [54] and BGI [8] are both compile time approaches for instrumenting drivers with checks for detecting faults are runtime. SafeDrive prevents corruption of kernel and driver data by common memory safety faults like buffer overflows, by using developer annotations of pointer bounds to insert the appropriate runtime checks. BGI provides stronger fault isolation guarantees than SafeDrive, including control flow integrity and temporal memory access control, without requiring source annotations. BGI maintains access rights for each byte of virtual memory, which is updated by the instrumentation code and an interposition library that mediates communication between kernel and driver. XFI [22] prevents a defective driver binary from writing outside its allocated memory to corrupt kernel data. Since type information is not available during binary instrumentation, XFI cannot detect corruption of driver data e.g via buffer overflows, rather it's runtime checks: (1) ensure that faulty writes are restricted to driver memory, and (2) enforce control flow integrity to prevent the write checks from being skipped by a malicious driver.

SafeDrive and BGI cannot be used for driver binaries since they are compile time techniques, unlike our lifeguard approach (and XFI), however the lack of type information in binaries means compile time approaches can more precisely detect type faults. SafeDrive, BGI, and XFI are instrumentation techniques and have the weakness that atomic execution of driver code and the corresponding guard code is not guaranteed. Although that is not an issue for our log based lifeguard approach, our approach requires error containment due to delayed execution of checking code, and this is not an issue for instrumentation approaches. Finally, these prior approaches (bar BGI perhaps) seem to be appropriate for detecting low level issues like memory safety, and not high level issues like concurrency or OS/device protocol violations. Lifeguards on the other hand have been successfully applied for both low level and high level correctness issues in the user space.

4 Preliminary Work

We have completed some initial studies as sanity check for this thesis, and to develop useful frameworks and garner insights for designing lifeguards for detecting device driver faults. These preliminary studies focused on (i) logging driver execution, and (ii) evaluating whether application lifeguards, with minor modifications, would be suitable for analyzing driver execution. Linux device drivers are the focus of our research, and our preliminary work was done using a 32 bit Fedora Core 5 Linux (2.6.17) kernel. We obtain two key results from our preliminary studies. First, we designed a technique for identifying and extracting instruction grained trace of driver execution within kernel mode

execution. Next, we learned that application lifeguards cannot simply be ported for analyzing drivers, for example, the Eraser data race lifeguard [46] that we ported for driver analysis, reported many false positives on two network drivers. We observed that these false warnings were due to the fact that unlike applications, drivers execute in a more constrained environment, under invariants enforced by the kernel and the device, thus more sophisticated lifeguard designs are required for drivers. This research work is being conducted in the context of a Log Based Architectures (LBA) system [9] which is simulated using Wind River Simics simulation system [50]. In the rest of this section, we provide background on LBA and lifeguards before describing our preliminary studies.

4.1 Log Based Architectures Background

Log Based Architectures (LBA) [9] augments CMP systems with a hardware log for efficiently capturing instruction grained execution trace of an application thread running on one processor, and delivering the trace to a lifeguard thread running on another processor. By enabling an application thread and its monitoring lifeguard thread to run concurrently on separate cores, LBA instantly improves lifeguard performance significantly, compared to DBI, where application and lifeguard execution compete for resources of a single processor. Additional hardware techniques were proposed for optimizing common lifeguard functionalities, e.g metadata management, which accelerated lifeguards like AddrCheck [34] enough for online monitoring. However, more sophisticated instruction grained lifeguards like Eraser [46], TaintCheck [35], and MemCheck [34] remain much slower than applications on LBA, because these lifeguards execute many instructions to analyze each application instruction, application threads are therefore stalled when their log is full until the lifeguard creates more space in the log. LBA also incorporates techniques for containing application faults within the application's process boundary until detection by the delayed lifeguard checking: (i) the lifeguard is run as separate process and is therefore isolated from application faults, (ii) application threads are stalled at system call boundaries to prevent OS corruption. Further LBA extensions have been proposed for monitoring parallel applications [26, 51], and accelerating lifeguards [43, 42].

4.2 Background on Lifeguards

Lifeguards are dynamic correctness checking tools that monitor execution of unmodified software binaries to find errors [46, 16, 35, 53, 34, 23, 30]. They are deployed by using software or hardware to instrument [7, 10, 34, 15] or log [13, 36, 9] the execution of the monitored program. Powerful lifeguards that perform instruction-by-instruction analysis can detect a wide range of program faults including low level issues such as memory [34] and security [16, 35] errors, as well as high level ones such as concurrency errors [46, 53, 23] and interface errors in multilingual programs [30]. Although different lifeguards are designed to check for different correctness issues, they do share common characteristics as described below.

C1. A lifeguard maintains a data structure that records state information ("metadata") about the monitored application's address space (e.g., which addresses have been allocated or tainted). There is a 1-1 mapping between application data and lifeguard metadata at some granularity (e.g., each application byte maps 1-1 to a metadata bit).

C2. A lifeguard is interested in observing many, but possibly not all, application execution events. Some of the events are needed solely to maintain the metadata. For other, "interesting" events, the lifeguard checks the current metadata state, and reports an error if an anomaly has occurred.

C3. There is a mapping from monitored application execution events to specific lifeguard functionality ("handlers") based on the event type (load, store, etc.); these handlers are invoked in response to the sequence of execution events.

4.3 Logging Driver Execution

We extended LBA for capturing the instruction grained execution trace of a monitored driver in an efficient manner with regards to logging bandwidth and storage overheads. Monitoring driver execution posed new challenges to LBA, since it was originally designed for monitoring application execution which is quite different from driver execution. The issues tackled in extending LBA for drivers are: (i) how to identify driver execution within kernel mode execution (4.3.1), (ii) how to associate logs with execution of driver code by kernel threads (4.3), and (iii) how to identify the context (process/interrupt) of driver code execution (4.3.3), this is important for identifying concurrency issues. However, there are other issues which were addressed for monitoring drivers using existing LBA solutions

for application monitoring. First, drivers and applications manage system resources like memory using lower level functions that are implemented by the kernel for driver and by system libraries for applications. Application lifeguards identify such functions in the execution trace to precisely detect low level faults like double frees, and LBA simplifies their identification by instrumenting them. Similarly resource management functions like kmalloc are instrumented in driver execution trace. Second, driver code can be concurrently executed on different processors of a CMP system, and detecting concurrency faults in the driver's execution trace requires a total ordering of the memory accesses that occurred on different processors. We adopt the ParaLog extension [51] of LBA to order the actions of kernel threads that are concurrently executing the monitored driver code.

4.3.1 How To Identify Driver Execution

The ideal situation is that only the execution trace of the monitored driver is captured in the log, otherwise disproportionally high overheads will be incurred in logging bandwidth and storage, since most drivers individually account for only a small portion of kernel execution. However, more importantly the lifeguard will incur avoidable significant slowdowns to filter the trace. This is also the situation in application monitoring where only the execution of the monitored application is logged by LBA. Therefore logging all of kernel mode execution [52] is not acceptable, rather entry to driver code (from outside the driver) and the corresponding exits must be detected.

Unlike application thread monitoring, where LBA relies on modest OS scheduler support to enable (disable) logging when the thread is scheduled (descheduled), our solution instruments exported driver functions similar to [49, 28, 37], to detect entry (exit) and enable (disable) logging. We learn the entry points of a dynamically loaded driver by instrumenting the kernel module loading function (sys_init_module), and the kernel functions for registering device driver functions (e.g register_netdev () for network drivers). Drivers call other kernel functions for managing resources like memory or registering entry points, those functions could in turn call other functions of the driver causing nested execution of driver entry points. Our solution carefully tracks such nestings to ensure that logging is disabled only exit from the initial driver entry point. Tracking nested driver entry points is greatly simplified by the absence of non local jumps (setjmp/longjmp) in the Linux kernel code. With this approach, executions of other kernel functions are logged only when called from driver code, which is the desired situation.

4.3.2 How To Associate Logs with Driver Execution

The concurrent execution of a monitored's driver code on different processors by kernel threads creates the opportunity for efficient logging by capturing driver traces in parallel into distinct logs. This however raises the question of how to maintain the logs, and the obvious alternatives are to maintain them per processor or per kernel thread. For application monitoring, maintaining logs per thread seems the more reasonable option, because the lifetime of threads are often closely related to that of the application (main thread), thus log management (creation/deletion) overheads correspond roughly to thread management overheads in the application. On the other hand, a driver could be executed by an unbounded number of kernel threads, basically all kernel threads in system while the driver is loaded. Moreover, these threads or their lifetimes are not related to the driver, and do not retain driver state when not executing in the driver. Thus in our LBA extensions for driver monitoring, logs are associated with processors rather than with kernel threads. In other words, for each processor, a unique log is used for capturing driver execution by different threads on that processor. Context switch events are recorded in the log to enable lifeguards identify the different threads.

4.3.3 How To Identify Context Of Driver Execution

Kernel threads execute either in process context when computing on behalf of a (user/kernel) process, or in interrupt context when performing interrupt handling related tasks. Interrupt handling in the Linux kernel is divided into two phases: (i) a fast top half that is invoked when a device generates an interrupt (conventional interrupt handler routine), and (ii) a bottom half that runs at a more convenient time to perform computationally intensive task such as copying received packets from a network device. While a kernel thread is executing driver code in process context, the driver's device could generate an interrupt causing the thread to temporarily execute the driver's interrupt handler in interrupt context. The kernel thread is now concurrently executing driver code in two independent execution contexts, and could possible race with itself on shared data accesses. It is therefore essential that concurrency lifeguards be aware of the execution contexts of kernel threads while executing driver code. Rather than including execution context information in the log, we observed that the interrupt handling routines (top and bottom halves) were the only driver entry points executed in interrupt context. Therefore we expect lifeguards that are interested in execution context information can easily retrieve it from the current trace format.

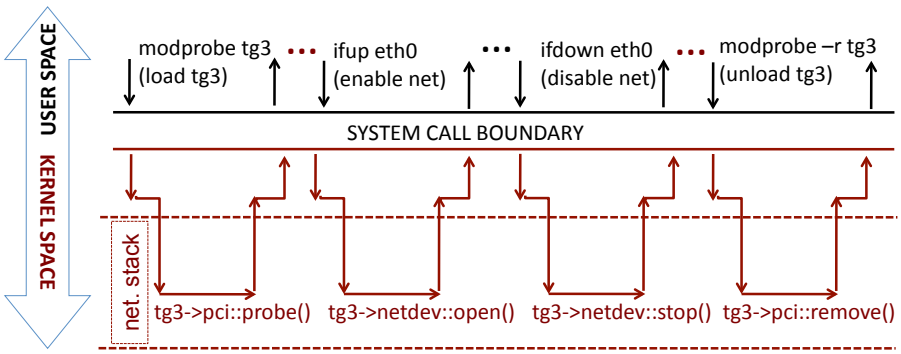

Figure 3. Flow of execution from user space to driver functions, at the base of the networking stack, via system calls.

4.4 Analyzing Device Drivers using Application Lifeguards

To gain insights for designing device driver lifeguards, we evaluated whether application lifeguards with minor modifications could be used to detect driver faults precisely, that is without missing real faults (no false negatives), and without false alarms (no false positives). For our initial driver lifeguard, we studied data race detection in drivers because although concurrency errors are common in drivers [44], they have received little research attention except for the recent DataCollider work [21]. We focused on PCI network drivers (tg3 and tulip) since they are highly multi-threaded. The LBA extensions for driver monitoring, described earlier, were used to collect execution traces of the network drivers for analysis by KernelEraser, which is an adaptation of the Eraser data race lifeguard [46] for kernel mode execution. The analyzed traces contained driver execution during network file transfer using scp, as well as during driver (un)loading and network interface enabling(disabling) shown in Figure 3. Unfortunately KernelEraser reported many false data races while analyzing the driver execution. We observed that the shared data accesses that were incorrectly flagged as racing, were infact ordered by synchronization events that occurred outside the driver, in the higher layers of the device protocol stack and in the device, and thus not observed by KernelEraser. We now describe in more details, our experience of analyzing drivers with KernelEraser, starting with a background on Eraser (4.4.1), then a description of the Eraser modifications (4.4.2), and finally an analysis of the false data races reported on the drivers (4.4.3).

4.4.1 Background on Eraser

Eraser [46] detects data races in multi-threaded application binaries at runtime, by verifying that shared data access is protected by a consistent set of locks (lockset). Eraser maintains a lockset per shared data that it updates on every access and which contains the intersection of the locks held by threads while accessing the data. In other words, every lock in a data's lockset was held by every thread that accessed the data in the past, and therefore data races are reported whenever the lockset becomes empty. Eraser's Lockset algorithm can even detect races that are not manifested in the current thread interleaving, this is a major benefit since data races are notoriously difficult to reproduce due to non deterministic thread scheduling. However, Eraser incorrectly reports races on accesses that are correctly synchronized using non lock mechanisms, such as the "happens-before" semantics of events like fork/join. Consequently, refinements to the Lockset algorithm, including happens-before tracking, were proposed [53, 47] to eliminate such false positives.

4.4.2 Modifying Eraser for Kernel Mode Execution

Unlike applications which manage system resources like memory through system libraries, drivers (and other kernel code) execute below system libraries, and therefore rely on kernel's resource management code such as kmalloc for obtaining memory. Thus our modifications to enable Eraser analyze drivers focused on supporting memory and synchronization management functions of the kernel. Supporting kernel memory management functions such as kmalloc and kfree was quite trivial since they behave in a similar manner to their user space parallels, malloc and free. However, the Linux kernel provides a richer set of synchronization primitives compared to user space, ranging from the familiar lock based primitives like spinlocks, mutexes, and semaphores, to more specialized ones like atomics, completion variables, and interrupts. We added support for the kernel synchronization methods that were used by the network drivers that we studied, in particular we represent interrupts with virtual locks similar to the extensions proposed by the Eraser

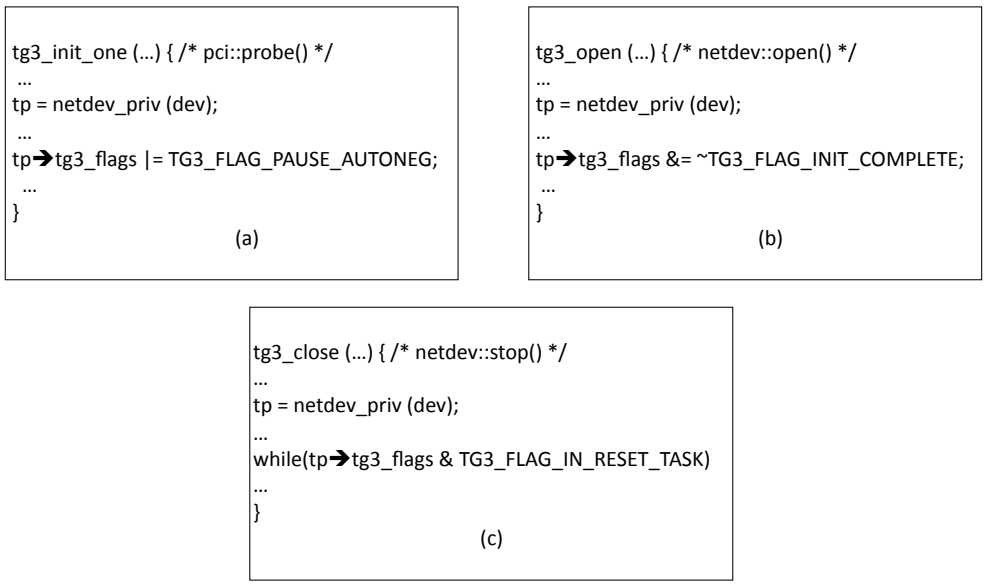

Figure 4. Snippets of tg3 driver functions that access shared driver data (tg3_flags) without a common lock and are executed by different threads. KernelEraser reports netdev::open() as racing with pci::probe() and with netdev::close().

authors for supporting interrupts in the SPIN OS. Specifically, local interrupt lines of a processor are represented by virtual locks which are acquired (released) when the interrupt lines are disabled (enabled) e.g via spin_lock_irq* calls. With these modifications, our KernelEraser was able to analyze execution traces of network drivers, albeit with false positives as discussed next.

4.4.3 Analysis of False Data Race Reports of Network Drivers

We examined the races reported by KernelEraser and found most of them to be false positives, while the remaining were benign races in the statistics processing of tg3. The false alarms involved data accesses which were serialized by events that occurred outside the driver, and thus not observed by KernelEraser since only driver execution was logged into the traces. We identified the "unlogged" serializing events to be of two types: (i) synchronization operations in the upper layers of the networking stack, and ii the semantics of a PCI network device. Below, we examine both types of events using candidate KernelEraser false alarms. But, first we provide some background on PCI network device drivers to help in understanding why the discussed data races are indeed false alarms.

The functions that a PCI network device driver is required to export can be roughly grouped into three categories based on the device operations they are related to. The first class of functions are for managing the core networking functionalities of the device, and includes open() and stop() for respectively activating and deactivating the device for operation. The second class of functions are for managing the device on the PCI bus to which it is connected, and includes probe() and remove() for respectively connecting to and disconnecting from the bus. The final class are the top and bottom halves for handling device interrupts. To enable easy identification of driver functions in the rest of this document, driver functions are prefixed appropriately. For example, open() will be referred to as netdev::open(), while probe() will referred to as pci::probe().

The first false alarm reported by KernelEraser that we discuss involved the write and read of tg3_flags by netdev::open() and netdev:close(), shown in Figure 4(a), and Figure 4(c) respectively. KernelEraser flagged these data accesses as races because they were performed by different kernel threads without holding a common lock. However as suggested in Figure 3, it is reasonable to expect that both functions are serialized in some way. This turns out to be the case, as invocations of netdev::open() and netdev::close() are serialized in the higher layers of the networking stack using a common lock.

Another false alarm involved the same netdev::open() write and the pci::probe() write shown in Figure 4(b). However, we observed that executions of pci::probe() and netdev::open() are serialized by the states of the device, since the device can be viewed as finite state machine (FSM) where states correspond to the device (registers) configurations, and state transitions are caused by manipulation of device registers by driver code. For example, during operation, a PCI network device transitions through different states (or configurations), including being inactive be-

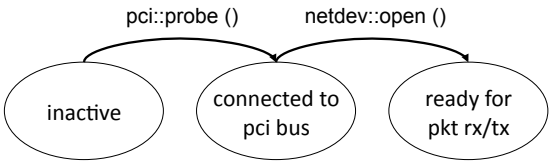

Figure 5. A subset of the states of a network device and the driver functions that cause state transitions. Since pci::probe() connects the device to the pci bus where netdev::open() can execute, both functions are serialized.

fore initialization by the driver, suspended to conserve power when idle, not generating interrupts on packet reception to support driver polling e.t.c. Moreover, the device states control which driver functions execute, for example, the interrupt handler will not execute while interrupts are disabled on the device, and the packet transmission function (netdev::hard_start_xmit()) will not execute while the device is inactive or suspended. Therefore a pair of driver functions without a common device state where both can execute are infact serialized and cannot race. Figure 5, which depicts a snippet of a PCI network device states, identifies pci::probe() and netdev::open() to be such a pair of serialized driver functions, and thus races reported by KernelEraser that involve them are false alarms. This is because, pci::probe() connects the device to the pci bus, while netdev::open() can only execute after the device is connected to the bus, in other words netdev::open() executes after pci::probe() completes as suggested in Figure 3.

Our above discussions suggest that the false alarms of KernelEraser can be eliminated by making it aware of relevant synchronization operations that occur outside the driver, and forms a part of the proposed research of this thesis. Proposed research also includes extending KernelEraser to monitor driver classes e.g. block, usb and graphics.

5 Proposed Research

We now describe our proposed research work in details below. As stated earlier, the primary goal of this thesis is to usher in a new class of powerful log based lifeguards that can precisely detect driver faults, and are efficient enough to use on production systems. We had earlier identified the following as fundamental requirements to achieving this goal: (i) an efficient logging strategy that captures only the driver's execution trace, since redundantly logged information wastes resources and creates additional filtering work for the lifeguard, (ii) dynamic binary analyzes to efficiently detect a wide range of driver faults, without reporting false alarms or missing actual faults, on production systems, and (iii) a strategy for efficiently containing faults within the driver, to avoid corrupting the lifeguard or the rest of the system. Efficient driver logging was addressed in our preliminary studies (Section 4.3), and the remaining research work are divided into three major tasks.

In the first two tasks, we focus on detecting concurrency faults and interface violations in drivers, since they are not as well studied as low level memory faults [54, 22, 8], and respectively account for 19% and 58 of 498 Linux driver faults (remaining 23% were due to well studied generic C errors) in a recent study [44]. The last task focuses on improving lifeguard performance to enable online monitoring of production drivers. We discuss each of these tasks in more details in the rest of this section.

5.1 Detecting Concurrency Faults in Drivers

As described earlier (Sections 2.1.2 & 2.1.3), concurrency is a fundamental feature of driver execution on modern OSes, even on uniprocessor systems, and the degree of concurrency is bound to grow with the increasing popularity of CMP systems. Since writing error free parallel program is very difficult, it is therefore not surprising that drivers suffer from various kinds of concurrency faults including data races, deadlocks, and atomicity violations, and the situation will only get worse in the future. For that reason, we decided to first study concurrency faults, beginning with data races in our preliminary studies (Section 4.4). We describe our current ideas for detecting data races below.

5.1.1 Detecting Data Races in Drivers

From our preliminary study of data races in drivers using KernelEraser (Eraser lifeguard adapted for kernel execution) in Section 4.4, we learned that in contrast to application monitoring where all the relevant serialization operations are captured in the log, accesses to shared data in drivers could be serialized by events outside the driver, leading to false alarms since they are not logged and thus not observed by KernelEraser. Moreover, these external serialization events serialize groups of driver interface functions, and occur either in the upper layers of the device protocol stack or due to the states of the device. For example invocations of the netdev::open(), netdev::close(), and netdev::do_ioctl()

functions of a network driver are serialized in the higher layers of the networking stack using a common lock(*rtnl_lock*), preventing each function from racing with others (including itself) in the group.

Our strategy for eliminating these false alarms is to make KernelEraser aware of driver function pairs that are externally serialized, so that it does not flag accesses to shared data during their execution by kernel threads as races. To do this, we will maintain extra metadata for shared data in the driver for tracking the other ways of serializing driver code besides driver locks, similar to hybrid data race detection proposals [53, 47]. However, unlike previous hybrid data race detection proposals which track happens-before relationships of shared data accesses, our approach is to track mutual exclusion relationships of driver interface functions that access shared data. Specifically, the additional pieces of metadata are: (i) the interface function that last accessed the data, and (ii) the state of device during the last data access. With this information, KernelEraser can determine that an access to shared data by an interface function is serialized with the previous access, either because (i) the function is serialized with the interface function that perform the previous access e.g netdev::open() and netdev::close() in network drivers, or (ii) it could not have executed in the state the device was in during the previous access e.g the interrupt handler for the device will not execute while the device interrupts are disabled. For accesses that are not serialized externally, lockset is used in the normal way for checking that they are correctly serialized using driver locks.

While driver functions that are serialized in the upper layers of the device protocol stack are documented, tracking the device state during driver execution poses a challenge for our strategy. Our current idea for tracking the state of the device is motivated by how upper layers of the kernel track device state transitions, since unlike drivers, they lack the device specific knowledge for parsing the device registers. Instead relevant device state information are shadowed in a shared data structure (later referred to as **dev_state_shadow** for convenience) that is maintained by the driver and upper layers of the protocol stack e.g netdev::state for network drivers. In other words, driver interface functions are invoked by the upper layers when value of dev_state_shadow indicates that is valid to do so. We plan to capture dev_state_shadow changes in the log to enable the lifeguard track device state transitions. However, our proposed strategy to eliminate KernelEraser false positives, assumes correct behavior of the external components that serialize driver execution, and that dev_state_shadow accurately reflects the state of the device. For example, we assume netdev::open() is only executed by a kernel thread that is holding *rtnl_lock*.

In addition to avoiding false positives, KernelEraser must not miss real races, especially when a particular kernel thread races with itself in interrupt and process execution contexts. In other words, a kernel thread must synchronize the accesses it makes to shared data in different execution contexts. Since we decided not include execution context information of kernel threads in the log, KernelEraser has to infer this information, which does not seem to be a difficult task since the top and bottom halves of the interrupt handler are the only driver interface functions that are executed in interrupt context.

5.2 Detecting Interface violations in Drivers

The driver plays the essential role of coordinating the communication between upper layers of an I/O stack (including applications), and the I/O device. For it to perform this function properly, the driver must interract with both the software layers above it, and the device below it. This requirement increases driver complexity because two different interfaces must be correctly implemented for (i) synchronous communication with software through main memory and CPU registers, and (ii) asynchronous communication with the device through device registers, I/O ports, and interrupts. Due to these additional complexities, interface violations are quite common in drivers [44].

5.2.1 Detecting OS Protocol violations in Drivers

I/O requests and the success/fail status of such requests are communicated between the upper layers of the I/O stack and the device, through the driver. Similarly, I/O data (e.g. network packets to transmit (or recieve)) are channelled through the driver. These communications takes place through different channels including dev_var_shadow updates, return status and arguments of driver functions, and arguments of higher layer functions called by the driver. Below, we examine a concrete example of driver/kernel communication protocol in the networking stack, and failures that could arise from violations by a faulty network driver.

netdev::hard_start_xmit() is called in the networking stack with a buffer holding the packet data to be transmitted as argument, and it returns either a success or failure status to the caller. A success return status means the driver has taken responsibility for: (i) ensuring the packet is eventually transmitted by the device, and (ii) freeing the buffer, and so the caller can return a success status up the stack to the application (possibly via send() system call). Conversely, a failure return status means that the driver can take neither of those responsibilities (at the moment), in which case the caller

can retry later. There are a number of ways through which defects in the driver could cause netdev::hard_start_xmit() to violate its protocol with and corrupt the networking stack. While the most obvious violations are incorrect return status, more subtle ones include failing to free the buffer, leading to kernel memory leaks.

We plan to study analyzes for detecting driver interface violations in device protocol stacks for different device classes including network devices. These analyzes need to be aware of the different interface functions of a driver, as well as their expected behavior as documented in the kernel. Since the protocol stacks of different device classes vary considerably, e.g network vs block device classes, device class specific lifeguards seem a reasonable approach.

5.2.2 Detecting Device Protocol violations in Drivers

The driver interacts directly with the device to meet the needs of the rest of the system, primarily satisfying I/O requests. Examples of driver interaction with device include device (de)activation, enabling (disabling) device interrupts generation, checking the status of the device, and moving data in and out of the device. To interact correctly with the device, the driver should read and write device registers or I/O ports in a manner prescribed in the device documentation provided by the device manufacturer. Device registers are mapped into the I/O memory space, and unlike main memory accesses, I/O memory accesses have side effects, and therefore not idempotent. Despite these differences between main memory and I/O memory accesses, they are performed using the same instructions (at least in x86), making it difficult to distinguish them in the binary. Moreover, interaction with the device could be asynchronous, for example it is quite common that values written to a device register can only be safely read after some specified delay, thus delays (udelay ()) are often used in drivers.

Unfortunately, violations of the device interface could lead to misconfiguring or misinterpreting device state, and result in a wide range of system failures, such as putting the device in an unusable state, data corruption (loss) during transfers to (from) the device, and failure to correctly handle device interrupts. We therefore plan to investigate analyzes for detecting device protocol violations in drivers. This class of analyzes are likely to be parameterized using information that is specific to a given device or at best to a family of devices with similar designs from the same maker. This is because different devices, even of the same device class are unlikely to have the same driver, required the same amount of delays for device register access, or be configured in the same way. A challenging part of this work will be incorporating the device model from the production sheets into lifeguard analyzes.

5.3 Accelerating Log Based Detection of Driver Faults

To be considered practical for detecting driver faults in production systems, our proposed log based lifeguards must execute efficiently and not significantly slowdown the system or consume too much resources. Since lifeguard performance is impacted by the three components of log based monitoring, i.e logging, analysis, and fault containment, it is worth considering each component for lifeguard optimization opportunities. One proposal from our preliminary studies is to avoid logging execution that is outside the monitored driver so the lifeguard can avoid the runtime cost of filtering them out of the log.

As discussed further below, remainder of this research will focus on using static analysis for lifeguard acceleration, dynamic optimizations of lifeguard analysis, and more efficient fault containment approaches than VM separation. However, since lifeguards do not significantly slowdown I/O bound applications [35], and most drivers are I/O bound, it may not be necessary to implement all the proposed optimization techniques before driver lifeguards become efficient enough for online monitoring.

5.3.1 Incorporating Static Analysis to Accelerate Detection of Driver Faults

The instruction-by-instruction analysis that sophisticated lifeguards perform on the monitored execution is another component of log based monitoring overheads. While static analysis cannot be used to eliminate all the faults in a program, it can however be used to reduce lifeguard work by performing some of the correctness checks before execution. XFI employed static analysis to reduce the number of instrumentation checks needed to detect control flow integrity violations and memory access errors in driver binaries [22] . Similarly, we expect that the driver lifeguards proposed in this research will benefit from static analysis, since tens to hundreds of instructions will likely be needed for the sophisticated analyzes on each driver instruction.

Our current thoughts are that interesting correctness properties of drivers can be statically verified because drivers have well defined structure (dictated by the kernel) and provide a narrow and standard set of services. In particular, drivers in the same device class export standard and documented interfaces to upper layers of device protocol stack. For example, some driver functions are executed exactly once during driver loading (module_init()) and unloading

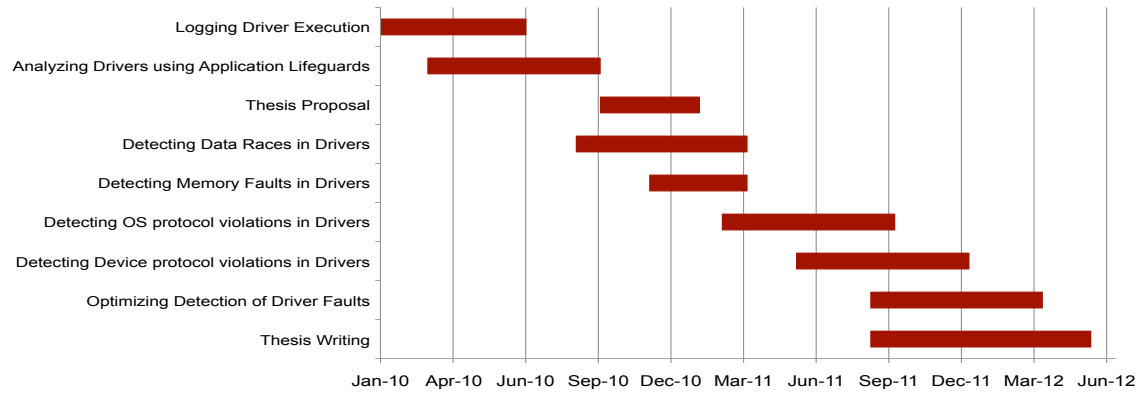

Figure 6. Proposed timeline to complete the thesis.

(module_cleanup()). Similarly, some data buffers used by drivers are available only during initialization, and are automatically released by kernel subsequently.

5.4 Dynamic Optimizations for Accelerating Detection of Driver Faults

Our decision to run the proposed driver lifeguards in user space means they can benefit from dynamic techniques for accelerating lifeguard analysis using hardware accelerators [17, 9], parallelism [36, 43], and compiler optimizations [41, 42]. However, those proposals were designed for monitoring execution of single threaded or multi threaded applications on a single core, and the requirements for monitoring multi-threaded execution on multiple cores [26, 51] violate their optimization strategies. In particular, those techniques improve lifeguard performance by identifying and skipping redundant correctness checks while analyzing paths of the monitored program. For example, the Addrcheck lifeguard [34] for finding unallocated memory access errors, needs to check only one of the many accesses to a particular location in a program path, if that location is not (de)allocated in the path. However, to apply this optimization for multi-threaded execution monitoring, one has to additionally guarantee that concurrently running threads are not (de)allocating that thread.

Our current ideas for applying dynamic optimizations to driver monitoring, are motivated by the hardware based mechanisms (*conflict alerts* and *delayed advertizing*) proposed by Paralog [51] for tracking inter-thread data dependencies, to enable hardware acceleration of multi-threaded application monitoring.

5.4.1 Exploring More Efficient Strategies for Containing Driver Faults

Our current plan is to use VM separation to isolate the lifeguard from driver faults, by running the lifeguard and monitored driver in separate VMs. While this approach adequately protects the lifeguard, it has the disadvantages of requiring VM resources, and log transportation across VM boundaries. Additionally, software in the driver's VM remain vulnerable to corruption by driver faults.

Consequently, we plan to explore fault containment techniques that are more efficient than VM separation, without sacrificing lifeguard safety. In other words, the lifeguard will run in the same VM as the monitored driver. This approach has the added requirement (or benefit) that the rest of the system, outside the driver, must be adequately protected from driver faults. Our current idea is to consider if the "block at critical points" approach used for application faults can be used for drivers. That is we are trying to identify a (hopefully) small number of events related to driver execution, where correctness checking has to catch up with execution to avoid irrecoverable system corruption. Our initial candidate events include accesses to the I/O ports and memory space by the driver, handling faults (e.g page faults) that are generated by the driver, and exposure of state updates by the driver to the lifeguard and other parts of the kernel and the device protocol stack. Identifying the required set of blocking events in driver execution and designing an efficient fault containment system around them will be the focus of the proposed research.

6 Timeline

The proposed timeline for completing this thesis is shown in Figure 6. The first main task of logging driver execution, and online analysis of the log by a lifeguard has been completed. With this infrastructure, I was able

to demonstrate that application lifeguards were not suitable for monitoring drivers, even with modest modifications, rather a new lifeguard design approach which leverages knowledge of driver execution is required. Now, I m working on the next major phase, i.e design lifeguards for detecting a variety of device driver defects. Specifically, I m working on detecting data races and memory faults in drivers. After completing the lifeguard development phase, I will explore ways of improving the performance of those lifeguards so that they are suitable for use in in production systems. Finally, I will begin writing the thesis report.

References

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

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

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

[4] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging System Software via Static Analysis. In *POPL*, 2002.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *SOSP*, 1995.

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

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

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

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

[10] Chi-Keung Luk and Robert Cohn and Robert Muth and Harish Patil and Artur Klauser and Geoff Lowney and Steven Wallace and Vijay Janapa Reddi and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[11] Vitaly Chipunov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective Symbolic Execution. In *HotDep*, 2009.

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

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

[14] JaeWoong Chung, Michael Dalton, Hari Kannan, and Christos Kozyrakis. Thread-Safe Dynamic Binary Translation using Transactional Memory. In *HPCA*, 2008.

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

[16] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, 2005.

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

[18] Jeff Dike. http://www.user-mode-linux.sourceforge.net.

[19] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *SOSP*, 2003.

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

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

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

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

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

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

[26] Michelle L. Goodstein, Evangelos Vlachos, Michael A. Kozuch, Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Butterfly analysis: Adapting Dataflow Analysis to Dynamic Parallel Monitoring. In *ASPLOS*, 2010.

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

[28] R. Krishnakumar. Kernel Korner: Kprobes-a Kernel Debugger. *Linux J.*, 2005(133), 2005.

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

[30] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *PLDI*, 2010.

[31] James G. Mitchell. JavaOS: Back to the future. In *OSDI*, 1996.

[32] Brendan Murphy. Automating Software Failure Reporting. *Queue*, 2004.

[33] Vijay Nagarajan and Rajiv Gupta. Architectural support for Shadow Memory in Multiprocessors. In *VEE*, 2009.

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

[35] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

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

[37] Marek Olszewski, Keir Mierle, Adam Czajkowski, and Angela Demke Brown. JIT Instrumentation: A Novel Approach to Dynamically Instrument Operating Systems. In *EuroSys*, 2007.

[38] PCI-SIG. Pci local bus specification, revision 3.0 edition. 2004.

[39] PCI-SIG. Pci express 2.0 base specification revision 0.9 edition. 2006.

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

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

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

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

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

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

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

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

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

[49] Ariel Tamches and Barton P. Miller. Fine-grained Dynamic Instrumentation of Commodity Operating System Kernels. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, 1999.

[50] Virtutech Simics. http://www.virtutech.com/.

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

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

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

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