Making operating systems resilient to hardware

Asim Kadav
Preliminary Examination Report
Computer Sciences Department, University of Wisconsin-Madison

Abstract

Device drivers are the the operating system's interface to the outside world. Drivers have been a constant source of unreliability and security problems in modern operating systems due to (1) driver development process comprising of thousands of developers and (2) driver architecture which gives drivers privileged access to system resources to provide maximum functionality. Both these factors have led to significant reliability research on drivers. This research has primarily looked at the driver-kernel interface and has applied coarse grained isolation techniques over device drivers. In my thesis proposal, I seek to improve the state of modern device drivers against the backdrop of reliability through three different techniques. First, I improve the tolerance of modern drivers against unreliable hardware and make drivers more robust by automatically patching driver code. Second, I will study the current isolation and recovery techniques and wish to demonstrate the effectiveness of using a finer grained driver isolation and low overhead recovery solutions. And finally, I will study the functional part of driver code, its workings, form and its abstraction architecture and propose some guidelines on how drivers can improve by organizing themselves better.

1 Introduction

Reliability remains a paramount problem for operating systems. As computers are further embedded within our lives, we demand higher reliability because there are fewer opportunities to compensate for their failure. At the same time, computers are increasingly dependent on attached devices for the services they provide.

Consequently, driver research over the past decade has significantly focused on reliability problems using software/hardware fault isolation techniques, bug detection tools, using special programming languages or environments for surviving crashes in modern operating systems due to faulty or malicious device drivers [4, 8, 9, 10, 11, 12, 17, 20, 31, 34, 41, 47, 51, 52, 53, 58, 64, 67, 69, 70]. However, there are some issues which this body of work fails to address.

Almost all reliability solutions, target the driver kernel interface and do not address reliability problems due to hardware issues. The device and driver interact through a protocol specified by the hardware. Failure in adherence to this protocol, or in its implementation can lead to serious security and reliability issues since drivers operate in privileged domain in modern operating systems. While most research has focused on problems in device drivers, there has been little research on bugs arising due to faulty hardware. Some failures are caused by wear-out or electrical interference [36]. In addition, internal software failures can occur in devices that execute embedded firmware, sometimes up to millions of lines of code [72]. Hence, we need to address issues arising out of hardware bugs. Applying the same solutions as used for driver software bugs do not address reliability due to unreliable hardware. Hence, it is important to assess the nature of hardware unreliability issues that plague modern drivers and design stronger detection schemes to address these issues.

One also needs to review the isolation and recovery mechanisms proposed by state of the art driver research. Driver isolation systems have not been popular because they impose significant overheads and also require interposing driver-kernel architecture that gives rise to maintainability issues. Ever since Nooks [64] demonstrated that drivers can be isolated and recovered automatically, most research has proposed newer isolation techniques applied indiscriminately to the whole driver. An exception to this rule is microdrivers [23]. Microdrivers isolates all functions in user-space except performance critical routines like interrupt handlers, which also cannot be supported in user-space. Furthermore, there have been no recovery techniques proposed to recover drivers at finer granularity, with all of them unloading/reloading the driver and replaying previous state before crash. Hence, drivers and devices lack the ability of isolating and recovering at a finer granularity. Such a functionality, should impose lower overheads and can be leveraged by numerous bug detection tools which detect bugs in drivers but do little to fix them.

Additionally, most research has focused on developing techniques to detect and fix driver bugs but one needs to understand drivers beyond the bugs. This will help us better understand the cause of these bugs and un-

reliability problems due to driver design. While systems researchers are broadly aware of the functionality of drivers, little is known about the huge functional part of driver code beyond these bugs or about few drivers from basic driver classes. Hence, studying code from all driver classes can be a useful exercise and can help us develop new insights on how to improve driver code. One reason to understand all drivers is to review if the driver research solutions being developed are applicable to all classes of drivers. Second, existing reliability and bug finding tools treat bugs as problems and either fix them in place or tolerate them at runtime. However, some times bugs are not the problems but they represent symptoms [38] of broader issues. To better understand and detect design issues, one needs to analyze the code with a broader goal.

In this dissertation proposal, I address three issues relating to device driver reliability. First, I develop techniques to identify drivers susceptible to hardware failures and how to fix them. Next, I describe techniques to improve the current isolation and recovery mechanisms in drivers. We review the problem to driver security to describe the utility of my solution. Finally, we will study the driver source code of modern operating to understand this protocol better and propose specific solutions to improve the device-driver-kernel protocol.

Tolerating hardware device failures in software To address the above problem, we have developed Carburizer, a code-manipulation tool and associated runtime that improves system reliability in the presence of faulty devices. Carburizer analyzes driver source code to and locations where the driver incorrectly trusts the hardware to behave. Carburizer identified almost 1000 such bugs in Linux drivers with a false positive rate of less than 8 percent. With the aid of shadow drivers for recovery, Carburizer can automatically repair 840 of these bugs with no programmer involvement. To facilitate proactive management of device failures, Carburizer can also locate existing driver code that detects device failures and inserts missing failure-reporting code. Finally, the Carburizer runtime can detect and tolerate interrupt-related bugs, such as stuck or missing interrupts.

On demand isolation To address performance issues due to isolation, we develop fine grained isolation and recovery mechanisms. To demonstrate the usability of such a problem, we address the problem of driver security against malicious hardware and userspace calls. First, I demonstrate that isolation at a finer granularity can indeed be useful. Then I discuss techniques that enable us to provide fine grained isolation in a single kernel address space. Finally, I demonstrate recovery techniques that does not require reloading and unloading the driver. I also explain how the driver isolation techniques does not require interposition on the driver-kernel interface which can be a source of maintainability issues.

Understanding and improving modern driver code To understand and improve driver code, we wish to study the source code of Linux drivers to determine whether assumptions made by most driver research, such as that all drivers belong to a class, are indeed true. I also develop a set of static-analysis tools to slice driver code across various axes. Broadly, the study addresses three important questions: (i) what is the function of all this driver code,(ii) what is the form of the driver code and can we get rid of some of the code by providing better interfaces? and (iii) can the driver be abstracted in a better way given the modern trends of hardware.

2 Tolerating Hardware Device Failures in Software

The first aspect of this proposal deals with developing techniques to make the modern hardware more robust in handling transient hardware failures.

2.1 Problem

Studies of Windows servers at Microsoft demonstrate the scope of the problem of transient hardware failures [2]. In one study of Windows servers, eight percent of systems suffered from a storage or network adapter failure [2]. Many of these failures are transient: hardware vendors repeatedly report that the majority of returned devices operate correctly and retrying an operation often succeeds [1, 3, 49]. In total, 9% of all unplanned reboots of servers at Microsoft during a separate study were caused by adapter or hardware failures. Most importantly, when running platforms with *the same adapters* and software that tolerates hardware faults, reported device failures rates drop from 8 percent to 3 percent [2]. This evidence suggests that (1) *device failure is a major cause of system crashes*, (2) *transient device failures are common*, and (3) *drivers that tolerate device failures can improve reliability*. Without addressing this problem, the reliability of operating systems is limited by the reliability of devices.

The Linux kernel mailing list contains numerous reports of drivers waiting forever and reminders from kernel experts to avoid infinite waits [37]. Nevertheless, this code persists. For example, the code below from the 3c59x.c network driver in the Linux 2.6.18.8 kernel will loop forever if the device never returns the right value:

```
while (ioread16(ioaddr + Wn7_MasterStatus))
        & 0x8000)
    ;
```

2.2 Carburizer

Major OS vendors provide recommendations to driver writers on how to tolerate device failures [2, 24, 29, 61].

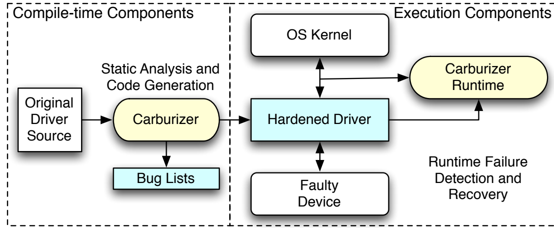

Figure 1: **The carburizer architecture. Existing kernel drivers are converted to hardened drivers and execute with runtime support for failure detection and recovery.**

Validation
Input validation. Check pointers, array indexes, packet lengths, and status data received from hardware [61, 24, 29]. ★
Unrepeatable reads. Read data from hardware once. Do not reread as it may be corrupt later [61]
DMA protection. Ensure that the device only writes to valid DMA memory [61, 29]
Data corruption. Use CRCs to detect data corruption if higher layers will not also check [61, 29]
Timing
Infinite polling. Ensure that spinning while waiting on the hardware can time out, and bound all loops [61, 29, 24]. ★
Stuck interrupts. Handle interrupts that cannot be dismissed [26, 61] ★
Lost request. Use a watchdog to verify hardware responsiveness [2, 24] ★
Excessive delay. Avoid delaying the OS, busy waiting, and holding locks for extended periods [2, 24]
Unexpected events. Handle out-of-sequence events [29, 24]
Reporting
Report hardware failures. Notify the operating system of errors, log all useful information [2, 24, 29, 61] ★
Recovery
Handle all failures. Handle error conditions, including generic and hardware-specific errors [2, 24, 61] ★
Cleanup properly. Ensure the driver cleans up resources after a fault [61, 29] ★
Conceal failure. Hide recoverable faults from applications [24] ★
Do not crash. Avoid halting the system [2, 24, 29, 55] ★
Test drivers. Test driver using fault injection [76, 26, 29]
Wrap I/O memory access. Use only wrapper functions to perform programmed/memory-mapped I/O [61, 29, 55]

Table 1: **Vendor recommendations for hardening drivers against hardware failures. Recommendations addressed by Carburizer are marked with a ★.**

Table 1 summarizes the recommendations of Microsoft, IBM, Intel, and Sun on how to prevent faulty hardware from causing system failures. The advice can be condensed to four major actions:

1. *Validate.* All input from a device should be treated as suspicious and validated to make sure that values lie within range.
2. *Timeout.* All interaction with a device should be subject to timeouts to prevent waiting forever when the device is not responsive.
3. *Report.* All suspect behavior should be reported to an OS service, allowing centralized detection and management of hardware failures.
4. *Recover.* The driver should recover from any device failure, if necessary by restarting the device.

To solve the above problem, we have developed a system called *Carburizer* that will implement the above recommendations automatically. Carburizer is a code-manipulation tool with an associated runtime that automatically hardens drivers. A hardened driver is one that can survive the failure of its device and if possible, return the device to its full function. Carburizer implements three major hardening recommendations: (1) validate inputs from the device, (2) verify device responsiveness, and (3) report hardware failures so that an administrator can proactively manage the failing hardware [2, 24, 29, 61].

Carburizer comprises of a static analysis component and a runtime The static analysis component runs on commodity drivers to detect where the driver uses data from device in critical control or data paths that can potentially cause the system to crash or hang if the device generates corrupt values. We categorize such uses as *hardware dependence bugs*. Carburizer also repairs this code by ensuring necessary bounds, range and timeout checks on device data before its risky use. Additionally, Carburizer statically checks for missing error reporting information in drivers about device failures and fixes them. Error reporting information about device failures be useful for central fault management systems to diagnosis system failures and save debugging time. The result of the static analysis phase is a hardened binary that executes with a runtime component of Carburizer. The runtime component ensures that stuck or missing interrupts do not occur by monitoring driver execution and device responses. We intend to resort to polling if an interrupt related bug is detected at runtime. The final piece of the runtime component provides online recovery. We leverage past work on shadow drivers [66, 32] to provide this functionality without any dependence on any performance reducing isolation mechanisms. Figure 1 describes the architecture of the above described system.

2.3 Results

We successfully implemented Carburizer [31] and were able to find 992 hardware dependence bugs in the Linux

2.6.18.8 driver tree. We also found approximately 1100 cases where the driver was missing error reporting information. Also, Carburizer hardened driver and runtime imposed less than one half percent CPU overhead when compared to a regular system [31].

3 On-demand isolation

Modern driver isolation systems are yet to be widely adopted and we suspect this is due to two primary reasons. First, driver isolation systems inflict performance overhead over driver execution. Second, drivers have a very inter-twined interaction with the kernel, which requires writing wrappers over the driver-kernel interface that bloats the isolation system as more and more drivers are supported. This poses maintainability issues that deters adoption of such systems. Furthermore, an unconventional way of interacting with the kernel, such as directly manipulating kernel data structures, either breaks the kernel or the reliability mechanisms.

We present *On-demand isolation*, a system that detects vulnerable entrypoints in the drivers and isolates them within the same address space. This isolation enables one to perform protection mechanisms exclusively on the vulnerable component, limiting the performance penalties of isolation. We also introduce zero overhead recovery, a technique that uses existing driver code to restore the driver and device to a previous consistent state without restarting the whole driver. In this proposal, we demonstrate the applicability of the solution in the context of driver security.

3.1 The problem of driver security

Device drivers represent a unique threat to system security and reliability because they (1) run in the kernel, where they have unfettered access to hardware and operating system data, and (2) interact with the outside world, outside the control of the OS. For example, there have been recent attacks demonstrated against USB devices, in which merely plugging a device into a USB port can compromise a system [44], and WIFI drivers, in which receiving a certain packet can similarly result in compromises. Furthermore, there are many known vulnerabilities in drivers from unprivileged user-mode code. A vulnerability in a Windows driver can enable remote code execution just by visiting a malicious webpage [43]. Also, devices have full access to system memory by virtue of DMA capability. In absence of any protection via IOMMUs, devices can DMA anywhere and bring the system down or execute any code in the system. These violations occur due to malicious inputs from the user-level or from the device. [15, 16, 19, 30]

Compounding the problem, device driver code is often provided by third-party device manufacturers or vendors, and hence does not reach the same level of assurance as other kernel code. We propose to address the problem of insecure drivers with a two prong approach: First, we detect suspicious code using static analysis. These results can be augmented by programmers to include untested code. Second, we perform isolation and recovery of these selected portions. The selective isolation ensures zero overhead to the system when running non-vulnerable code. Our isolation and recovery techniques can also be extended to untested driver code or recovery code.

To solve the above problem, we present a solution that is performant. We present a solution that isolates vulnerable driver entry points. The subsequent recovery mechanism also does not reload the driver but just reinitalizes the device to a consistent state.

As described earlier, drivers become vulnerable when they use values from hardware and user space that corrupts the privileged domain. Table 2 describes vulnerability types and how they occur in the context of device drivers. These vulnerabilities lead to denial of service attacks or lead to executing arbitrary code in the kernel.

Existing solutions of moving drivers to user-space or isolating them are too slow and require intrusive changes that are incompatible with existing driver code base [64, 70]. Most of these solutions also interpose on the standard driver-kernel interface, which is not a scalable approach. The isolation is also broken when drivers do not conform to class interfaces and provide non-standard ioctls, proc/sysfs entry points and module parameters. This is not uncommon. Our solution is aimed at automatically detecting the vulnerability and minimizing its effects of isolating a part of driver code rather than reducing the damage caused by this vulnerability by reducing privileges of the entire driver domain.

3.2 Threat Model

We focus improving the security of benign but buggy drivers, which unprivileged attackers either outside the kernel or outside the computer can exploit. While openly malicious drivers pose much greater problems, that problem is better solved by securing the distribution of drivers. Furthermore, the protocol between a driver and a device is private and not known to the operating system. Thus, the kernel cannot mediate access between the driver and device to prevent the driver or device from compromising security. For example, a driver may surreptitiously communicate with the device in ways unknown to the kernel: a compromised network driver could send a copy of every packet to an attacker, and a storage driver could intentionally corrupt key files or copy private data into public locations. Hence, we focus our efforts on vulnerable device drivers. The problems of driver security are broader than driver reliability. Apart from preventing kernel crashes, we also need mecha-

Problem	Manifestation in hardware	Driver vulnerability
Format string	Device/user passes unexpected values.	Driver prints device registers, ioctl params from user space.
Buffer overflow	Device/user passes unexpected values.	Driver performs memory operations using device values.
Interrupt Storm	Device sets the interrupt pin indefinitely.	Driver cannot detect interrupt storms.
Driver bugs	Device creates unexpected conditions.	Driver cannot handle timing delays from the device.
Live locks	Device or user keeps the driver waiting.	Driver uses device values for critical control flows.
Resource drain out	Device consumes kernel resources.	Driver allocates resources of device/ioctl value sizes.
Resource use after free	Device access dangling pointers.	Driver provides interface to freed resources.
DMA Issues	User/device overwrites kernel memory.	Driver does not use kernel provided addresses for DMA.

Table 2: **Manifestation of security problems in device drivers.**

nisms to prevent eavesdropping of kernel data structures and attacks involving code injections.

3.3 On-demand isolation

We intend to provide a solution that is performant and does not isolate the whole driver. It also should not interpose between kernel and driver, to avoid maintainability problems from supporting different driver classes and all drivers from a particular class. The recovery in case of a crash should also be non-intrusive and should not unload/re-load the whole driver. This is done by isolating small vulnerable portions of driver code called as *On-demand isolation*.

3.3.1 Propagation from vulnerability to crash

Providing On-demand isolation and recovery makes sense when the distance from vulnerability to crash is short. Prior work [25] has found this distance to be short in the kernel code. Additionally, around 96% of faults from hardware are shown to be detected or masked out within a short instruction window in the operating system [36]. In case of the problem of driver security from malicious user or hardware inputs, we suspect this distance to be even smaller. This is because drivers tend to read a value from device or userspace and tend to use it immediately instead of passing it along across different kernel subsystems or accessing it later. These results are encouraging. Moving selected vulnerable code portions like the WIFI handshake code path or the previously known bugs of USB device registration with the kernel can improve driver security without compromising performance of the whole system.

3.3.2 Determining isolation boundaries

In order to provide the benefit of minimal isolation, we need to automatically detect correct boundaries of isolation. We need to ensure that the boundaries of isolation are minimal enough to let the remaining system provide good performance yet large enough to support adequate driver and device recovery. We propose detecting isolation boundaries automatically when the system consumes device/user information in a potentially unsafe manner or through programming annotations from the user based on her apriori knowledge of vulnerability or untested code.

To meet the above goal, we develop a static analysis tool that marks appropriate isolation boundary given an approximate location of the bug. This tool can work with any auxiliary static analysis tools that detect vulnerability or can use programmer annotations. The tool is based on data flow analysis that detects when the malicious data is inserted into the system with respect to the vulnerability and generates the isolation boundaries. We use language based taint propagation techniques to detect driver code which uses device data in vulnerable scenarios. To do so, we re-use the Carburizer taint tracking infrastructure that uses taint propagation across inter-function boundaries for corrupt device inputs for detection of infinite loops, memory de-reference, array indexing and system panics [31]. We extended the existing Carburizer infrastructure to provide support to detect user and hardware security bugs and also support (1) taint propagation via procedure arguments (2) insecure uses of these tainted argument and 3)detection of tainted data from more functions and procedures rather than simple read/write from register or memory mapped I/O.

The above step helps us detect vulnerable components in a driver. Additionally, the programmer can also manually annotate untested code or code which has been declared vulnerable but has not yet been patched [13]. We use these techniques and mark the appropriate driver *entrypoint* as safe or vulnerable(hence isolated). Entrypoint granularity enables us to discard the state of a thread in the driver in case of a failure eliminating the need to maintain driver state.

3.3.3 On-demand isolation

The goal of our isolation mechanism is to provide memory protection guarantees for a small part of the driver code. In order to isolate selected parts of driver, we intend to split driver code into safe and vulnerable parts as shown in Figure 2. The vulnerable driver component lies in a single contiguous address space along with all its required data structures and memory referenced. Furthermore, the vulnerable component is subjected to memory

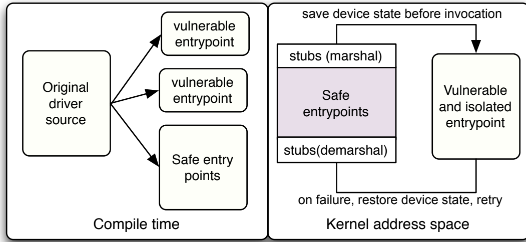

Figure 2: **On-demand isolation architecture. Existing kernel drivers are split into safe and unsafe components. The vulnerable component runs in a separate contiguous memory location with bounds check and separate stacks for return addresses. On failure, recovery is provided by discarding the execuion of vulnerable component and restoring the device to a known good state using existing suspend-resume functionality in the driver.**

bounds check on each operation using techniques like segmentation or software fault isolation. The vulnerable component also executes with a separate stack for return addresses.

The two components communicate using procedure calls wrapped by stubs. In the safe component, direct calls to functions implemented in the vulnerable component are replaced with upcalls through stubs. Stubs marshal data structures accessed by the called function and unmarshal them when the call returns. A symmetric downcall mechanism enables the user-mode component to invoke kernel functions. On-demand isolation does not update the kernel data structures concurrently. Updates to kernel data structures are withheld until the isolated portion of driver code finishes. This is also true for data structures representing the driver state shared by other entry points. Hence, the vulnerable component maintains copies of resources requested and those shared by the whole driver in its own contiguous address space. An object tracker, similar to the one used by Nooks [65], synchronizes copies of a data structures across the two components.

To split drivers, we used a modified version of DriverSlicer from microdrivers [23]. The goal of microdrivers was to move maximum functionality into user-space i.e. in a separate address space. The core driver components which cannot be handled in user-space, like interrupt handler remained in the kernel. However, On-demand isolation aims to provide isolation for minimal portion of driver code within the same address space. The communication between different components in On-demand isolation is simple procedure calls instead of RPC, since On-demand isolation operates within the same address space. This also helps retain interrupt functionality even in the isolated component, unlike microdrivers.

In the above architecture safety is ensured by 1)Executing vulnerable component in a separate contiguous space, 2) the marshaling properties and call by value-result semantics give automatic validation to the structural properties of kernel data structures. Marshaling data also ensures that data from user/device or from driver execution is only copied onto specific areas in kernel memory as controlled by marshal ling.

3.4 No overhead Recovery

Existing recovery mechanisms clean up the whole drivers and unload and reload the driver and replay the state [17, 67, 64]. These solutions are too drastic and since drivers require significant time to initialize [32] these solutions are slow. As mentioned earlier, we propose providing recovery at entrypoint granularity. Upon failure we discard the entry point state and restore the device to last good state without unregistering the driver with the kernel. There are several issues which need to be addressed while providing such a recovery mechanism in drivers:

(i) We need a mechanism to save and restore state of a running device that works across a range of devices/drivers. Often, devices have specific nuances that require special considerations while starting/stopping the driver.

(ii) Since such a recovery is provided on a running driver, we need to ensure that the other threads accessing the device are dealt with appropriately and do not corrupt themselves, device state or the recovery process.

(iii) Devices can also make persistent, irreversible changes that cannot be recovered. We need to identify such code early on and not provide any recovery guarantees for such code.

To solve the first issue, I propose reusing the *suspend* and *resume* functions in existing drivers. These functions already save device configuration state and suspend to RAM. Upon resume, this code restores state and already handles device specific “quirks” to get the device to a previous working state. We intend to automatically generate recovery code by re-using this code from drivers and stripping off code that suspends the device. The I/O memory and address space are saved before being mapped to the address space of the isolated process and restore prior to restoring the configuration registers and invoking resume.

When we attempt to recover the driver to a known state and reinitialize device configuration, other threads may be running inside the driver if the driver is designed to be multi-threaded. In this case, we need to leverage existing locks in the driver to stall other driver threads them until the recovery finishes. For threads already running, we mark driver code pages as no-execute. When threads execute and raise an exception for running

non-executable code, we trap the exception and halt the threads in the driver temporarily. We, however, allow the recovery threads to continue and restore the device. This method of restoring the driver is safe because the kernel and driver state is unaffected by the isolated vulnerable component in the event of a failure. While accessing the device, drivers ensure that they synchronize operations with other threads using kernel synchronization constructs like `spin_lock_irq_save`. While the driver synchronization constructs ensure that multiple threads do not clobber device state, the object tracker constructs ensures that these locks are removed in the event of a recovery to ensure other threads do not remain blocked.

To ensure that we do not make recovery guarantees for a non-recoverable case, we can detect when the isolation boundaries lie on such irreversible path and warn the user at compile time and during code execution.

4 Understanding and improving modern driver code

Device drivers are the single largest contributor to operating system kernel code with over 5 million lines of code in the Linux kernel and cause significant complexity, unreliability, and development cost. Recent years have seen a flurry of research aimed at improving the reliability and simplifying the development of drivers. However, little is known about what constitutes this huge body of code apart from few bugs that appear in these research studies.

In this part of my thesis, we study the source code of Linux drivers to determine whether assumptions made by most driver research, such as that all drivers belong to a class, are indeed true. We also review driver code and abstractions to review whether drivers can benefit from code re-organization or changing hardware trends. We develop a set of static-analysis tools to slice driver code across various axes. Broadly, our study addresses looks at three aspects of driver code (i) what is the function of driver code to find out what does all this driver code do,(ii) what is the abstraction of driver code, i.e. how does the driver code interact with the kernel and the driver code (iii) what is the form of driver code, are there similarities that can cause driver cause reduction. We make certain hypothesis in all three aspects of driver code and review if they hold true through our studies.

4.1 Overview

Modern computer systems are communicating with increasing number of devices, each of which requires a driver. For example, a modern desktop PC may have tens of devices, including keyboard, mouse, display, storage, and USB controller. Several studies have shown that drivers are the dominant cause of OS crashes [21, 46]. As a result, there has been a recent surge of interest in techniques to tolerate faults in drivers [20, 64, 67, 74], to improve the quality of driver code [10, 33], and for new driver architectures that remove driver code from the kernel [8, 22, 34, 40, 51, 70].

However, most research on device drivers focuses on a small set of devices, often a network card, sound card, and storage device, all using the PCI bus. However, these are but a small set of all drivers, and results from these devices may not generalize to the full set of drivers. For example, many devices for consumer PCs are connected over USB. Similarly, the tested devices are fairly mature and have standardized interfaces, but many other devices may have significant functionality differences. Furthermore, there are over sixty classes of drivers in Linux, so any small number may not generalize well.

This part of my dissertation presents a study over *all* the driver code in the Linux kernel in order to broadly characterize driver code. We focus on (i) what driver code does, meaning what is the function of the millions of lines of code comprising drivers, (ii) where are there opportunities for abstracting driver functionality into common libraries or subsystems?, and (iii) what are the oppurtunites in providing different abstractions to driver code given the demands of isolation and changing technology trends. We use two sets of static analysis tools to obtain above results. To understand function of driver code, we develop a control-flow analysis tool that detects properties of drivers at the granularity of functions. We also develop a a tool based on *shape analysis* [39] for detecting similar code.

Through this study, we seek to

- Show a high-level taxonomy of Linux drivers by interface, identify active areas of driver development.
- Find the characteristics of functionality provided by different parts of driver code and verify whether modern driver research applies to all drivers.
- Find instances of driver functionalities that are substantially similar across multiple drivers and determine the best way to abstract these commonalities.
- Discuss the driver-device communication and discuss trade-offs of abstracting drivers using different driver architectures.

4.2 Modern device drivers

Driver/Device Taxonomy The core operating system kernel interacts with device drivers through a small set of interfaces that abstract the fundamental nature of the device. In Linux, the three categories of drivers are *character* drivers, which are byte-stream oriented; *block* drivers, which support random-access to blocks; and *network* drivers, which support streams of packets. However, below these top-level interfaces, support libraries provide

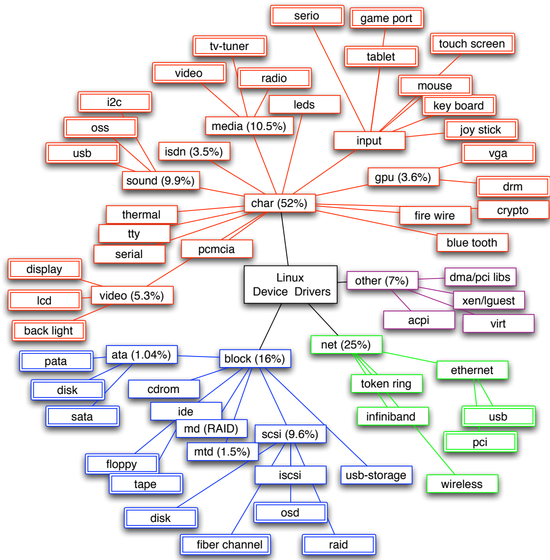

Figure 3: **The Linux driver taxonomy in terms of basic driver classes. The size (in percentage of lines of code) is mentioned for 5 biggest classes. Not all driver classes are mentioned.**

common interfaces for many other families of devices, such as keyboards and mice within character drivers.

Figure 3 shows the hierarchy of drivers in Linux according to their interfaces, starting from basic driver types (*i.e.*, char, block and net). Note that while Linux organizes related drivers in directories, this taxonomy is not the same as the Linux directory organization: all network drivers are under the *drivers/net* directory, but block drivers are split by their interfaces under *drivers/scsi*, *drivers/ide* and others.

We detect the class of a driver not by the location of its code, but by the interfaces it registers: *e.g.*, `register_netdev` indicates a driver is a network device. We consider a single driver as a module of code that can be compiled independently of other code. Hence, a single driver can span multiple files, and a single driver can serve multiple pieces of hardware. We consider all device drivers, bus drivers and virtual drivers that constitute the driver directories in Linux.

In contrast to the rich diversity of Figure 3, most research neglects the heavy tail of character devices that represent driver code. For example, video and GPU drivers contribute significantly towards driver code, but are untouched by driver research.

4.3 Research assumptions

Most research makes some simplifying assumptions about the problem being solved, and driver research is no different. For example, Shadow Drivers [64] assume that all drivers are members of a class and there are no unique interfaces to them. Similarly, the Termite driver-synthesis system assumes that drivers are state machines and perform no computations [53].

Systems that interpose on driver/device communication, such as Nooks [67], typically assume that communication occurs over procedure calls and not shared memory. Similarly, Shadow Drivers assume that the complete state of the device is available in the driver, by capturing kernel/driver interactions [64]. However, network cards that do TCP-offload may have significant protocol state that is only available in the device, and hence cannot be captured by monitoring the kernel/driver interface.

Recent efforts at synthesizing drivers from a formal specification look at generating a driver for a single chipset [53]. However, many drivers support more than one chipset. Hence, synthesizing the replacement for a single driver may require many more drivers. Similarly, enforcing safety properties for specific devices [70] may be cumbersome if many chipsets must be supported for each driver. Other efforts at reverse engineering drivers [11] may also be complicated by the support of many chipsets with different hardware interfaces. Furthermore, these synthesis and verification systems assume that the device always behaves correctly, and may fail unpredictably with faulty hardware. An overwhelming support for chipsets would also indicate driver interface support for multiple chipsets.

Another assumption made by driver research is that drivers are largely a conduit for communicating data and signaling to the device, and that they perform little processing. Neither, RevNIC [11] nor Termite [53] support data processing with the driver, because it is too complex to model as a simple state machine.

While these assumptions all hold true for many drivers, this research seeks to quantify their generality. If these assumptions are true for all drivers, then these research ideas have broad applicability. If not, then perhaps new research is needed to address the outliers.

4.4 Driver Code Functionality

We seek to develop an understanding of what driver code does. The goal of this study is to verify the driver assumptions described in the previous section, and to identify major driver functions that could benefit from additional research. We also review how the driver subsytems developed are applicable to all drivers.

Methodology To study the driver code, we developed static analysis tools using CIL [45] to detect code properties in individual drivers. The tool takes as input unmodified drivers in the Linux kernel source tree and a list of driver data-structure types and driver entry points. As drivers only execute when invoked from the kernel, these entry points allow us to determine the purpose of

particular driver functions. We then construct a control-flow graph of the driver that allows us to determine all the functions reachable through each entry point.

We use a *tagging* approach to labeling driver code: the tool tags a function with the label of each entry point from which it is reachable. Thus, a function called only during initialization will be labeled initialization only, while code common to initialization and shutdown will receive both labels. Through our studies, we should be able to prove or disprove the following hypotheses.

1. *Device drivers cannot be completely synthesized because (i) Not all drivers can be completely defined by the class definition of drivers, (ii) A single driver supports multiple chipsets and (iii) Drivers perform significant processing rather than just being a conduit of data.*

2. *A significantly small amount of driver code is dedicated to perform the core I/O functionality.*

To test the above hypotheses, we answer the following questions about driver code using our static analysis.

Do drivers belong to classes? Driver research often assumes that drivers belong to class. However, many drivers support proprietary extensions to the class interface. In Linux drivers, these manifest as private `ioctl` commands, options exported through `/proc` or `/sysfs`, and as load-time parameters. If a driver has none of these code features, we assume it implements only the specified class functionality. However, drivers with one or more of these extensions may have additional behaviors not captured by the class.

Do drivers to significant processing? As devices become more powerful and features processors of their own, it is often assumed that drivers perform little processing and simply shuttle data between the OS and the device. However, if drivers require substantial CPU time, for example to compute parity for RAID, checksums for networking, or computing display data for video drivers then processing power must be reserved.

How many chipsets does a single driver support? We measure the number of chipsets supports by each Linux driver by counting the number of PCI, USB or other bus device IDs (*i.e.*, i2c, ieee1394) that the driver recognizes. This will help us know the efficiency of driver code and also how applicable are the driver synthesis techniques.

Where is the driver code changing? Over time, the focus of new driver development shifts as new device classes become popular. This guides driver research, by showing what driver code is actively developed and likely to benefit from new reliability, security, and performance techniques.

How is the functionality provided by device drivers distributed? We measure the amount of code dedicated to the functionality provided by device drivers across driver classes such as initialization, cleanup, error handling, configuration, I/O, power management.

4.5 Driver code form

Given that all the drivers for a class perform essentially the same task, one may ask why so much code is needed. In some cases, such as SCSI and IDE devices, related devices share most of the code with a small amount of per-device code. Most drivers, though, replicate most driver functionality for every device. Without a global view of drivers and the type of sharing going on, it can be difficult to tell whether there are yet more opportunities to share common code. In this section, we hypothise that *hundreds of drivers repeat most functionality and one can replace significant amounts of driver code and replace them with a table or domain specific language*.

To address this question, we will develop a tool for discovering similar code patterns at function granularity across related drivers and applied it to Linux drivers. The goal of this work is to find driver entry points with substantially similar code, indicating that perhaps the common code could be abstracted into a library and removed from all drivers.

Methodology Our similarity tool is based on *shape analysis* [39]. The tool generates a set of multidimensional coordinates using static analysis for every function in every driver, and then detects as related two functions whose coordinate sets (its *shape*) are nearby. The similarity tool processes a driver function and adds a point to the driver's shape for every loop, kernel function call, device interaction, variable assignment, and return statement. The coordinates of the point are the offset into the function (line number) and the statement type. Thus, the shape of each driver function is a cloud of points on plane.

To simplify comparison of two driver functions, we further reduce the shape of a driver down to a single *signature* value. We compute the signature as a function of the Euclidean distance between all the points in the code cluster obtained above. Thus, two functions with identical code will have identical signatures, and code that is similar, in that it has a similar structure of loops and I/O operations, will have nearby signatures.

Our preliminary results demonstrate that there are large swaths of driver code that are virtually identical across drivers. This code is unnecessarily complicated, and its repetition causes extra work when kernel/driver interfaces changes.

4.6 Driver Code Abstractions

The preceding section focused on the *function* of driver code, and here we turn to the *form* of driver code. We review whether it is possible to move all driver code to user-space to provide benefits of isolation and additional processing power. We look at the patterns of interaction between the driver, the kernel and the device, with a focus on (i) the patterns of kernel interactions and (ii)resource consumption, and (iii) the patterns of device interaction. The goal of this study is to investigate the *architecture* of drivers, to design new architectures that achieve better reliability, higher performance, or lower complexity.

Methodology We apply the static-analysis tool from the previous section again. However, while the preceding section was a top-down analysis of the call graph, propagating labels from driver entrypoints through all driver functions, here we start at the bottom: kernel and device interactions. Using a list of known kernel functions and device I/O methods supporting memory-mapped I/O, port I/O, and DMA, we label driver functions according to the methods they invoke. Through our studies, we should be able to prove or disprove the following hypotheses.

1. *Most modern devices provide enough resources to execute drivers of their corresponding class.*
2. *USB drivers provide the best trade-offs between functionality and isolation in modern driver architectures.*

To test the above hypotheses, we review the following questions to understand the resource allocation and the granularity of device and kernel accesses by device drivers.

How often do drivers talk to devices? We review what mechanisms to access the devices remain relevant and also how frequent is the device functionality invoked. The nature of interaction should guide us in understanding the abstraction of the driver architectures.

What is the kernel resource usage of drivers? Drivers use memory, CPU and support of kernel services for their own functioning. We review the kernel resource usage of the drivers and try to ascertain how much percentage of devices can support the processing and memory requirements to run their corresponding drivers.

Both the questions above will help us better understand the tradeoffs of isolation, functionality and performance. Too frequent interactions, too many resource requirements such as memory, CPU and kernel services prohibhit moving driver code out of the kernel.

What is the concurrency model for drivers? We also investigate the programming styles for drivers: do they tend towards threaded code, saving state on the stack and blocking for events, or toward event-driven code, registering callbacks either as completion routines (for USB drivers) or interrupt handlers and timers for PCI devices. Threaded code is usually error prone and leads to concurrency bugs and crashes especially when handling interrupt code.

Which bus architecture best isolates drivers? We review the PCI, USB and Xenbus architecture and review the trade-off between the isolation provided by the driver versus the functionality exposed in these drivers.

4.7 Conclusion

Through this study, we understand how much the assumptions made by different researchers apply to driver code. We also intend to propose some insight on driver functionality, abstraction and form that will help us better understand device drivers.

5 Related work

My dissertation draws inspiration from past projects on driver reliability, bug finding, automatic patch generation, device interface specification, and recovery.

Driver isolation Past work on driver reliability has focused on preventing driver bugs from crashing the system. Much of this work can apply to hardware failures, as they manifest as a bug causing the driver to access invalid memory or consume too much CPU. In contrast to our mechanisms, these tools are all heavyweight: they require new operating systems (Singularity [58], Minix [27], Nexus [70]), VINO [54], new driver models (Windows UMDF [42], Linux user-mode drivers [35]), runtime instrumentation of large amounts of code (XFI [69] and SafeDrive [75]), adoption of a hypervisor (Xen [20] and iKernel [68]), or a new subsystem in the kernel (Nooks [67]).

Apart from these source level isolation approaches, there are binary approaches that provide software fault isolation. However, trying to segregate driver kernel boundaries at binary level may be too late for drivers where driver and kernel memory are heavily intertwined with callbacks. Hence, these approaches have been limited to user-space programs [48, 73].

In comparison, the tools introduced in my proposal are light weight. Carburizer instead fixes specific bugs, which reduces the code needed in the kernel to just recovery and not fault detection or isolation. On-demand isolation also attempts to isolate only the vulnerable components in the driver to reduce isolation costs. Microdrivers [23] isolates all functions in user-space except performance critical routines like interrupt handlers, which cannot be supported in user-space. Due

to their limited overheads, we feel that Carburizer and On-demand isolation may be easier to integrate into existing kernel development processes. Furthermore, our mechanisms detects hardware failures and breaches before they cause corruption, while driver reliability systems using memory detection mentioned may not detect it until much later, after the corruption propagates through the system.

Bug finding Tools for finding bugs in OS code through static analysis [5, 6, 18] have focused on enforcing kernel-programming rules, such as proper memory allocation, locking and error handling. However, these tools enforce kernel API protocols, but do not address the hardware protocol. Furthermore, these tools only find bugs but do not automatically fix them.

Hardware dependence errors are commonly found through synthetic fault injection [2, 26, 61, 76]. This approach requires a machine with the device installed, while Carburizer and On-demand fault isolation operates only on source code. Furthermore, fault injection is time consuming, as it requires injection of many possible faults into each I/O operation made by a driver.

Automatic patch generation Carburizer is complementary to prior work on repairing broken error handling code found through fault injection [62]. Error handling repair is an alternate means of recovering when a hardware failure occurs by re-using existing error handling code instead of invoking a generic recovery function. Other work on automatically patching bugs has focused on security exploits [14, 56, 57]. These systems also address how to generate repair code automatically, but focus on bugs used for attacks, such as buffer overruns, and not the infinite loop problems caused by devices.

Hardware Interface specification Several projects, such as Devil [41], Dingo [52], HAIL [60], Nexus [70], Laddie [71] and others, have focused on reducing faults on the driver/device interface by specifying the hardware interface through a domain specific language. These languages improve driver reliability by ensuring that the driver follows the correct protocol for the device. However, these systems all assume that the hardware is perfect and never misbehaves. Without runtime checking they cannot verify that the device produces correct output.

Recovery Carburizer relies on shadow drivers [63] for recovery which was one of the first systems to propose driver recovery. However, Carburizer's implementation of shadow drivers does not integrate any isolation mechanism, the overhead of recovery support is very low. Other systems that recover from driver failure, including SafeDrive [75], and Minix [27], rely on similar mechanisms to restore the kernel to a consistent state and release resources acquired by the driver could be used as well. CuriOS provides transparent recovery and further ensures that client session state can be recovered [17]. However, CuriOS is a new operating system and requires specially written code to take advantage of its recovery system, while our mechanisms work with existing driver code in existing operating systems. The recovery system proposed by On-demand isolation is much fine grained and light weight and offers a new perspective that drivers already provide much of the necessary functionality to provide recovery mechanisms.

To achieve high reliability in the presence of hardware failures, fault tolerant systems often use multiple instances of a hardware device and switch to a new device when one fails [7, 28, 59]. These systems provide an alternate recovery mechanism to shadow drivers. However, this approach still relies on drivers to detect failures, and the tools proposed in this proposal improves that ability.

6 Timeline

This section outlines the timeline of finishing the remaining work:

1. Spring 2011: Work on driver study.
2. Summer 2011: Internship
3. Fall 2011: Work on on-demand isolation and driver study.
4. Spring 2012: Work on on-demand isolation.
5. Summer 2012: Finish dissertation.

7 Other Research

In this section I briefly describe other research performed as a graduate student that is not a part of my dissertation. To improve functionality of virtual machines, I developed a system, called shadow driver migration, that enables live migration of guest operating systems that are directly attached to physical devices(for performance reasons). Without this work, virtual machines that leverage the performance advantage of direct access to I/O devices cannot reap the benefit of migration. My approach saves the state of a device in a shadow driver, that reconnects the OS to the local devices after migration. Shadow driver migration provides a low overhead, transparent migration mechanism that can be incorporated with moderate implementation efforts [8]. It also demonstrates that in order to make devices virtualization ready, vendors need to reduce device initialization time which can be a critical bottleneck in migration scenarios [32].

More recently, I looked at how to test device drivers without hardware using SymDrive, a system to test device drivers using symbolic execution. Symbolic exe-

cution provides provides the ability to test a driver completely without requiring any of the different device hardware the driver supports. The system also consists of a specification system that allows the developer to test a set of conditions each time the driver interacts with the kernel. This allows for thorough testing of drivers rather than just bug-finding. I worked on this project to support testing for USB drivers and developed an in-kernel USB subsystem for user mode linux, a USB trace record/replay system and did testing for PCI and USB network drivers [50].

8 Conclusion

The research described in this proposal will significantly improve the systems of tomorrow. It improves existing drivers, through improved tolerance of hardware failures, and proposes new isolation and recovery mechanisms to improve driver security. The dissertation proposal focuses on device drivers, these techniques are broadly applicable to most operating system extensions. I also hope that our driver study shapes future driver research and development in the right direction.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proc. of the Eighth IEEE HOTOS*, May 2001.

[2] S. Arthur. Fault resilient drivers for Longhorn server, May 2004. Microsoft Corporation, WinHec 2004 Presentation DW04012.

[3] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proc. of the 7th SIGMETRICS*, June 2007.

[4] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proc. of the 2006 EuroSys Conference*, 2006.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proc. of the 2006 EuroSys Conference*, Apr. 2006.

[6] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of the 29th POPL*, 2002.

[7] J. F. Bartlett. A NonStop kernel. In *Proc. of the 8th ACM SOSP*, Dec. 1981.

[8] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *USENIX ATC*, 2010.

[9] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *SOSP*, 2009. ACM.

[10] P. Chandrashekaran, C. Conway, J. M. Joy, and S. K. Rajamani. Programming asynchronous layers with CLARITY. In *Proc. of the 15th Annual Symposium on Foundations of Software Engineering*, Sept. 2007.

[11] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *Eurosys*, Apr. 2010.

[12] T.-c. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. *OSR*, 33:140–153, 1999.

[13] W. Cui, M. Peinado, H. Wang, and M. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. 2007.

[14] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proc. of the IEEE Symposium on Security and Privacy*, 2007.

[15] CVE. ntty_ioctl_tiocgicount function causes information leak. `http://www.cvedetails.com/cve/CVE-2010-4077/`, Nov 2010. CVE-2010-4077.

[16] CVE. Win32k improper user input validation vulnerability. `http://www.cvedetails.com/cve/CVE-2011-0086/`, Feb 2011. CVE-2011-0086.

[17] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. of the 8th USENIX OSDI*, December 2008.

[18] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of the 4th USENIX OSDI*, Oct. 2000.

[19] Exploit-DB. Windows vista/server 2008 ntusercheckaccessforintegritylevel use-after-free vulnerability. `http://www.exploit-db.com/exploits/14156/`, July 2010. EDB-ID: 14156.

[20] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *OASIS Workhop*, 2004.

[21] A. Ganapathi, V. Ganapathi, and D. A. Patterson. Windows xp kernel crash analysis. In *LISA*, 2006.

[22] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A new architecture for device drivers. In *Proc. of the Eleventh IEEE HOTOS*, 2007.

[23] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proc. of the 13th ACM ASPLOS*, Mar. 2008.

[24] S. Graham. Writing drivers for reliability, robustness and fault tolerant systems. `http://www.microsoft.com/whdc/archive/FTdrv.mspx`, Apr. 2004.

[25] W. Gu, Z. Kalbarczyk, R. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. 2003.

[26] S. R. Hanson and E. J. Radley. Testing device driver hardening, May 2005. US Patent 6,971,048.

[27] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proc. of the 2007 IEEE DSN*, June 2007.

[28] Hewlett Packard Corp. Parallel processing of TCP/IP with ethernet adapter failover. http://h20223.www2.hp.com/NonStopComputing/downloads/EAFailoverTCP-IP-PL.pdf, 2002.

[29] Intel Corporation and IBM Corporation. Device driver hardening design specification draft release 0.5h. http://hardeneddrivers.sourceforge.net/downloads/DDH-Spec-0.5h.pdf, Aug. 2002.

[30] IT Security Database. Rhsa-2010:0842: kernel security and bug fix update. http://www.itsecdb.com/oval/definition/oval/com.redhat.rhsa/def/20100842/RHSA-2010-0842-kernel-security-and-bug-fix-update-Importan.html, Nov 2010. CVE-2010-3904, CVE-2010-3437, CVE-2010-3442 ,CVE-2010-3705, CVE-2010-3698, CVE-2010-3432, CVE-2010-3301, CVE-2010-3084, CVE-2010-3081, CVE-2010-3079, CVE-2010-2962, CVE-2010-3904, CVE-2010-2955, CVE-2010-2803, CVE-2010-3705, CVE-2010-3698, CVE-2010-3442, CVE-2010-3437, CVE-2010-3432, CVE-2010-3301, CVE-2010-3084, CVE-2010-3081, CVE-2010-3079, CVE-2010-2962, CVE-2010-2955, CVE-2010-2803.

[31] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *SOSP*, 2009. ACM.

[32] A. Kadav and M. M. Swift. Live migration of direct-access devices. In *ACM SIGOPS Operating Systems Review, 'Best papers from VEE and Best papers from WIOV', Volume 43, Issue 3.*, 2009.

[33] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX ATC*, 2010.

[34] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 20(5), 2005.

[35] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal Computer Science and Technology*, 20(5), Sept. 2005.

[36] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proc. of the 13th ACM ASPLOS*, Mar. 2008.

[37] Linux Kernel Mailing List. Fixes for uli5261 (tulip driver). http://lkml.org/lkml/2006/8/19/59, Aug. 2006.

[38] LKML. Re:do not misuse coverity please. http://lkml.org/lkml/2005/3/27/131, March 2005. Jean Delware.

[39] S. Loncaric. A survey of shape analysis techniques. *Pattern Recognition*, 31(8):983–1002, 1998.

[40] A. Menon, S. Schubert, and W. Zwaenepoel. Twindrivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. In *ASPLOS*. ACM, 2009.

[41] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proc. of the 4th USENIX OSDI*, Oct. 2000.

[42] Microsoft Corporation. Introduction to the WDF user-mode driver framework. http://www.microsoft.com/whdc/driver/wdf/umdf_intro.mspx, May 2006.

[43] Microsoft Corporation. Vulnerabilities in windows kernel-mode drivers could allow remote code execution (969947). http://www.microsoft.com/technet/security/Bulletin/ms09-065.mspx, November 2009. Microsoft Security Bulletin MS09-065 - Critical.

[44] MWR InfoSecurity. Mwr infosecurity security advisory linux usb device driver buffer overflow. http://labs.mwrinfosecurity.com/files/Advisories/mwri_linux-usb-buffer-overflow_2009-10-29.pdf, October 2009.

[45] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the 11th International Conference on Compiler Construction*, 2002.

[46] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft WinHec Presentation DDT301, 2003.

[47] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proc. of the 2008 EuroSys Conference*, apr 2008.

[48] D. Peek, E. B. Nightingale, B. D. Higgins, P. Kumar, and J. Flinn. Sprockets: safe extensions for distributed file systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, 2007. USENIX Association.

[49] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of the 5th FAST*, 2007.

[50] M. J. Renzelmann, A. Kadav, and M. M. Swift. Poster: Symdrive:testing drivers without devices. In *OSDI*, 2010.

[51] M. J. Renzelmann and M. M. Swift. Decaf: Moving device drivers to a modern language. In *USENIX ATC*, June 2009.

[52] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proc. of the 200 EuroSys Conference*, Apr. 2009.

[53] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *SOSP*, 2009.

[54] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. *SIGOPS Operating Systems Review*, 30:213–228, 1996.

[55] T. Shureih. HOWTO: Linux device driver dos and don'ts. http://janitor.kernelnewbies.org/docs/driver-howto.html, Mar. 2004.

[56] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[57] A. Smirnov and Tzi-ckerChiueh. Automatic patch generation for buffer overflow attacks. In *Proc. of the 3rd Symposium on Information Assurance and Security*, 2007.

[58] M. Spear, T. Roeder, O. Hodson, G. Hunt, and S. Levi. Solving the starting problem: Device drivers as self-describing artifacts. In *Proc. of the 2006 EuroSys Conference*, Apr. 2006.

[59] S. Y. H. Su and R. J. Spillman. An overview of fault-tolerant digital system architecture. In *Proc. of the National Computer Conference (AFIPS)*, 1977.

[60] J. Sun, W. Yuan, M. Kallahalla, and N. Islam. HAIL: A language for easy and correct device access. In *Proc. of the 5th ACM International Conference on Embedded Software*, Sept. 2005.

[61] Sun Microsystems. *Solaris Express Software Developer Collection: Writing Device Drivers*, chapter 13: Hardening Solaris Drivers. Sun Microsystems, 2007.

[62] M. Süßkraut and C. Fetzer. Automatically finding and patching bad error handling. In *Proc. of the 6th EDCC*, Oct. 2006.

[63] M. Swift, M. Annamalau, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), Nov. 2006.

[64] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. of the 6th USENIX OSDI*, 2004.

[65] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. of the 19th ACM SOSP*, Oct. 2003.

[66] M. M. Swift, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation*, Dec. 2004.

[67] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), Feb. 2005.

[68] L. Tan, E. M. Chan, R. Farivar, N. Mallick, J. C. Carlyle, F. M. David, and R. H. Campbell. iKernel: Isolating buggy and malicious device drivers using hardware virtualization support. In *Proc. of the 3rd DASC*, 2007.

[69] Úlfar Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: software guards for system address spaces. In *Proc. of the 7th USENIX OSDI*, 2006.

[70] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of the 8th USENIX OSDI*, 2008.

[71] L. Wittie, C. Hawblitzel, and D. Pierret. Generating a statically-checkable device driver I/O interface. In *Workshop on Automatic Program Generation for Embedded Systems*, Oct. 2007.

[72] J. Yang. Zero-penalty RAID controller memory leak detection and isolation method and system utilizing sequence numbers, 2007. Patent application 11715680.

[73] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009.

[74] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th USENIX OSDI*, 2006.

[75] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the 7th USENIX OSDI*, Nov. 2006.

[76] L. Zhuang, S. Wang, and K. Gao. Fault injection test harness. In *Proc. of the Ottawa Linux Symposium*, June 2003.