Tolerating Hardware Device Failures in Software

Asim Kadav, Matthew J. Renzelmann and Michael M. Swift
Computer Sciences Department,
University of Wisconsin-Madison
{kadav,mjr,swift}@cs.wisc.edu

ABSTRACT

Hardware devices can fail, but many drivers assume they do not. When confronted with real devices that misbehave, these assumptions can lead to driver or system failures. While major operating system and device vendors recommend that drivers detect and recover from hardware failures, we find that there are many drivers that will crash or hang when a device fails. Such bugs cannot easily be detected by regular stress testing because the failures are induced by the device and not the software load.

This paper describes Carburizer, a code-manipulation tool and associated runtime that improves system reliability in the presence of faulty devices. Carburizer analyzes driver source code to find 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.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability; D.3.4 [**Programming Languages**]: Processors

General Terms

Reliability, Design

Keywords

Device Drivers, Reliability, Recovery, Debugging, Code Generation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.
Copyright 2009 ACM 978-1-60558-752-3/09/10 ...$10.00.

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.

Applications invoke devices through device drivers. The device and driver interact through a protocol specified by the hardware. When the device obeys the specification, a driver may trust any inputs it receives. Unfortunately, devices do not always behave according to their specification. Some failures are caused by wear-out or electrical interference [25]. In addition, internal software failures can occur in devices that execute embedded firmware, sometimes up to millions of lines of code [50].

Studies of Windows servers at Microsoft demonstrate the scope of the problem [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, 31]. 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.

Device hardware failures cause system hangs or crashes when drivers cannot detect or tolerate the failure. The Linux kernel mailing list contains numerous reports of drivers waiting forever and reminders from kernel experts to avoid infinite waits [26]. 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)
    ;
```

To address this problem, major OS vendors have issued recommendations on how to harden drivers to device failures [16, 41, 20]. These recommendations include validating all inputs from a device, ensuring that all code waiting for a device will terminate, and reporting all hardware failures. Despite these recommendations, we found that a large number of Linux drivers do not properly tolerate hardware failures. We see two reasons for this: (1) testing drivers against hardware failures is difficult, and (2) hardening drivers by hand is challenging. Common testing procedures, such as stress testing, will not detect failures related to hardware. Instead, fault-injection testing is required [2, 17, 52]. Unlike other software testing, device drivers require that an instance of the device be present, which limits the number of machines that can run tests.

Previous work on driver fault tolerance has concentrated on two major approaches: static bug finding [4, 6, 12, 32] and run-time fault tolerance [48, 46, 18, 51, 44]. Static approaches check for bugs in the interface between the driver and the kernel to ensure that the driver does not violate kernel-programming rules, such as by failing to release a lock. But, these tools do not verify that the driver validates inputs received from the device.

Systems that tolerate faults at run time, such as SafeDrive [51] and Nooks [44], either instrument driver code or execute it in an isolated environment. These systems detect faults, including hardware-induced faults, dynamically and trigger a recovery mechanism. However, these systems have had limited deployment, perhaps due to the heavyweight nature of the solution.

This paper presents Carburizer,[1] a code-manipulation tool and 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, 16, 20, 41].

Carburizer analyzes driver code to find where it accepts input from the device. If the driver uses device data without checking its correctness, Carburizer modifies the driver to insert validation code. If the driver checks device data for correctness, Carburizer inserts code to report a failure if the data is incorrect. Finally, the Carburizer runtime detects stuck interrupts and non-responsive devices and causes the driver to poll the device. To automatically repair bugs, Carburizer also invokes a generic recovery service that can reset the device. We rely on shadow drivers [43] to provide this recovery service.

Despite the common application of static analysis tools to the Linux kernel [9], Carburizer uncovers a large number of problems. Carburizer identified 992 bugs in existing Linux drivers where a hardware failure may cause the driver to crash or hang. With manual inspection of a random subset, we determined that the false positive rate is 7.4%, for approximately 919 true bugs found. Discounting for false positives, Carburizer repairs approximately 845 real bugs by inserting code to detect hardware failures and recover at runtime. When run with common I/O workloads, drivers modified by Carburizer perform similarly to native drivers.

In the remainder of this paper, we first discuss hardware failures and OS vendor guidelines for hardening drivers. We then present the three major functions of Carburizer in Sections 3, 4 and 5. Section 6 presents the overhead of our code changes, and we finish with related work in Section 7 and conclusions.

2. DEVICE HARDWARE FAILURES

In this section, we describe the problem of hardware device failures and vendor recommendations on how to tolerate and manage device failures.

2.1 Failures Types

Modern CMOS devices are prone to internal failures and without significant design changes, this problem is expected to worsen as transistors shrink. Prior studies indicate that these devices experience transient *bit-flip* faults, where a single bit changes value; permanent *stuck-at* faults, when a bit assumes a fixed value for an extended period; and *bridging faults* when an adjacent pair of bits are electrically mated, causing a logical-and or logical-or gate between the bits [47, 25]. Environmental conditions such as electromagnetic interference and radiation can cause transient faults. Wear-out and insufficient burn-in may result in stuck-at and bridging faults in the devices.

In addition, when a device contains embedded firmware, or even an embedded operating system [50], any software-related failure is possible, such as out-of-resource errors from memory leaks or concurrency bugs.

Failure manifestations.

Device drivers observe failures when they access data generated by the device. For PCI drivers, which perform I/O through memory or I/O ports, the driver reads incorrect values from the device. For USB drivers, which use a request/response protocol, a device failure may cause a response packet to contain incorrect data [25]. Sources at Microsoft report that device hangs and interrupt storms are common manifestations of faulty hardware [14].

Many hardware failures are likely to manifest as corrupt values in device registers. A single bit-flip internal to a device controller may propagate to other internal registers before the device driver reads a garbled value exposed through a device register. Similarly, an internal stuck-at failure may result in a transient corruption in a device register, a stuck value in a register, a stuck interrupt request line, or unpredictable DMA accesses. Bugs in device firmware may manifest as incorrect output values or timing failures, when a device does not respond within the specified time period.

2.2 Vendor Recommendations

Major OS vendors provide recommendations to driver writers on how to tolerate device failures [2, 16, 20, 41]. 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.

[1] Carburizing is a process of hardening steel through heat treatment.

Validation
Input validation. Check pointers, array indexes, packet lengths, and status data received from hardware [41, 16, 20]. ★
Unrepeatable reads. Read data from hardware once. Do not reread as it may be corrupt later [41]
DMA protection. Ensure that the device only writes to valid DMA memory [41, 20]
Data corruption. Use CRCs to detect data corruption if higher layers will not also check [41, 20]
Timing
Infinite polling. Ensure that spinning while waiting on the hardware can time out, and bound all loops [41, 20, 16]. ★
Stuck interrupts. Handle interrupts that cannot be dismissed [17, 41] ★
Lost request. Use a watchdog to verify hardware responsiveness [2, 16] ★
Excessive delay. Avoid delaying the OS, busy waiting, and holding locks for extended periods [2, 16]
Unexpected events. Handle out-of-sequence events [20, 16]
Reporting
Report hardware failures. Notify the operating system of errors, log all useful information [2, 16, 20, 41] ★
Recovery
Handle all failures. Handle error conditions, including generic and hardware-specific errors [2, 16, 41] ★
Cleanup properly. Ensure the driver cleans up resources after a fault [41, 20] ★
Conceal failure. Hide recoverable faults from applications [16] ★
Do not crash. Avoid halting the system [2, 16, 20, 34] ★
Test drivers. Test driver using fault injection [52, 17, 20]
Wrap I/O memory access. Use only wrapper functions to perform programmed/memory-mapped I/O [41, 20, 34]

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

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.

The goal of our work is to *automatically implement* these recommendations. First, we seek to make drivers tolerate and recover from device failures, so device failures do not lead to system failures. For this aspect of our work, we focus on transient failures that do not recur after the device is reset. Second, we seek to make drivers report device failures so that administrators learn of transient failures and can proactively replace faulty devices.

Carburizer addresses *all* four aspects of vendor recommendations described above. Section 3 addresses bugs that can be found through static analysis, including infinite polling and input validation. Section 4 addresses reporting hardware failures to a centralized service. Section 5 addresses runtime support for tolerating device failures, including recovery, stuck interrupts, and lost requests. The recommendations that Carburizer can apply automatically are marked in Table 1. The remaining recommendations can be addressed with other techniques, such as an IOMMU for DMA memory protection, or cannot be applied without semantic information about the device.

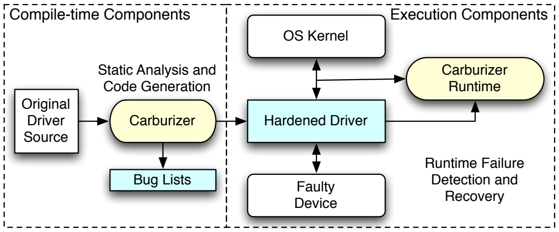


Figure 1: The Carburizer architecture. Existing kernel drivers are converted to hardened drivers and execute with runtime support for failure detection and recovery.

3. HARDENING DRIVERS

This section describes how Carburizer finds and fixes infinite polling and input validation bugs from Table 1. These are *hardware dependence* bugs that arise because the software depends on the hardware's correctness for its own correctness. The goal of our work is to (1) find places where driver code uses data originating from a device, (2) verify that the driver checks the data for validity before performing actions that could lead to a crash or hang, and if not, (3) automatically insert validity or timing checks into the code. These checks invoke a generic recovery mechanism, which we describe in Section 5. When used without a recovery service, Carburizer identifies bugs for a programmer to fix.

Figure 1 shows the overall architecture of our system. Carburizer takes unmodified drivers as input and with a set of static analyses produces (1) a list of possible bugs and (2) a driver with these bugs repaired, i.e. drivers that validate all input coming from hardware before using it in critical control or data flow paths. The Carburizer runtime detects additional hardware failures at runtime and can restore functionality after a hardware failure.

We implement Carburizer with CIL [30]. CIL reads in pre-processed C code and produces an internal representation of the code suitable for static analysis. Tools built with CIL can then modify the code and produce a new pre-processed source file as output.

We next describe the analyses for hardening drivers in Carburizer and our strategies for automatically repairing these bugs. We experiment with device drivers from the Linux 2.6.18.8 kernel.

3.1 Finding Sensitive Code

Carburizer locates code that is dependent on inputs from the device. When a driver makes a control decision, such as a branch or function call, based on data from the device, the control code is *sensitive* because it is dependent on the correct functioning of the device. If code uses a value originating from a device in an address calculation, for example as an array index, use of the address is dependent on the device. Carburizer finds hardware-dependent code that is incorrect for some device inputs.

Carburizer's analyses are performed in two passes. The first pass is common to all analyses and identifies variables that are *tainted*, or dependent on input from the device. Carburizer consults a table of functions known to perform I/O, such as `readl` for memory-mapped I/O or `inb` for port I/O. Initially, Carburizer marks all heap and stack variables that receive results from these functions as tainted. Carburizer then propagates taint to variables that are computed from

```
1 static int amd8111e_read_phy(.......)
2 {
3    .
4   reg_val = readl(mmio + PHY_ACCESS);
5   while (reg_val & PHY_CMD_ACTIVE)
6     reg_val = readl( mmio + PHY_ACCESS );
7    .
8 }
```

Figure 2: The AMD 8111e network driver (amd8111e.c) can hang if the readl() call in line 6 always returns the same value.

or aliased to the tainted variables. Carburizer considers the static visibility of variables but does not consider possible calling contexts. For compound variables such as structures and arrays, the analysis is field insensitive and assumes that the entire structure is tainted if any field contains a value read from the device. We find that in practice this occurs rarely, and therefore yields a simpler analysis that is almost as precise as being sensitive to fields.

The output of the first pass is a table containing all variables in all functions indicating if the variable is tainted. Carburizer also stores a list of tainted functions that return values calculated from device inputs. The table from the first pass is used by second-pass analyses described below.

3.1.1 Infinite Polling

Drivers often wait for a device to enter a given state by polling a device register. Commonly, the driver sits in a tight loop reading the device register until a bit is set to the proper value, as shown in Figure 2. If the device *never sets* the proper value, this loop will cause the system to hang. Driver developers are expected to ensure these loops will timeout eventually. We find, though, that in many cases device drivers omit the timeout code and loops terminate only if the device functions correctly.

To identify these unbounded loops, we implement an analysis to detect control paths that wait forever for a particular input from the device. Carburizer locates all loops where the terminating conditions are tainted (*i.e.*, dependent on the device). For each loop, Carburizer computes the set of conditions that cause the loop to terminate through `while` clauses as well as conditional `break`, `return` and `goto` statements. If all the terminating conditions for a loop are hardware dependent, the loop may iterate infinitely when the device misbehaves. Figure 2 shows a bug detected by our analysis. The code in lines 5-6 can loop infinitely if `readl`, a function to read a device register, never returns the correct value. While this is a simple example, our analysis can detect complex cases, such as loops that contain `case` statements or that call functions performing I/O.

3.1.2 Checking Array Accesses

Many drivers use inputs from a device to index into an array. When the range of the variable (*e.g.*, 65536 for a short) is larger than the array, an incorrect index can lead to reading an unmapped address (for large indices) or corrupting adjacent data structures. Figure 3 shows a loop in the Pro Audio sound driver (pas2_card.c) that does not check for bounds while accessing an array. While many drivers always check array bounds, some drivers are not as conscientious. Furthermore, a single driver may be inconsistent in its checks.

```
1  static void __init attach_pas_card(...) {
2    .
3    if ((pas_model = pas_read(0xFF88)))
4    {
5      char  temp[100];
6
7      sprintf(temp,
8        "%s rev %d",
9         pas_model_names[(int) pas_model],
10        pas_read(0x2789));
11     .
12 }
```

Figure 3: The Pro Audio Sound driver (pas2_card.c) uses the `pas_model` variable as an array index in line 9 without any checks.

```
1  static void orc_interrupt(...)  {
2    .
3    bScbIdx = ORC_RD(hcsp->HCS_Base,
4                     ORC_RQUEUE);
5
6    pScb = (ORC_SCB *) ((ULONG)
7        hcsp->HCS_virScbArray
8        + (ULONG)
9        (sizeof(ORC_SCB) * bScbIdx));
10
11   pScb->SCB_Status = 0x0;
12
13   inia100SCBPost((BYTE *)
14                  hcsp, (BYTE *) pScb);
15   .
16 }
```

Figure 4: The `pScbIdx` variable is used in pointer arithmetic in line 11 without any check in the a100 SCSI driver (a100u2w.c).

We implement an analysis in Carburizer to determine whether tainted variables are used as array indices in static arrays. If the array is accessed using a tainted variable, Carburizer flags the access as a potential hardware dependence bug. The analysis can detect when values returned by one function are used as array indices in another. In addition, when an array index is computed from multiple variables, Carburizer checks whether all the input variables are untainted.

Carburizer also detects dynamic (variable-sized) array dereferencing with tainted variables. CIL converts all dynamic array accesses into pointer arithmetic and memory dereferencing, so it requires a separate analysis from static arrays (those declared as arrays with a fixed size). In the second analysis pass, Carburizer detects whether a tainted variable is used for pointer arithmetic or as the address of a memory dereference. In both cases, Carburizer detects a potentially unsafe memory reference. We report a bug where the pointer arithmetic is performed rather than where a dereference occurs; this is the location where a bounds check is required, as the offset may not be available when memory is actually dereferenced. If the pointer is never used, this may result in a false positive.

Figure 4 shows driver code where unsafe data from device is used for pointer arithmetic. At line 3, `bScbIdx` is assigned value from the `ORC_RD` macro, which reads a 32-bit value from the device. At line 9, this value is used as an offset for pointer `pScb`. If a single bit of the incoming data is flipped, the pointer dereference in line 11 could cause memory corruption or, if the address is unmapped, a system crash.

```
1  void hptiop_iop_request_callback( ... ) {
2    .
3    p = (struct hpt_iop_cmd __iomem *)req;
4    arg = (struct hi_k *)
5              (readl(&req->context) |
6      ((u64) readl(&req->context_hi32)<<32));
7
8    if (readl(&req->result) == IOP_SUCCESS) {
9      arg->result = HPT_IOCTL_OK;
10   }
11   .
12 }
```

Figure 5: The HighPoint RR3xxx SCSI driver (hptiop.c) reads `arg` from the controller in line 5 and dereferences it as a pointer in line 9.

```
static int amd8111e_read_phy(.......)
{
  .
  unsigned long long delta = (cpu/khz/HZ)*2;
  unsigned long long _start = 0;
  unsigned long long _cur = 0;
  timeout = rdtscll(start) + delta ;

  reg_val = readl(mmio + PHY_ACCESS);
  while (reg_val & PHY_CMD_ACTIVE)  {
    reg_val = readl( mmio + PHY_ACCESS );

    if (_cur < timeout) {
        rdtscll(_cur);
    } else {
      __shadow_recover();
    }
  .
}
```

Figure 6: The code from Figure 2 fixed by Carburizer with a timeout counter.

While rare, drivers may also read a pointer directly from a device. Figure 5 shows an example from a SCSI driver where the driver reads a 64-bit pointer in lines 5 and 6 and dereferences it in line 9. Carburizer also flags this use of pointers as a bug.

3.1.3 *Removing False Positives*

False positives may arise when the driver has a timeout in a loop or validates input that our analysis does not detect. From the suspect loops, Carburizer determines whether the programmer has already implemented a timeout mechanism by looking for the use of a *timeout counter*. A timeout counter is a variable that is (1) either incremented or decremented in the loop, (2) not used as an array index or in pointer arithmetic, and (3) used in a terminating condition for the loop, such as a `while` clause or in an `if` before a `break`, `goto`, or `return` statement. If a loop contains a counter, Carburizer determines that it will not loop infinitely. We also detect the use of the kernel `jiffies` clock as a counter.

False positives for unsafe pointer dereferencing and array indexing may occur if the driver already validates the pointer or index with a comparison to NULL or a shift/-mask operation on the incoming pointer data from the device. Carburizer does not flag a bug when these operations occur between the I/O operation and the pointer arithmetic or pointer dereference.

Carburizer removes false positives that occur when a tainted variable is used multiple times without an intervening I/O operation and when a tainted variable is re-assigned with an untainted value. We keep track of where in the code a variable becomes tainted, and only detect a bug if the pointer dereference or array index occurs after the taint.

We find that the false positive techniques have been helpful. Identifying validity checks and repeated use of a variable reduced the number of detected dynamic-array access bugs from 650 to 150, and the other techniques further reduced it by almost half. For infinite polling, these techniques identified half the results as false positives where the driver correctly broke out of the loop.

3.2 Repairing Sensitive Code

Finding driver bugs alone is valuable, but reliability does not improve until the bug is fixed. After finding a bug, Carburizer in many cases can generate a fix. Repairing sensitive code consists of inserting a test to detect whether a failure occurred and code to handle the failure. To recover, Carburizer inserts code that invokes a generic recovery function capable of resetting the hardware. While repeating a device read operation may fix the bug, this is not safe in general because device-register reads can have side effects. As recovery affects performance, we ensure it will not be invoked unless an unhandled failure occurs and the driver could otherwise crash or hang.

Carburizer relies on a generic recovery function common to all drivers. However, some drivers already implement recovery functionality. For example, the E1000 gigabit Ethernet driver provides a function to shutdown and resume the driver when it detects an error. For such drivers, it may be helpful to modify Carburizer to generate code invoking a driver-specific function instead.

Fixing infinite polling.

When Carburizer identifies a loop where a driver may wait infinitely, it generates code to break out of the loop after a fixed delay. We selected maximum delays based on the delays used in other drivers. For loops that do not sleep, we found that most drivers wait for two timer ticks before timing out; we chose five ticks, a slightly longer delay, to avoid incorrectly breaking out of loops. For loops that invoke a sleep function such as `msleep`, we insert code that breaks out of loops after five seconds, because the delay does not impact the rest of the system. This is far longer than most devices require and ensures that if our analysis does raise false positives, the repair will not break the driver. As shown in Figure 6, for tight loops Carburizer generates code to read the processor timestamp counter before the loop and breaks out of the loop after the specified time delay. When the loop times out, the driver invokes a generic recovery function. This repair will only be invoked after a previously infinite loop times out, ensuring that there will not be any falsely detected failures.

Fixing invalid array indices.

When array bounds are known, Carburizer can insert code to detect invalid array indices with a simple bounds check before the array is accessed. Carburizer computes the size of static arrays and inserts bounds checks on array indices when the index comes from the device. When an array index is used repeatedly, Carburizer only inserts a bounds check before the first use of the tainted array indice.

```
static void __init attach_pas_card(...)
{
  .
  if ((pas_model = pas_read(0xFF88)))
  {
    char  temp[100];

  if ((int )pas_model < 0 ||
      (int )pas_model >= 5) {
      __shadow_recover();
  }
    sprintf(temp,
      "%s rev %d",
       pas_model_names[(int) pas_model],
       pas_read(0x2789));
    .
}
```

Figure 7: The code from Figure 3 fixed by Carburizer with a bounds check.

```
 1 void hptiop_iop_request_callback( ... ) {
 2   .
 3   p = (struct hpt_iop_cmd __iomem *)req;
 4   arg = (struct hi_k *)
 5            (readl(&req->context) |
 6     ((u64) readl(&req->context_hi32)<<32));
 7
 8   if (readl(&req->result) == IOP_SUCCESS) {
 9     if (arg == NULL)
10       __shadow_recover();
11     arg->result = HPT_IOCTL_OK;
12   }
13   .
14 }
```

Figure 8: The code from Figure 5 after repair. Carburizer inserts a null-pointer check in line 9.

For dynamically sized arrays, the bound is not available. Carburizer reports the bug but does not generate a repair. With programmer annotations indicating where array bounds are stored [15, 51], Carburizer could also generate code for dynamic bounds checking.

Figure 7 shows the code from Figure 3 after repair. In this code, the array size is declared statically and Carburizer automatically generates the appropriate range check. This check will only trigger a recovery if the index is outside the array bounds, so it never falsely detects a failure.

When repairing code that reads a pointer directly from a device, Carburizer does not know legal values for the pointer. As result, it only ensures that the pointer is non-NULL. Unlike other fixes, this only prevents a subset of crashes, because legal values of the pointer are not known. Figure 8 shows repaired code where data from device is dereferenced.

Fixing driver panics.

Carburizer can also fix driver code that intentionally crashes the system when hardware fails. Many drivers invoke `panic` when they encounter abnormal hardware situations. While OS vendors discourage this practice, it is used when driver developers do not know how to recover and ensures that errors do not propagate and corrupt the system. If a recovery facility is available then crashing the system is not necessary. Carburizer incorporates a simple analysis to identify calls to `panic`, `BUG`, `ASSERT` and other system halting functions and replace them with calls to the recovery function.

3.3 Summary

The static analysis performed by Carburizer finds many bugs but is neither sound nor complete: it may produce false positives, and identify code as needing a fix when it is in fact correct, and false negatives by missing some bugs. Nonetheless, we find that it identifies many true bugs.

False positives may occur when the driver already contains a validity check that Carburizer does not recognize. For example, if the timeout mechanism for a loop is implemented in a separate function, Carburizer will not find it and will falsely mark the loop as a bug. Carburizer only detects counters implemented as standard integer types. When drivers use custom data-types, Carburizer does not detect the counter and again falsely marks the loop as an error. For array indexing, Carburizer does not consider shift operations as a validity check because, if the array is not a power of two in size, some index values will cause accesses past the end of the array.

False negatives can occur because our interprocedural analysis only passes taint through return values. When a tainted variable is passed as an argument, Carburizer does not detect its use as sensitive code. Carburizer also cannot detect silent failures that occur when the hardware produces a legal but wrong value, such as in incorrect index that lies within the bounds of the array.

3.4 Analysis Results

We ran our code across all drivers in the Linux 2.6.18.8 kernel distribution. In total, we analyzed 6359 source files across the `drivers` and `sound` directories. For major driver classes, Table 2 shows the number of bugs found of each type. Despite analyzing over 2.8 million lines of code, on a 2.4 GHz Core 2 processor the analysis only takes thirty seven minutes to run, output repaired source files and compile the driver files.

The results show that hardware dependence bugs are widespread, with 992 bugs found across various driver classes. Of these, Carburizer can automatically repair the 903 infinite loop and static array index bugs. Only the 89 dynamic-array dereferences require programmer involvement.

We estimate the false positive rate by randomly sampling bugs and inspecting the code. With weighted sampling across all classes of bugs, we compute that Carburizer is able to detect bugs at a false positive rate of 7.4% ± 4.3% with 95% confidence.For the infinite loop bugs, we inspected 140 cases and found only 5 false positives. In these cases, the timeout mechanism was implemented in a function separate from the loop, which Carburizer does not detect. However, Carburizer's timeout was *more relaxed than the driver's, and as a result did not harm the driver.* This low false positive rate demonstrates that a fairly simple and fast analysis can detect infinite loops with high accuracy.

For static arrays, we randomly sampled 15 identified bugs and found 6 true bugs that could cause a system crash if the hardware experienced a transient failure, such as a single bit flip in a device register. Most of the remaining false positives occurred because the array was exactly the size of the index's range, for example 256 entries for an unsigned byte index. However, even in the case of false positives, the code added by Carburizer correctly checked array bounds and does not falsely detect a failure. The only harm done to the driver is the overhead of an unnecessary bounds check. More advanced analysis could remove these false positives.

Driver Class	Infinite Polling Found	Static Array Found	Dynamic Array Found	Panic Fixed
net	117	2	21	2
scsi	298	31	22	121
sound	64	1	0	2
video	174	0	22	22
other	381	9	57	32
Total	860	43	89	179

Table 2: Instances of hardware dependencies by modern Linux device drivers.(2.6.18.8 kernel)

For dynamic arrays and memory dereferencing, we sampled 35 bugs and found 25 real bugs for a programmer to fix. Most false positives manifested in drivers that use mechanisms other than a mask or comparison for verifying an index. For example, the Intel i810_audio driver uses the modulo operation on a dynamic array offset. The SIS graphic driver calls a function to validate all inputs, and Carburizer's analysis cannot detect validation done in a separate function. Better interprocedural analysis is needed to prevent these false positives.

Overall, we found that 498 driver modules out of the 1950 analyzed contained bugs. The bugs followed two distributions. Many drivers had only one or two hardware dependence bugs. The developers of these drivers were typically vigilant about validating device input but forgot in a few places. A small number of drivers performed very little validation and had a large number of bugs. For example, Carburizer detected 24 infinite loops in the telespci ISDN driver and 80 in the ATP 870 SCSI driver.

These bugs demonstrate that language or library constructs can improve the quality of driver code. For example, constructs to wait for a device condition safely, with internally implemented timeouts, reduce the problem of hung systems due to devices. Past work on language support for concurrency in drivers has investigated providing similar language features to avoid correctness violations [8] .

3.5 Experimental Results

We verify that the Carburizer's repair transformation works by testing it on three Ethernet drivers. Testing every driver repair is not practical because it would require obtaining hundreds of devices. We focus on network drivers because we have only implemented the recovery mechanism for this driver class. We test whether carburized drivers, those modified by Carburizer, can detect and recovery from hardware faults.

Of the devices at our disposal, through physical hardware or emulation in a virtual machine, only two 100Mbps network interface cards use drivers that had bugs according to our analysis: a DEC DC21x4x card using the de4x5 driver, and a 3Com 3C905 card using the 3c59x driver. We also tested the forcedeth driver for NVIDIA MCP55 Pro gigabit devices because it places high performance demands on the system (see Section 6). In the case of forcedeth, since there are no bugs in the driver, we emulate problematic code by manually inserting bugs, running Carburizer on the driver, and testing the resulting code.

We inject hardware faults with a simple fault injection tool that modifies the return values of the `read(b,w,l)` and `in(b,w,l)` I/O functions. We modified the forcedeth driver by inserting code that returns incorrect output for a specific device read operation on a device register. We then simulated a series of transient faults in the register of interest. We injected hardware read faults at three locations in the de4x5 driver to induce an infinite-loop in interrupt context. The loop continued even if the hardware returned 0xffffffff, a code used to indicate that the hardware is no longer present in the system. We injected a similar set of faults into the 3c59x driver to create an infinite loop in the interrupt handler and trigger recovery. We did not test all the bugs in each driver, because a single driver may support many devices, and some bugs only occur for a specific device. As a result, we could not force the driver through all buggy code paths with a single device.

In each test, we found that the driver quickly detected the failure with the generated code and triggered the recovery mechanism. After a short delay while the driver recovered, it returned to normal function without interfering with applications. We stopped injecting faults in the de4x5 and 3c59x drivers after they each recovered four times. The forcedeth driver successfully recovered from more than ten of these transient faults. These tests demonstrate that automatic recovery can restart drivers after hardware failures.

4. REPORTING HARDWARE FAILURES

A transient hardware failure, even while recoverable, reduces performance and may portend future failures [31]. As a result, OS and hardware vendors recommend monitoring hardware failures to allow proactive device repair or replacement. For example, the Solaris Fault Management Architecture [40] feeds errors reported by device drivers and other system components into a diagnosis engine. The engine correlates failures from different components and can recommend a high-level action, such as disabling or replacing a device. In reading driver code, we found Linux drivers only report a subset of errors and often omit the failure details.

When Carburizer repairs a hardware dependence bug, it also inserts error-reporting code. Thus, a centralized fault management system can track hardware errors and correlate hardware failures to other system reliability or performance problems. Currently, we use `printk` to write to the system log, as Linux does not have a failure monitoring service.

To support administrative management of hardware failures, Carburizer will also insert monitoring code into existing drivers where the driver itself detects a failure. Carburizer in this case relies on the driver to detect hardware failures, through the timeouts and sanity checks. Figure 9 shows code where the driver detects a failure with a timeout and returns an error, but does not report any failure. In this case, Carburizer will insert logging code where the error is returned and include standard information, such as the driver name, location in the code, and error type (timeout or corruption). If the driver already reports an error, then we assume its report is sufficient and Carburizer does not introduce additional reporting.

```
static int phy_reset(...) {
  .
  while (miicontrol & BMCR_RESET) {
    msleep(10);
    miicontrol = mii_rw(...);
    if (tries++ > 100)
      return -1;
  }
  .
}
```

Figure 9: The forcedeth network driver polls the BMCR_RESET device register until it changes state or until a timeout occurs. The driver reports only a generic error message at a higher level and not the specific failure where it occurred.

We implement analyses in Carburizer to detect when the driver either detects a failure of the hardware or returns an error specifically because of a value read from the hardware. These analyses depend on the bug-finding capabilities from the preceding section to find sensitive code. In this case, what would have been a false positive, because the failure *is* handled by the driver, becomes the condition to search.

4.1 Reporting Device Timeouts

Carburizer detects locations where a driver correctly times out of a polling loop. This code indicates that a device failure has occurred because the device did not output the correct value within the specified time. This analysis is the same as the false-positive analysis used for pruning results for infinite loops, except that the false positives are now the code we seek. Figure 9 shows an example of code that loops until either a timeout is reached or the device produces the necessary value. Carburizer detects whether a logging statement, which we consider a function taking a string as a parameter, occurs either before breaking out of the loop or just after breaking out. If so, Carburizer determines that the driver already reports the failure.

Once loops that timeout are detected, Carburizer identifies the predicate that holds when the loop breaks due of a timeout. Carburizer identifies any return statements based on such predicates and places a reporting statement just before the return. The resulting code is shown in Figure 10. If the test is incorporated into `while` or `for` loop predicate then Carburizer inserts code into the loop to report a failure if the expression holds. CIL converts `for` loops into `while(1)` loops with `break` statements so that code can be inserted between the variable update and the condition evaluation. Thus, the driver will test the expression, report a failure, test the expression again, and break out of the loop.

4.2 Reporting Incorrect Device Outputs

Carburizer analyzes driver code to find driver functions that return errors due to hardware failures. This covers range tests on array indices and explicit comparisons of status or state values. Carburizer identifies that a hardware failure has occurred when the driver returns an error as a result of reading data from a device. Specifically, it identifies code where three conditions hold: (a) a driver function returns a negative integer constant; (b) the error return value is only returned based on the evaluation of a conditional expression, and (c), the expression references variables that were read from the device. We further expand the analysis to detect sites where an error variable is set, such as when the driver sets the return value and jumps to common cleanup code. If these conditions hold, Carburizer inserts a call to the reporting function just before the return statement to signify a hardware failure.

```
static int phy_reset(...) {
  .
  while (miicontrol & BMCR_RESET) {
    msleep(10);
    miicontrol = mii_rw(...);
     if (tries++ > 100) {
      printk("...");
      return -1;
  }
 }
  .
}
```

Figure 10: Carburizer inserts a reporting statement automatically in the case of a timeout, which indicates the device is not operating according to specification.

Driver Class	Device Timeout found/fixed	Incorrect Output found/fixed
net	483/321	249/97
scsi	302/249	137/110
sound	359/297	81/53
other	411/268	361/207
Total	1555/1135	828/467

Table 3: Instances of device-reporting code inserted by Carburizer. Each entry shows the number of device failures detected by the driver, followed by the number where the driver did not report failures and Carburizer inserted reporting code.

4.3 Results

Table 3 shows the result of our analysis. In total, Carburizer identified 1555 locations where drivers detect a timeout. Of these, drivers reported errors only 420 times, and Carburizer inserted error-reporting code 1135 times. Carburizer detected 828 locations where the driver detected a failure with comparisons or range tests. Of these, the driver reported a failure 361 times and Carburizer inserted an error report 467 times.

We evaluate the effectiveness of Carburizer at introducing error-reporting code by performing the same analysis by hand to see whether it finds all the locations where drivers detect a hardware failure. For the drivers listed in Table 4, we identified every location where the original driver detects a failure and whether it reports the failure through logging.

We manually examined the three drivers, one from each major class, and counted as an error any code that clearly indicated the hardware was operating outside of specification. This code performs any of the following actions on the basis of a value read from the device: (1) returning a negative value, (2) printing an error message indicating a hardware failure, or (3) detecting a failed self-test. We did not count errors found in any code removed during preprocessing, such as `ASSERT` statements.

Table 4 shows the number of failures the driver detects (according to our manual analysis), whether reported or not, compared with the number of errors reported by Carburizer.

Driver	Class	Actual errors	Reported Errors
bnx2	net	24	17
mptbase	scsi	28	17
ens1371	sound	10	9

Table 4: Instances of fault-reporting code inserted by Carburizer compared against all errors detected in the driver. Each entry shows the actual number of errors detected in the driver followed by the number of errors reported using Carburizer.

In these three drivers, Carburizer did not produce any false positives: all of the errors reported did indicate a device malfunction. However, Carburizer missed several places where the driver detected a failure. Out of 62 locations where the driver detected a failure, Carburizer identified 43.

We found three reasons for these false negatives. First, some drivers, such as the bnx2 network driver, wrap several low-level read operations in a single function, and return the tainted data via an out parameter. Carburizer does not propagate taint through out parameters. Second, Carburizer's analysis is not sophisticated enough to track tainted structure members across procedure boundaries. The mptbase SCSI driver reads data into a member variable in one procedure and returns an error based on its value in another, and we do not detect the member as tainted where the failure is returned. Finally, some drivers detect a hardware failure and print a message but do not subsequently return an error. Thus, Carburizer does not identify that a hardware failure was detected.

To verify the operation of the reporting statements, we injected targeted faults designed to cause the carburized driver to report a failure. We tested four drivers with fault injection to ensure they reported failures. We injected synthetic faults into the ens1371 sound driver and the de4x5, 8139cp, and 8139too network drivers using the tool from Section 3. We verified that targeted fault injection triggered every reporting statement that applies to these hardware devices.

The only false positive we found occurred in the 8139too network driver during during device initialization. This driver executes a loop that is expected to time out, and Carburizer falsely considers this a hardware fault. The other carburized drivers do not report any false positives. We injected faults with a fixed probability every time the driver invoked a port or I/O memory read operation, both during driver initialization and while running a workload. The drivers did not report any additional errors compared to unmodified drivers under these conditions, largely because none of the injected faults would lead to a system crash. As future work, we plan to examine the problem of reporting if a device is malfunctioning even if the malfunction does not cause a crash.

Overall, we found that Carburizer was effective at introducing additional error logging to drivers where logging did not previously exist. While it does not detect every hardware failure, Carburizer increases the number of failures logged and can therefore improve an administrator's ability to detect when hardware is failing, as compared to driver failures caused by software.

5. RUNTIME FAULT TOLERANCE

The Carburizer runtime provides two key services. First, it provides an automatic recovery service to restore drivers and devices to a functioning state when a failure occurs. Second, it detects classes of failures that cannot be addressed by static analysis and modification of driver code, such as tolerating stuck interrupts.

5.1 Automatic Recovery

Static analysis tools have proved useful as bug finding tools. But, programmers must still write code to repair the bugs that are found. Carburizer circumvents this limitation by relying on *automatic recovery* to restore drivers and devices to a functioning state when a failure is detected. The driver may invoke a recovery function at any time, which will reset the driver to a known-good state. For stuck-at hardware failures, resetting the device can often correct the problem. We rely on the same mechanism to recover from transient failures, although a full reset may not be required in every case.

We leverage shadow drivers [43] to provide automatic recovery because they conceal failures from applications and the OS. A shadow driver is a kernel agent that monitors and stores the state of a driver by intercepting function calls between the driver and the kernel. During normal operation, the shadow driver *taps* all function calls between the driver and the kernel. In this *passive mode*, the shadow driver records operations that change the state of the driver, such as configuration operations and requests currently being processed by the driver.

Shadow drivers are class drivers, in that they are customized to the driver interface but not to its implementation. Thus, a separate shadow driver is needed to recover from failures in each unique class, such as network, sound, or SCSI. We have only implemented recovery for network drivers so far, although other work shows that they work effectively for sound, storage [43] and video drivers [23] .

When the driver invokes the recovery function, the shadow driver transitions into *active mode*, where it performs two functions. First, it proxies for the device driver, fielding requests from the kernel until the driver recovers. This process ensures that the kernel and application software is unaware that the device failed. Second, shadow drivers unload and release the state of the driver and then restart the driver, causing it to reinitialize the device. When starting this driver, the shadow driver uses its log to configure the driver to its state prior to recovery, including resubmitting pending requests. Once this is complete, the shadow driver transitions back to passive mode, and the driver is available for use.

The shadow driver recovery model works when resetting the device clears a device failure. For devices that fail permanently or require a full power cycle to recover, shadow drivers will detect that the failure is not transient when recovery fails and can notify a management agent.

We obtained the shadow driver implementation used for virtual machine migration [22] and ported the recovery functions for network device drivers to the 2.6.18.8 kernel. However, we did not port the entire Nooks driver isolation subsystem [44]. Nooks prevents memory corruption and detects failures through hardware traps, which are unnecessary for tolerating hardware failures. Nooks' isolation also causes a performance drop from switching protection domains, which

Carburizer avoids. The remaining code consists of wrappers around the kernel/driver interface, code to log driver requests, and code to restart and restore driver state after a failure. In addition, we export the `__shadow_recover` function from the kernel, which a driver may call to initiate recovery after a hardware failure.

5.2 Tolerating Missing Interrupts

In addition to providing a recovery service, the Carburizer runtime also detects failures that cannot be detected through static modifications of driver code. Devices may fail by generating too many interrupts or by not generating any. The first case causes a system hang, because no useful work can occur while the interrupt handler is running, while the second case can result in an inoperable device.

To address the scenario in which the device stops generating interrupts, Carburizer monitors the driver and invokes the interrupt handler automatically if necessary. With monitoring, an otherwise operative device need not generate interrupts to provide service. Unlike other hardware errors, we do not force the driver to recover in this case because we cannot detect precisely whether an interrupt is missing. Instead, the Carburizer runtime pro-actively calls the driver's interrupt handler to process any pending requests

The Carburizer runtime increments a counter each time a driver's interrupt handler is called. Periodically, a low priority kernel thread checks this counter. If the counter value has changed, Carburizer does nothing since the device appears to be working normally. If, however, the interrupt handler has not been executed, the device may not be delivering interrupts.

The Carburizer runtime detects whether there has been recent driver activity that should have caused an interrupt by testing whether driver code has been executed. Rather than recording every driver invocation, Carburizer polls the reference bits on the driver's code pages. If any of the code pages have been referenced, Carburizer assumes that a request may have been made and that the interrupt handler should be called soon.

Because every driver is different, Carburizer implements a dynamic approach to increase or decrease the polling interval exponentially, depending on whether previous calls were productive or not. By default, Carburizer checks the referenced bits every 16ms. We chose this value because it provides a relatively good response time in the event of a single missing interrupt. If Carburizer's call to the interrupt handler returns `IRQ_NONE`, indicating the interrupt was spurious, then Carburizer doubles the polling interval, up to a maximum of one second. Conversely, if the interrupt handler returns `IRQ_HANDLED`, indicating that there was work for the driver, then Carburizer decreases the polling interval to a minimum of 4ms. Thus, Carburizer calls the interrupt handler repeatedly only if it detects that the driver is doing useful work during the handler.

Relying on the handler return value to detect whether the handler was productive works for devices that support shared interrupts. Spurious interrupt handler invocations can occur with shared interrupts because the kernel cannot detect which of the devices sharing the interrupt line needs service. However, some drivers report `IRQ_HANDLED` even if the device does not require service, leading Carburizer to falsely detect that it has missed an interrupt. We are examining alternate mechanisms to distinguish productive and unproductive calls to interrupt handlers to improve performance and reduce unnecessary polling, such as timing the duration of the handler or detecting which code pages are accessed during the handler.

Carburizer's polling mechanism adds some overhead when the kernel invokes a driver but does not cause the device to generate an interrupt. For network drivers, this occurs when the kernel invokes an ethtool management function. The Carburizer runtime will call the interrupt handler even though it is not necessary for correct operation. The driver treats this call to its interrupt handler as spurious. Because Carburizer decreases the polling interval in these cases, there is little unnecessary polling even when many requests are made of a driver that do not generate interrupts.

Some Linux network drivers, through the `napi` interface, already support polling. In addition, many network drivers implement a watchdog function to detect when the device stops working. For these drivers, it may be sufficient to direct the kernel to poll rather than relying on a separate mechanism. However, this approach only works for network drivers, while the Carburizer runtime approach works across all driver classes.

5.3 Tolerating Stuck Interrupts

The Carburizer runtime detects stuck interrupts and recovers by converting the device from interrupts to polling by periodically calling the driver's exported interrupt function. A stuck interrupt occurs when the device does not lower the interrupt request line even when directed to do so by the driver. The Carburizer runtime detects this failure when a driver's interrupt handler has been called many times without intervening progress of other system functions, such as the regular timer interrupt. The Linux kernel can detect unhandled interrupts [27], but it recovers by disabling the device rather than enabling it to make progress.

Similar to missing interrupts, the Carburizer runtime does not trigger full recovery here (although that is possible), but instead disables the interrupt request line with `disable_IRQ`. It then relies on the polling mechanism previously described to periodically call the driver's interrupt handler.

5.4 Results

We experiment with stuck and missing interrupts using fault injection on the E1000 gigabit Ethernet driver, the ens1371 sound driver, and a collection of interdependent storage drivers: ide-core, ide-generic, and ide-disk. On all three devices, we simulate missing interrupts by disabling the device's interrupt request line. We simulate stuck interrupts with the E1000 by inserting a command to generate an interrupt from inside the interrupt handler. For E1000, we compare throughput and CPU utilization between an unmodified driver, a driver undergoing monitoring for stuck-/disabled interrupts, and a driver whose interrupt line has been disabled.

In the case of E1000, we found that the Carburizer runtime was able to detect both failures promptly, and that the driver continued running in polling mode. Because interrupts occur only once every 4ms in the steady state, receive throughput drops from 750 Mb/s to 130 Mb/s. With more frequent polling, the throughput would be higher. Similarly, Carburizer detected both failures for the IDE driver. The IDE disk operated correctly in polling mode but throughput decreased by 50%. The ens1371 driver in polling mode

NVIDIA MCP55 Pro gigabit NIC (forcedeth)

System	Throughput	CPU Utilization
Linux 2.6.18.8 Kernel	940 Mb/s	31%
Carburizer Kernel (with shadow driver)	935 Mb/s	36%

Intel Pro/1000 gigabit NIC (E1000)

System	Throughput	CPU Utilization
Native Kernel	721 Mb/s	16%
Carburizer Kernel (with shadow driver)	720 Mb/s	16%

Table 5: TCP streaming send performance with netperf for regular and carburized drivers with automatic recovery mechanism for the E1000 and forcedeth drivers.

Intel Pro/1000 gigabit NIC (E1000)

System	Throughput	CPU %
Native Kernel - TCP	750 Mb/s	19%
Carburizer Monitored - TCP	751 Mb/s	19%
Native Kernel - UDP-RR	7328 Tx/s	6%
Carburizer Monitored - UDP-RR	7310 Tx/s	6%

Table 6: TCP streaming and UDP request-response receive performance comparison of the E1000 between the native Linux kernel and a kernel with the Carburizer runtime monitoring the driver's interrupts.

played back sound with a little distortion, but otherwise operated normally. These tests demonstrate that Carburizer's stuck and missing interrupt detection mechanism works and can keep devices functioning in the presence of a failure.

6. OVERHEAD EVALUATION

The primary cost of using Carburizer is the time spent running the tool and fixing bugs that cannot be automatically repaired. However, the code transformations introduced by Carburizer, shadow driver recovery, and interrupt monitoring introduce a small runtime cost. In this section we measure the overhead of running carburized drivers.

We measure the performance overhead on gigabit Ethernet drivers, as they are the most performance-intensive of our devices: a driver may receive more than 75,000 packets to deliver per second. Thus, any overhead of Carburizer's mechanisms will show up more clearly than on lower-bandwidth devices. Past work on Nooks and shadow storage drivers showed a greater difference in performance than for the network, but the CPU utilization differences were far greater for network drivers [43].

We measure performance with netperf [21] between two Sun Ultra 20 workstation with 2.2Ghz AMD Opteron processors and 1GB of RAM connected via a crossover cable. We configure netperf to run enough experiments to report results accurate to 2.5% with 99% confidence.

Table 5 shows the throughput and CPU utilization for sending TCP data with a native Linux kernel and one with the Carburizer runtime with shadow driver recovery enabled and a carburized network driver. The network throughput with Carburizer is within one-half percent of native performance, and CPU utilization increases only five percentage points for forcedeth and not at all for the E1000 driver. These results demonstrate that supporting the generic recovery service, even for high-throughput devices, has very little runtime cost.

Table 6 shows performance overhead of interrupt monitoring but with no shadow driver recovery. The table shows the TCP receive throughput and CPU utilization for the E1000 driver on the native Linux kernel, and on a kernel with Carburizer interrupt monitoring enabled. The TCP receive and transmit socket buffers were left at their default sizes of 87,380 and 655,360 bytes, respectively. The table also shows UDP request-response performance with 1-byte packets, a test designed to highlight driver latency. While these results are for receiving packets, we also compared performance with TCP and UDP-RR transmit benchmarks and found similar results: the performance of the native kernel and the kernel with monitoring are identical.

These two sets of experiments demonstrate that the cost of tolerating hardware failures in software, either through explicit invocation of a generic recovery service or through run-time interrupt monitoring, is low. Given this low overhead, Carburizer is a practical approach to tolerate even infrequent hardware failures.

7. RELATED WORK

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

Driver reliability.

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 Carburizer, these tools are all heavyweight: they require new operating systems (Singularity [37], Minix [18], Nexus [48]), new driver models (Windows UMDF [29], Linux user-mode drivers [24]), runtime instrumentation of large amounts of code (XFI [46] and SafeDrive [51]), adoption of a hypervisor (Xen [13] and iKernel [45]), or a new subsystem in the kernel (Nooks [44]). Carburizer instead fixes specific bugs, which reduces the code needed in the kernel to just recovery and not fault detection or isolation. Thus, Carburizer may be easier to integrate into existing kernel development processes. Furthermore, Carburizer detects hardware failures before they cause corruption, while driver reliability systems using memory detection 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, 12] 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, 17, 41, 52]. This approach requires a machine with the device installed, while Carburizer 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 [42]. 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 [10, 35, 36]. 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 [28], Dingo [33], HAIL [39], Nexus [48], Laddie [49] 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 [43] for recovery. However, since our 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 [51], and Minix [18], 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 [11]. However, CuriOS is a new operating system and requires specially written code to take advantage of its recovery system, while Carburizer works with existing driver code in existing operating systems.

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, 19, 38]. These systems provide an alternate recovery mechanism to shadow drivers. However, this approach still relies on drivers to detect failures, and Carburizer improves that ability.

8. CONCLUSIONS

System reliability is limited by the reliability of devices. Evidence suggests that device failures cause a measurable fraction of system failures, and that most hardware failures are transient and can be tolerated in software. Carburizer improves reliability by *automatically hardening* drivers against device failures without new programming languages, programming models, operating systems, or execution environments. Carburizer finds and repairs hardware dependence bugs in drivers, where the driver will hang or crash if the hardware fails. In addition, Carburizer inserts logging code so that system administrators can proactively repair or replace hardware that fails.

In an analysis of the Linux kernel, Carburizer identified over 992 hardware dependence bugs with fewer than 8% false postives. Discounting for false positives, Carburizer could automatically repair approximately 845 real bugs by inserting code to detect when a failure occurs and invoke a recovery service. Repairs made to false positives have no correctness impact. In performance tests, hardening drivers had almost no visible performance overhead.

There are still more opportunities to improve device drivers. Carburizer assumes that if a driver detects a hardware failure, it correctly responds to that failure. In practice, we find this is often not the case. In addition, Carburizer does not assist drivers in handling unexpected events; we have seen code that crashes when the device returns a flag before the driver is prepared. Thus, there are yet more opportunities to improve driver quality.

Acknowledgements

This work is supported in part by the National Science Foundation (NSF) grants CCF 0621487 and CNS 0745517, and by the Wisconsin Alumni Research Foundation. We would also like to thank Ben Liblit for helpful discussions during the initial stages of the project and our shepherd Miguel Castro for his useful advice. Swift has a financial interest in Microsoft Corp.

9. 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] 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.

[9] Coverity. Anaylsis of the Linux kernel, 2004. Available at `http://www.coverity.com`.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] N. Ganapathy, 2009. Architect, Microsoft Windows Driver Experience team, personal communication.

[15] 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.

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

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

[18] 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.

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

[20] 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.

[21] R. Jones. Netperf: A network performance benchmark, version 2.1, 1995. Available at `http://www.netperf.org`.

[22] A. Kadav and M. M. Swift. Live migration of direct-access devices. In *First Workshop on I/O Virtualization (WIOV '08)*, Dec. 2008.

[23] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-independent graphics acceleration. In *Proc. of the 3rd VEE*, June 2007.

[24] 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.

[25] 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.

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

[27] Linux Kernel Mailing List. Improve behaviour of spurious irq detect. `http://lkml.org/lkml/2007/6/7/211`, June 2007.

[28] 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.

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

[30] 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.

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

[32] H. Post and W. Kuchlin. Integrated static analysis for Linux device driver verification. In *Proc. of the 6th International Conference on Integrated Formal Methods*, July 2007.

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

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

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

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

[37] 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.

[38] 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.

[39] 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.

[40] Sun Microsystems. Opensolaris community: Fault management. `http://opensolaris.org/os/community/fm/`.

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

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

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

[44] 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.

[45] 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.

[46] 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.

[47] D. Walker, L. Mackey, J. Ligatti, G. A. Reis, and D. I. August. Static typing for a faulty lambda calculus. In *Proc. of the ICFP Conference*, Sept. 2006.

[48] 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.

[49] 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.

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

[51] 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.

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

Understanding Modern Device Drivers

Asim Kadav and Michael M. Swift

Computer Sciences Department, University of Wisconsin-Madison
{kadav, swift} @cs.wisc.edu

Abstract

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, bugs and development costs. 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 beyond the small set of drivers used for research.

In this paper, we study the source code of Linux drivers to understand what drivers actually do, how current research applies to them and what opportunities exist for future research. We determine whether assumptions made by driver research, such as that all drivers belong to a class, are indeed true. We also analyze driver code and abstractions to determine whether drivers can benefit from code re-organization or hardware trends. We develop a set of static-analysis tools to analyze driver code across various axes. Broadly, our study looks at three aspects of driver code (i) what are the characteristics of driver code functionality and how applicable is driver research to *all* drivers, (ii) how do drivers interact with the kernel, devices, and buses, and (iii) are there similarities that can be abstracted into libraries to reduce driver size and complexity?

We find that many assumptions made by driver research do not apply to all drivers. At least 44% of drivers have code that is not captured by a class definition, 28% of drivers support more than one device per driver, and 15% of drivers do significant computation over data. From the driver interactions study, we find that the USB bus offers an efficient bus interface with significant standardized code and coarse-grained access, ideal for executing drivers in isolation. We also find that drivers for different buses and classes have widely varying levels of device interaction, which indicates that the cost of isolation will vary by class. Finally, from our driver similarity study, we find 8% of all driver code is substantially similar to code elsewhere and may be removed with new abstractions or libraries.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Measurement, Design

Keywords Device Drivers, Measurement

1. Introduction

Modern computer systems are communicating with an 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 controllers. Device drivers constitute 70% of the Linux code base [32], and likely are a greater fraction of the code written for the Windows kernel, which supports many more devices. Several studies have shown that drivers are the dominant cause of OS crashes in desktop PCs [14, 28]. As a result, there has been a recent surge of interest in techniques to tolerate faults in drivers [12, 38, 39, 47], to improve the quality of driver code [20]; and in creating new driver architectures that improve reliability and security [4, 15, 21, 23, 24, 31, 44].

However, most research on device drivers focuses on a small subset of devices, typically a network card, sound card, and storage device, all using the PCI bus. These are but a small subset 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 devices studied are fairly mature and have standardized interfaces, but many other devices may have significant functionality differences.

Thus, it is important to study all drivers to review how the driver research solutions being developed are applicable to all classes of drivers. In addition, a better understanding of driver code can lead to new abstractions and interfaces that can reduce driver complexity and improve reliability.

This paper presents a comprehensive study of all the drivers in the Linux kernel in order to broadly characterize their code. We focus on (i) what driver code does, including where driver development work is concentrated, (ii) the interaction of driver code with devices, buses, and the kernel, and (iii) new opportunities for abstracting driver functionality into common libraries or subsystems. We use two static analysis tools to analyze driver code. To understand properties of driver code, we developed DrMiner, which performs data-flow analyses to detect properties of drivers at the granularity of functions. We also developed the DrComp tool, which uses geometric shape analysis [3] to detect similar code across drivers. DrComp maps code to points in coordinate space based on the structure of individual driver functions, and similar functions are at nearby coordinates.

The contributions of this paper are as follows:

- First, we analyze what driver code does in order to verify common assumptions about driver code made by driver research. We show that while these assumptions hold for most drivers, there are a significant number of drivers that violate these assumptions. We also find that several rapidly growing driver classes are not being addressed by driver research.
- Second, we study driver *interactions* with the kernel and devices, to find how existing driver architecture can adapt to a world of multicore processors, devices with high-power processors and virtualized I/O. We find that drivers vary widely

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'12, March 3–7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03$10.00. . . $10.00

by class, and that USB drivers are more efficient in supporting multiple chipsets than PCI drivers. Furthermore, we find that XenBus drivers may provide a path to executing drivers outside the kernel and potentially on the device itself.

- Third, we study driver code *contents* to find opportunities to reduce or simplify driver code. We develop new analysis tools for detecting similar code structures and their types that detect over 8% of Linux driver code is very similar to other driver code, and offer insights on how this code can be reduced.

In the remainder of this paper, we first discuss device driver background and develop a taxonomy of drivers. We then present the three broad classes of results on driver behavior in Sections 3 and 4. In Section 5 we present results showing the extent of repeated code in drivers. Section 6 discusses our findings.

2. Background

A device driver is a software component that provides an interface between the OS and a hardware device. The driver configures and manages the device, and converts requests from the kernel into requests to the hardware. Drivers rely on three interfaces: (i) the interface between the driver and the kernel, for communicating requests and accessing OS services; (ii) the interface between the driver and the device, for executing operations; and (iii) the interface between the driver and the bus, for managing communication with the device.

2.1 Driver/Device Taxonomy

The core operating system kernel interacts with device drivers through a 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. Below these top-level interfaces, support libraries provide common interfaces for many other families of devices, such as keyboards and mice within character drivers.

In order to understand the kinds of drivers Linux supports, we begin by taxonomizing drivers according to their interfaces. 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. We consider all device drivers, bus drivers and virtual drivers that constitute the driver directories (`/sound` and `/drivers`) in the Linux 2.6.37.6 kernel, dated April, 2011. We perform our analyses on all drivers that compile on the x86 platform, using the kernel build option to compile all drivers. Overall, we consider 3,217 distinct drivers. While there are a significant number of Linux drivers that are distributed separately from the kernel, we do not consider them for this work.

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 further classify the classes into sub-categories to understand the range of actual device types supported by them through manual classification, using the device operations they register, the device behavior and their location. While Linux organizes related drivers in directories, this taxonomy is not the same as the Linux directory organization: network drivers are split under *drivers/net*, *drivers/atm* and other directories. However, block drivers are split by their interfaces under *drivers/scsi*, *drivers/ide* and other directories

Figure 1 shows the hierarchy of drivers in Linux according to their interfaces, starting from basic driver types i.e. char, block and net. We identify 72 unique classes of drivers. The majority (52%) of driver code is in character drivers, spread across 41 classes. Network drivers account 25% of driver code, but have only 6 classes. In contrast to the rich diversity of Figure 1, Table 1 lists the driver types used in research. Most driver research (static-analysis tools excepted) neglects the heavy tail of character devices. For example, video and GPU drivers contribute significantly towards driver code (almost 9%) due to complex devices with instruction sets that change each generation, but these devices are largely ignored by driver research due to their complexity.

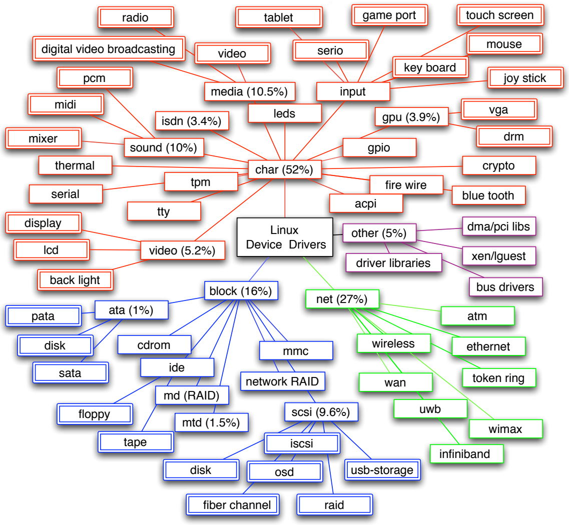


Figure 1. The Linux driver taxonomy in terms of basic driver classes. The size (in percentage of lines of code) is mentioned for 5 biggest classes. Not all driver classes are mentioned.

We also looked at low-level buses, which provide connectivity to devices and discover the devices attached to a system. The majority of devices are either PCI (36% of all device identifiers) or USB (35%), while other buses support far fewer: I2C represents 4% of devices, PCMCIA is 3% and HID is 2.5% (mostly USB devices). The remaining devices were supported by less popular or legacy buses such as ISA or platform devices. We also found that 8% of devices perform low-level I/O without using a bus, interconnect, or support virtual devices. Higher-level protocols such as SCSI (8.5%) and IDE (2%) use one of these underlying low-level buses such as PCI and USB. While PCI drivers still constitute the greatest fraction of drivers, the number of devices supported by USB is similar to PCI. Hence, driver research should validate their performance and reliability claims on USB devices as well.

2.2 Driver Research Assumptions

Most research makes some simplifying assumptions about the problem being solved, and driver research is no different. For example, Shadow Drivers [38] assume that all drivers are members of a class and there are no unique interfaces to a driver. Similarly, the Termite driver-synthesis system assumes that drivers are state machines and perform no computations [33]. Table 2 lists the assumptions made by recent research into device drivers.

We separate these assumptions into two categories: (i) *interactions* refers to assumptions about how drivers interact with the kernel, and (ii) *architecture* refers assumptions about the role of the driver: is it a conduit for data, or does it provide more substantial processing or services? Interaction assumptions relate to how the kernel and driver interact. For example, systems that interpose on driver/device communication, such as Nooks [39], typically assume that communication occurs over procedure calls and not shared memory. Nooks' isolation mechanism will not work otherwise. Similarly, Shadow Drivers assume that the complete state of the device is available in the driver by capturing kernel/driver in-

Improvement type	**System**	**Driver classes tested**	**Drivers tested**
New functionality	Shadow driver migration [19]	net	1
	RevNIC [6]	net	4
Reliability (H/W Protection)	CuriOS [9]	serial port, NOR flash	2
	Nooks [39]	net, sound	6
	Palladium [7]	custom packet filter	1
	Xen [12]	net, sound	2
Reliability (Isolation)	BGI [5]	net, sound, serial, ramdisk, libs	16
	Shadow Drivers [38]	net, sound, IDE	13
	XFI [40]	ram disk, dummy	2
Specification	Devil [25]	scsi, video	2
	Dingo [32]	net	2
	Laddie [45]	net, UART, rtc	3
	Nexus [44]	net, sound, USB-mouse, USB-storage	4
	Termite [33]	net, mmc	2
Static analysis tools	Carburizer [18]	All/net	All/3
	Cocinelle [29]	Various	All
	SDV [2]	basic ports, storage, USB, 1394-interface, mouse, keyboard, PCI battery	126
Type safety	Decaf Drivers [31]	net, sound, USB controller, mouse	5
	Safedrive [47]	net, USB, sound, video	6
	Singularity [37]	Disk	1
User-level device drivers	Micro-drivers [15]	net, sound, USB controller	4
	SUD [4]	net, wireless, sound, USB controllers, USB	6/1
	User-level drivers [21]	net, disk (ata)	2

Table 1. Research projects on drivers, the improvement type, and the number and class of drivers used to support the claim end to end. Few projects test all driver interfaces thoroughly. Static analysis tools that do not require driver modifications are available to check many more drivers. Also, some solutions, like Carburizer [18], and SUD [4] support the performance claims on fewer drivers.

Driver interactions
Class membership: Drivers belong to common set of classes, and the class completely determines their behavior.
Procedure calls: Drivers always communicate with the kernel through procedure calls.
Driver state: The state of the device is completely captured by the driver.
Device state: Drivers may assume that devices are in the correct state.
Driver architecture
I/O: Driver code functionality is only dedicated to converting requests from the kernel to the device.
Chipsets: Drivers typically support one or a few device chipsets.
CPU Bound: Drivers do little processing and mostly act as a library for binding different interfaces together.
Event-driven: Drivers execute only in response to kernel and device requests, and to not have their own threads.

Table 2. Common assumptions made in device driver research.

teractions [38]. However, network cards that do TCP-offload may have significant protocol state that is only available in the device, and cannot be captured by monitoring the kernel/driver interface.

Several recent projects assume that drivers support a single chipset, such as efforts at synthesizing drivers from a formal specification [33]. 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 [44] may be cumbersome if many chipsets must be supported for each driver. Other efforts at reverse engineering drivers [6] similarly may be complicated by the support of many chipsets with different hardware interfaces. Furthermore, these synthesis and verification systems assume that devices always behave correctly, and their drivers may fail unpredictably with faulty hardware.

Another assumption made by driver research is that drivers are largely a conduit for communicating data and for signaling the device, and that they perform little processing. Neither RevNIC [6]) nor Termite [33] 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 real generality. If these assumptions are true for all drivers, then these research ideas have broad applicability. If not, then new research is needed to address the outliers.

3. What Do Drivers Do?

Device drivers are commonly assumed to primarily perform I/O. A standard undergraduate OS textbook states:

> "A device driver can be thought of a translator. Its input consists of high-level commands such as "retrieve block 123." Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system."
>
> *Operating Systems Concepts* [36]

However, this passage describes the functions of only the small portion of driver code that which actually performs I/O. Past work revealed that the bulk of driver code is dedicated to initialization and configuration for a sample of network, SCSI and sound drivers [15].

We seek to develop a comprehensive understanding of what driver code does: what are the tasks drivers perform, what are the interfaces they use, and how much code does this all take. 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.

3.1 Methodology

To study the driver code, we developed the *DrMiner* static analysis tool using CIL [27] to detect code properties in individual drivers. DrMiner takes as input unmodified drivers 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. For example, we find the different devices and chipsets supported by analyzing the `device_id` structures registered (*e.g.*, `pci_device_id`, `acpi_device_id` etc.) with the kernel. We also identify the driver entry points from the driver structure registered (*e.g.*, `pci_driver`, `pnp_device`) and the device operations structure registered (*e.g.*, `net_device`). DrMiner analyzes the function pointers registered by the driver to determine the functionality of each driver. We then construct a control-flow graph of the driver that allows us to determine all the functions reachable through each entry point, including through function pointers.

We use a *tagging* approach to labeling driver code: DrMiner 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. In addition, DrMiner identifies specific code features, such as loops indicating computation.

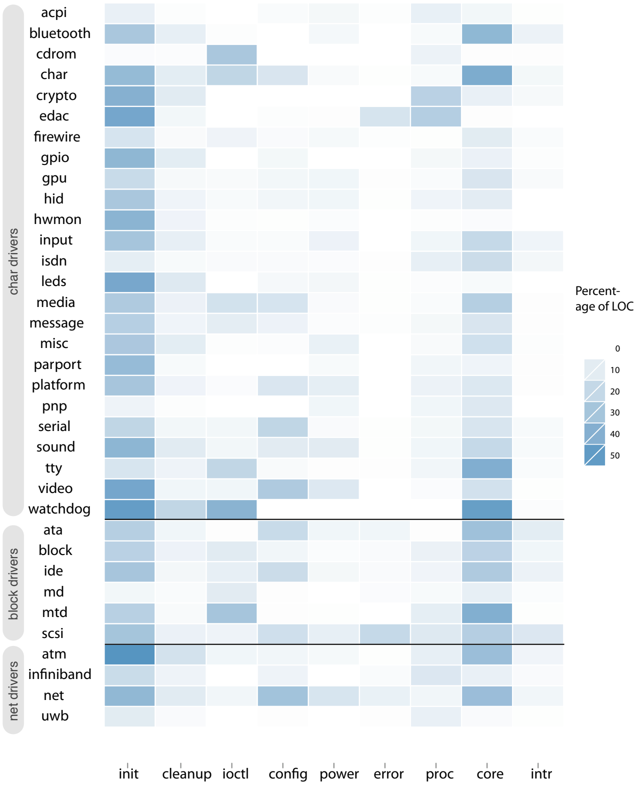


Figure 2. The percentage of driver code accessed during different driver activities across driver classes.

We run these analyses over the entire Linux driver source and store the output in a SQL database. The database stores information about each driver as well as each function in the driver. The information about the driver consists of name, path, size, class, number of chipsets, module parameters, and interfaces registered with the kernel. The information about each driver function consists of function name, size, labels, resources allocated (memory, locks etc.), and how it interacts with the kernel and the device. From the database, determining the amount of code dedicated to any function is a simple query. In our results, we present data for about 25 classes with the most code.

3.2 What is the function breakdown of driver code?

Drivers vary widely in how much code they use for different purposes; a simple driver for a single chipset may devote most of its code to processing I/O requests and have a simple initialization routine. In contrast, a complex driver supporting dozens of chipsets may have more code devoted to initialization and error handling than to request handling.

Figure 2 shows the breakdown of driver code across driver classes. The figure shows the fraction of driver code invoked during driver initialization, cleanup, ioctl processing, configuration, power management, error handling, `/proc` and `/sys` handling, and most importantly, core I/O request handling (*e.g.*, sending packet for network devices, or playing audio for sound card) and interrupt handling across different driver classes.

The largest contributors to driver code are initialization and cleanup, comprising almost 36% of driver code on average, error handling (5%), configuration (15%), power management (7.4%) and ioctl handling (6.2%). On average, only 23.3% of the code in a driver is dedicated to request handling and interrupts.

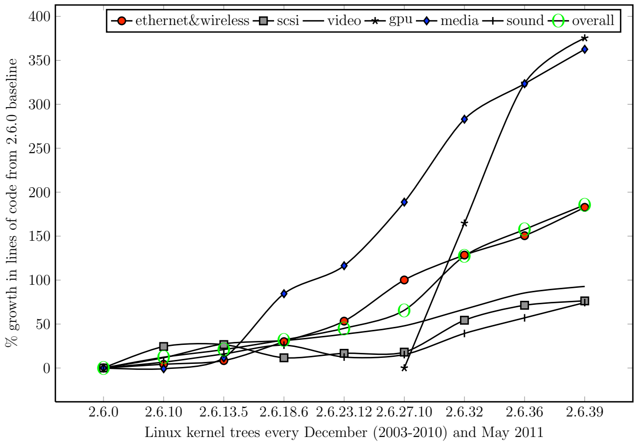


Figure 3. The change in driver code in terms of LOC across different driver classes between the Linux 2.6.0 and Linux 2.6.39 kernel.

Implications: These results indicate that efforts at reducing the complexity of drivers should not only focus on request handling, which accounts for only one fourth of the total code, but on better mechanisms for initialization and configuration. For example, as devices become increasingly virtualization aware, quick ways to initialize or reset are critical for important virtualization features such as re-assignment of devices and live migration [19]. Drivers contain significant configuration code (15%), specifically in network (31%) and video (27%) drivers. As devices continue to become more complex, driver and OS research should look at efficient and organized ways of managing device configuration [35].

3.3 Where is the driver code changing?

Over time, the focus of driver development shifts as new device classes become popular. We compared the breakdown of driver code between the 2.6.0 and 2.6.39 for new source lines of code added annually to different driver classes. We obtain the source lines of code across different classes in 9 intermediate revisions (every December since 2.6.0) using sloccount [43].

Figure 3 shows the growth in driver across successive years from the 2.6.0 baseline for 8 major driver classes. Overall, driver code has increased by 185% over the last eight years. We identify three specific trends in this growth. First, there is additional code for new hardware. This code includes wimax, GPU, media, input devices and virtualization drivers. Second, there is increasing support for certain class of devices, including network (driven by wireless), media, GPU and SCSI. From 2.6.13 to 2.6.18, the devices supported by a vendor (QLogic) increased significantly. Since, they were very large multi-file SCSI drivers, the drivers where coalesced to a single driver, reducing the size of SCSI drivers in the driver tree. In Section 5, we investigate whether there are opportunities to reduce driver code in the existing code base. Third, there is minor code refactoring. For example, periodically, driver code is moved away from the driver or bus library code into the respective classes where they belong. For example, drivers from the i2c bus directory were moved to misc directory.

Implications: While Ethernet and sound, the common driver classes for research, are important, research should look further into other rapidly changing drivers, such as media, GPU and wireless drivers.

```
drivers/ide/ide-cd.c:
static int cdrom_read_tocentry(...) {
  // Read table of contents data
  for (i = 0; i <= ntracks; i++) {
    if (drive->atapi_flags &
        IDE_AFLAG_TOCADDR_AS_BCD) {
      if (drive->atapi_flags &
          IDE_AFLAG_TOCTRACKS_AS_BCD)
        toc->ent[i].track =
          bcd2bin(toc->ent[i].track);
      msf_from_bcd(&toc->ent[i].addr.msf);}
    toc->ent[i].addr.lba =
      msf_to_lba(toc->ent[i].addr.msf.minute,
                 toc->ent[i].addr.msf.second,
                 toc->ent[i].addr.msf.frame);
  }
}
```

Figure 4. The IDE CD-ROM driver processes table-of-contents entries into a native format.

3.4 Do drivers belong to classes?

Many driver research projects assume that drivers belong to a class. For example, Shadow Drivers [38] must be coded with the semantics of all calls into the driver so it can replay them during recovery. However, many drivers support proprietary extensions to the class interface. In Linux drivers, these manifest as private `ioctl` options, `/proc` or `/sys` entries, and as load-time parameters. If a driver has one of these features, it may have additional behaviors not captured by the class.

We use DrMiner to identify drivers that have behavior outside the class by looking for load-time parameters and code to register `/proc` or `/sys` entries. We do not identify unique `ioctl` options. Overall, we find that most driver classes have substantial amounts of device-specific functionality. Code supporting `/proc` and `/sys` is present in 16.3% of drivers. Also, 36% of drivers have load-time parameters to control their behavior and configure options not available through the class interface. Overall, 44% of drivers use atleast one of the two non-class features. Additionally, `ioctl` code comprises 6.2% of driver code, can also cause non-class behavior.

As an example of how these class extensions are used, the e1000 gigabit network driver has 15 load-time parameters that allow control over interrupt processing and transmit/receive ring sizing, and interrupt throttling rate. This feature is not part of any standard network device interface and is instead specific to this device. Similarly, the i915 DRM GPU driver supports load parameters for down-clocking the GPU, altering graphic responsibilities from X.org to the kernel, and power saving. These parameters change the code path of the driver during initialization as well as during regular driver operations. While the introduction of these parameters does not affect the isolation properties of the reliability solutions, as the interfaces for setting and retrieving these options are standard, it limits the ability to restart and restore the driver to a previous state since the semantics of these options are not standardized.

Implications: While most driver functionality falls into the class behavior, many drivers have significant extensions that do not. Attempts to recover driver state based solely on the class interface [38] or to synthesize drivers from common descriptions of the class [6, 33] may not work for a substantial number of drivers. Thus, future research should explicitly consider how to accommodate unique behaviors efficiently.

3.5 Do drivers do significant processing?

As devices become more powerful and feature 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 processing, for example to compute parity for RAID, checksums for networking, or display data for video drivers, then processing power must be reserved. Furthermore, in a virtualized setting, heavy I/O from one guest VM could substantially reduce CPU availability for other guest VMs.

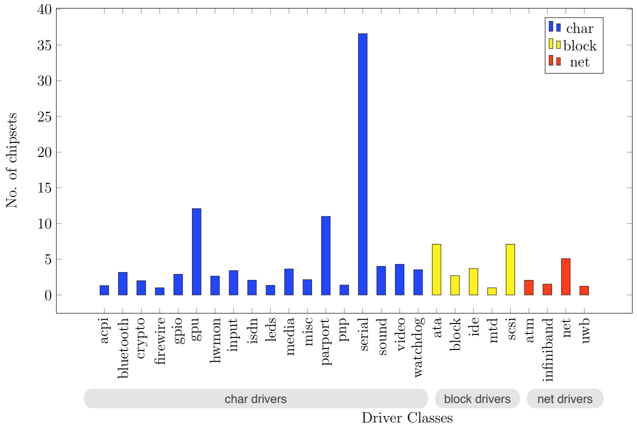


Figure 5. The average number of chipsets supported by drivers in each class.

DrMiner detects processing in drivers by looking for loops that (i) do no I/O, (ii) do not interact with the kernel, and (iii) are on core data paths, such as sending/receiving packets or reading/writing data. This ensures that polling loops, common in many drivers, are not identified as performing processing.

We find that 15% of drivers have at least one function that performs processing, and that processing occurs in 1% of all driver functions. An even higher fraction (28%) of sound and network drivers do processing. Wireless drivers, such as ATH, perform processing to interpolate power levels of the device under different frequencies and other conditions. Many network drivers provide the option of computing checksums on the outgoing/incoming packets. Finally, even CD-ROM drivers, which largely read data off the device, do computation to analyze the table of content information for CD-ROMs, as shown in Figure 4.

Implications: A substantial fraction of drivers do some form of data processing. Thus, efforts to generate driver code automatically must include mechanisms for data processing, not just converting requests from the OS into requests to the device. Furthermore, virtualized systems should account for the CPU time spent processing data when this processing is performed on behalf of a guest VM. These results also point to new opportunities for driver and device design: given the low cost of embedded processors, can all the computation be offloaded to the device, and is there a performance or power benefit to doing so?

3.6 How many device chipsets does a single driver support?

Several driver research projects require or generate code for a specific device chipset. For example, Nexus requires a safety specification that is unique to each device interface [44]. If a driver supports only a single device, this requirement may not impose much burden. However, if a driver supports many devices, each with a different interface or behavior, then many specifications are needed to fully protect a driver.

We measure the number of chipsets or hardware packagings supported by each Linux driver by counting the number of PCI, USB or other bus device IDs (*i.e.*, i2c, ieee1394) that the driver recognizes. These structures are used across buses to identify (and match) different devices or packagings that are supported by the

```
static int __devinit cy_pci_probe(...)
{
 if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo) {
     ...
 if (pci_resource_flags(pdev,2)&IORESOURCE_IO){
      ..
 if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo ||
      device_id == PCI_DEVICE_ID_CYCLOM_Y_Hi) {
        ..
}else if (device_id==PCI_DEVICE_ID_CYCLOM_Z_Hi)
 ....
 if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo ||
     device_id == PCI_DEVICE_ID_CYCLOM_Y_Hi) {
  switch (plx_ver) {
   case PLX_9050:
   ...
  default:    /* Old boards, use PLX_9060 */
   ...
}
```

Figure 6. The cyclades character drivers supports eight chipsets that behaves differently at each phase of execution. This makes driver code space efficient but extremely complex to understand.

driver. Figure 5 shows the average number of chipsets supported by each driver in each driver class. While most drivers support only a few different devices, serial drivers support almost 36 chipsets on average, and network drivers average 5. The Radeon DRM driver supports over 400 chipsets, although many of these may indicate different packagings of the same internal chipset. Generic USB drivers such as usb-storage and usb-audio support over 200 chipsets each, and the usb-serial driver supports more than 500 chipsets. While not every chipset requires different treatment by the driver, many do. For example, the `3c59x` 100-megabit Ethernet driver supports 37 chipsets, 17 sets of features that vary between chipsets, and two complete implementations of the core send/receive functionality. Overall, we find that 28% of drivers support more than one chipset and these drivers support 83% of the total devices.

In order to measure the effects of number of chipsets on driver code size, we measured the least-square correlation coefficient between the number of chipsets support by a driver and the amount of code in the driver and found them to be weakly correlated (0.25), indicating that drivers supporting more chipsets were on average larger than those that did not. However, this does not completely explain the amount of initialization code, as the correlation between the number of chipsets and the percentage of initialization code was 0.07, indicating that the additional chipsets increased the amount of code throughout the driver.

Implications: These results indicate that Linux drivers support multiple chipsets per driver and are relatively efficient, supporting 14,070 devices with 3,217 device and bus drivers, for an average of approximately 400 lines of code per device. Any system that generates unique drivers for every chipset or requires per-chipset manual specification may lead to a great expansion in driver code and complexity. Furthermore, there is substantial complexity in supporting multiple chipsets, as seen in Figure 6, so better programming methodologies, such as object-oriented programming [31] and automatic interface generation, similar to Devil [25], should be investigated.

3.7 Discussion

The results in this section indicate that while common assumptions about drivers are generally true, given the wide diversity of drivers, one cannot assume they always hold. Specifically, many drivers contain substantial amounts of code that make some of the existing research such as automatic generation of drivers difficult, due to code unique to that driver and not part of the class, code that processes data, and code for many chip sets.

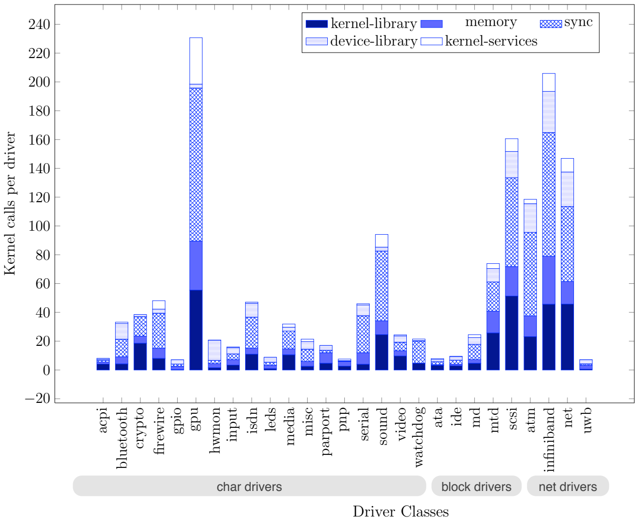


Figure 7. The average kernel library, memory, synchronization, kernel device library, and kernel services library calls per driver (bottom to top in figure) for all entry points.

4. Driver Interactions

The preceding section focused on the *function* of driver code, and here we turn to the *interactions* of driver code: how do drivers use the kernel, and how do drivers communicate with devices? We see three reasons to study these interactions. First, extra processing power on devices or extra cores on the host CPU provide an opportunity to redesign the driver architecture for improved reliability and performance. For example, it may be possible to move many driver functions out of the kernel and onto the device itself. Or, in virtualized systems, driver functionality may execute on a different core in a different virtual machine. Second, much of the difficulty in moving drivers between operating systems comes from the driver/kernel interface, so investigating what drivers request of the kernel can aid in designing more portable drivers. Third, the cost of isolation and reliability are proportional to the size of the interface and the frequency of interactions, so understanding the interface can lead to more efficient fault-tolerance mechanisms.

We examine the patterns of interaction between the driver, the kernel and the device, with a focus on (i) which kernel resources drivers consume, (ii) how and when drivers interact with devices, (iii) the differences in driver structure across different I/O buses, and (iv) the threading/synchronization model used by driver code.

4.1 Methodology

We apply the DrMiner tool from Section 3 to perform this analysis. However, rather than propagating labels down the call graph from entry points to leaf functions, here we start at the bottom with kernel and device interactions. Using a list of known kernel functions, bus functions, and I/O functions, we label driver functions according to the services or I/O they invoke. Additionally, we compute the number of invocations of bus, device and kernel invocations for each function in a driver. These call counts are also propagated to determine how many such static calls could be invoked when a particular driver entry point is invoked.

4.2 Driver/Kernel Interaction

Drivers vary widely in how they use kernel resources, such as memory, locks, and timers. Here, we investigate how drivers use these resources. We classify all kernel functions into one of five categories:

1. Kernel library (*e.g.*, generic support routines such as reporting functions,[1] timers, string manipulation, checksums, standard data structures)
2. Memory management (*e.g.*, allocation)
3. Synchronization (*e.g.*, locks)
4. Device library (*e.g.*, subsystem libraries supporting a class of device and other I/O related functions)
5. Kernel services (*e.g.*, access to other subsystems including files, memory, scheduling)

The first three are generic library routines that have little interaction with other kernel services, and could be re-implemented in other execution contexts. The fourth category, device library, provides I/O routines supporting the driver but does not rely other kernel services, and is very OS dependent. The final category provides access to other kernel subsystems, and is also OS dependent.

Figure 7 shows, for each class of drivers, the total number of function calls made by drivers in every class. The results demonstrate several interesting features of drivers. First, the majority of kernel invocations are for kernel library routines, memory management and synchronization. These functions are primarily *local* to a driver, in that they do not require interaction with other kernel services. Thus, a driver executing in a separate execution context, such as in user mode or a separate virtual machine, need not call into the kernel for these services. There are very few calls into kernel services, as drivers rarely interact with the rest of the kernel.

The number of calls into device-library code varies widely across different classes and illustrates the abstraction level of the devices: those with richer library support, such as network and SCSI drivers, have a substantial number of calls into device libraries, while drivers with less library support, such as GPU drivers, primarily invoke more generic kernel routines.

Finally, a number of drivers make very little use of kernel services, such as ATA, IDE, ACPI, and UWB drivers. This approach demonstrates another method for abstracting driver functionality when there is little variation across drivers: rather than having a driver that invokes support library routines, these drivers are themselves a small set of device-specific routines called from a much larger common driver. This design is termed a “miniport” driver in Windows. Thus, these drivers benefit from a common implementation of most driver functionality, and only the code differences are implemented in the device-specific code. These drivers are often quite small and have little code that is not device specific.

These results demonstrate a variety of interaction styles between drivers and the kernel: drivers with little supporting infrastructure demonstrate frequent interactions with the kernel for access to kernel services but few calls to device support code. Drivers with a high level of abstraction demonstrate few calls to the kernel over all. Drivers with a support library demonstrate frequent calls to kernel generic routines as well as calls to device support routines.

Implications: Drivers with few calls into device libraries may have low levels of abstraction, and thus are candidates for extracting common functionality. Similarly, drivers with many kernel interactions and device library interaction may benefit from converting to a layered on “miniport” architecture, where more driver functionality is extracted into a common library.

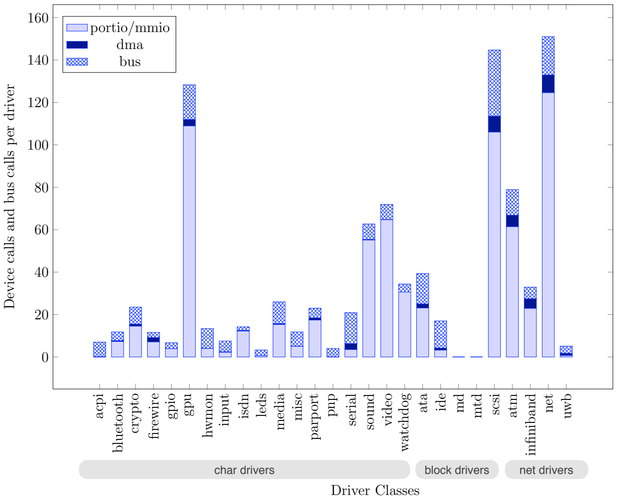


Figure 8. The device interaction pattern representing port I/O, memory mapped I/O, bus resources (bottom to top) invoked via all driver entry points.

Furthermore, a large fraction of driver/kernel interactions are for generic routines (memory, synchronization, libraries) that do not involve other kernel services. Thus, they could be implemented by a runtime environment local to the driver. For example, a driver executing in a separate virtual machine or on the device itself can make use of its local OS for these routines, and drivers in user space can similarly invoke user-space versions of these routines, such as the UML environment in SUD [4].

4.3 Driver/Device Interaction

We next look at interaction between the driver and the device. We analyzed all functions in all drivers, and if a function does I/O itself, or calls a function that results in an I/O, we label it as *perform I/O*. We categorize driver/device interactions around the type of interaction: access to memory-mapped I/O (MMIO) regions or x86 I/O ports (port IO) are labeled *mmio/portio*, *DMA* is DMA access, either initiated by the driver or enabled by the driver creating a DMA mapping for a memory region, and calls to a bus, such as USB or PCI (*bus*). We could not determine statically when a device initiates DMA, although we do count calls to map a page for future DMA (*e.g.*, `pci_map_single`) as a DMA action. Our analysis can detect memory mapped I/O through accessor routines such as `read/writeX` family, `ioread/iowrite` family of routines and port I/O using the `in/outX` family. DrMiner cannot identify direct dereferences of pointers into memory-mapped address ranges. However, direct dereference of I/O addresses is strongly discouraged and most non-conforming drivers have been converted to use accessor routines instead. We also note that all I/O routines on x86 eventually map down to either port or MMIO. Here, though, we focus on the I/O abstractions used by the driver.

Figure 8 shows, for each class of device, the number of device interactions in the entire driver. The results demonstrate that driver classes vary widely in their use of different I/O mechanisms. IDE and ATA drivers, both block devices, show very different patterns of interaction: IDE drivers do very little port or MMIO, because they rely on the PCI configuration space for accessing device registers. Hence, they show a greater proportion of bus operations. Additionally, virtual device classes such as md (RAID), do page-level writes by calling the block subsystem through routines like `submit_bio` rather than by accessing a device directly.

[1] We leave out `printk` to avoid skewing the numbers from calls to it.

	Kernel Interactions					Device Interactions			
BUS	*mem*	*sync*	*dev lib.*	*kern lib.*	*kern services*	*port/mmio*	*dma*	*bus*	*avg devices/driver*
PCI	15.6	57.8	13.3	43.2	9.1	125.1	7.0	21.6	7.5
USB	9.6	25.5	5.6	9.9	3.0	0.0	2.2[2]	13.8	13.2
Xen	10.3	8.0	7.0	6.0	2.75	0.0	0.0	34.0	1/All

Table 3. Comparison of modern buses on drivers across all classes. Xen and USB drivers invoke the bus for the driver while PCI drivers invoke the device directly.

Second, these results demonstrate that the cost of isolating drivers can vary widely based on their interaction style. Direct interactions, such as through ports or MMIO, can use hardware protection, such as virtual memory. Thus, an isolated driver can be allowed to access the device directly. In contrast, calls to set up DMA or use bus routines rely on software isolation, and need to cross protection domains. Thus, drivers using higher-level buses, like USB, can be less efficient to isolate, as they can incur a protection-domain transition to access the device. However, as we show in the next section, access devices through a bus can often result in far fewer operations.

Implications: The number and type of device interactions vary widely across devices. Thus, the cost of isolating drivers, or verifying that their I/O requests are correct (as in Nexus [44]) can vary widely across drivers. Thus, any system that interposes or protects the driver/device interaction must consider the variety of interaction styles. Similarly, symbolic execution frameworks for drivers [20] must generate appropriate symbolic data for each interaction style.

4.4 Driver/Bus Interaction

The plurality of drivers in Linux are for devices that attach to some kind of PCI bus (*e.g.,* PCIe or PCI-X). However, several other buses are in common use: the USB bus for removable devices and XenBus for virtual devices [46]. Architecturally, USB and Xen drivers appear to have advantages, as they interact with devices over a message-passing interface. With USB 3.0 supporting speeds up to 5 Gbps [42] and Xen supporting 10 Gbps networks [30], it is possible that more devices will be accessed via USB or XenBus.

In this section, we study the structural properties of drivers for different buses to identify specific differences between the buses. We also look for indications that drivers for a bus may have better architectural characteristics, such as efficiency or support for isolation. We focus on two questions: (i) does the bus support a variety of devices efficiently, (ii) will it support new software architectures that move driver functionality out of the kernel onto a device or into a separate virtual machine? Higher efficiency of a bus interface results from supporting greater number of devices with standardized code. Greater isolation results from having less device/driver specific code in the kernel. If a bus only executes standardized code in the kernel, then it would be easier to isolate drivers away from kernel, and execute them inside a separate virtual machine or on the device itself such as on an embedded processor.

Table 3 compares complexity metrics across all device classes for PCI, USB, and XenBus. First, we look at the efficiency of supporting multiple devices by comparing the number of chipsets supporting by a driver. This indicates the complexity of supporting a new device, and the level of abstraction of drivers. A driver that supports many chipsets from different vendors indicates a standardized interface with a high level of common functionality. In contrast, drivers that support a single chipset indicate less efficiency, as each device requires a separate driver.

The efficiency of drivers varied widely across the three buses. PCI drivers support 7.5 chipsets per driver, almost always from the same vendor. In contrast, USB drivers average 13.2, often from many vendors. A large part of the difference is the effort at standardization of USB protocols, which does not exist for many PCI devices. For example, USB storage devices implement a standard interface [41]. Thus, the main USB storage driver code is largely common, but includes call-outs to device-specific code. This code includes device-specific initialization, suspend/resume (not provided by USB-storage and left as an additional feature requirement) and other routines that require device-specific code. While there are greater standardization efforts for USB drivers, it is still not complete

Unlike PCI and USB drivers, XenBus drivers do not access devices directly, but communicate with a back-end driver executing in a separate virtual machine that uses normal Linux kernel interfaces to talk to any driver in the class. Thus, a single XenBus driver logically supports *all* drivers in the class. in a separate domain so we report them as a single chipset. However, device-specific behavior, described above in Section 3.4, is not available over XenBus; these features must be accessed from the domain hosting the real driver. XenBus forms an interesting basis for comparison because it provides the minimum functionality to support a class of devices, with none of the details specific to the device. Thus, it represents a "best-case" driver.

We investigate the ability of a bus to support new driver architectures through its interaction with the kernel and device. A driver with few kernel interactions may run more easily in other execution environments, such as on the device itself. Similarly, a driver with few device or bus interactions may support accessing devices over other communication channels, such as network attached devices [26]. We find that PCI drivers interact heavily with the kernel unless kernel resources are provided by an additional higher-level virtual bus (*e.g.,* ATA). In contrast, Xen drivers have little kernel interaction, averaging only 34 call sites compared to 139 for PCI drivers. A large portion of the difference is that Xen drivers need little initialization or error handling, so they primarily consist of core I/O request handling.

The driver/device interactions also vary widely across buses: due to the fine granularity offered by PCI (individual bytes of memory), PCI drivers average more device interactions (154) than USB or XenBus devices (14-34). Thus, USB drivers are more economical in their interactions, as they batch many operations into a single request packet. XenBus drivers are even more economical, as they need fewer bus requests during initialization and as many operations as USB for I/O requests. Thus, USB and XenBus drivers may efficiently support architectures that access drivers over a network, because access is less frequent and coarse grained.

Implications: These results demonstrate that the flexibility and performance of PCI devices comes with a cost: increased driver complexity, and less interface standardization. Thus, for devices that can live within the performance limitations of USB or in a virtualized environment for XenBus, these buses offer real architectural advantages to the drivers. With USB, significant standardization enables less unique code per device, and coarse-grained access allows efficient remote access to devices [1, 10, 17].

[2] USB drivers invoke DMA via the bus.

XenBus drivers push standardization further by removing *all* device-specific code from the driver and executing it elsewhere. For example, it may be possible to use XenBus drivers to access a driver running on the device itself rather than in a separate virtual machine; this could in effect remove many drivers from the kernel and host processor.

The mechanism for supporting non-standard functionality also differs across these buses: for PCI, a vendor may write a new driver for the device to expose its unique features. For USB, a vendor can add functionality to the existing common drivers just for the features. For XenBus, the features must be accessed from the domain executing the driver and are not available to a guest OS.

4.5 Driver Concurrency

Another key requirement of drivers in all modern operating systems is the need to multiplex access to the device. For example, a disk controller driver must allow multiple applications to read and write data at the same time, even if these applications are not otherwise related. This requirement can complicate driver design, as it increases the need for synchronization among multiple independent threads. We investigate how drivers multiplex access across long-latency operations: 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. If drivers are moved outside the kernel, the driver and kernel will communicate with each other using a communication channel and supporting event-driven concurrency may be more natural.

We determine that a driver entry point requires a threaded programming style if it makes blocking calls into the kernel, or busy-waits for a device response using `msleep()` which enables blocking. All other entry points are considered "event friendly", in that they do not suspend the calling thread. We did not detect specific routines that use event-based synchronization, as they often rely on the device to generate the callback via an interrupt rather than explicitly registering with the kernel for a callback .

The results, shown in Figure 9 in the bars labeled *event friendly* and *threaded*, show that the split of threaded and event-friendly code varies widely across driver classes. Overall, drivers extensively use both methods of synchronization for different purposes. Drivers use threaded primitives to synchronize driver and device operations while initializing the driver, and updating driver global data structures, while event-friendly code is used for core I/O requests. Interestingly, network drivers and block drivers, which are not invoked directly by user-level code, have a similar split of code to sound drivers, which are invoked directly from application threads. This arises because of the function of most driver code, as reported in Section 3.2: initialization and configuration. This code executes on threads, often blocking for long-latency initialization operations such as device self-test.

Implications: Threaded code is difficult to run outside the kernel, where the invoking thread is not available. For example, Microdrivers [16] executes all driver code in an event-like fashion, restricting invocation to a single request at a time. Converting drivers from threads to use event-based synchronization internally would simplify such code. Furthermore, events are a more natural fit when executing driver code either in separate virtual machine or on a device itself, as they naturally map to a stream of requests arising over a communication channel [34].

5. Driver Redundancy

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 IDE devices, related devices share most of the code with a small amount of per-device code. Most device classes, though, replicate functionality for every driver. The problem of writing repeated/redundant code is well documented. It causes maintainability issues in software development [13], and is also a significant cause of bugs in the Linux kernel [8, 22, 29]. Providing the right abstractions also helps in code standardization and integrating kernel services such as power management in a correct fashion across all drivers. Without a global view of drivers, it can be difficult to tell whether there are opportunities to share common code.

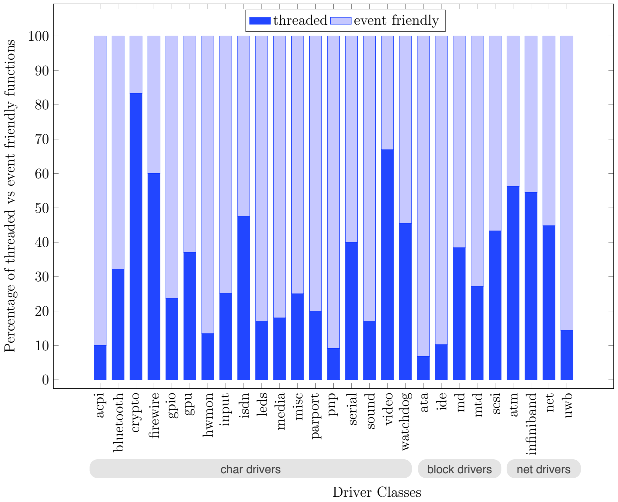


Figure 9. The percentage of driver entry points under coverage of threaded and event friendly synchronization primitives.

```
DrComp signature: 1.798865
static int hpt374_fn1_cable_detect(....)
{
  struct pci_dev *pdev = to_pci_dev(...);

  unsigned int mcrbase = 0x50
             + 4 * ap->port_no;

  u16 mcr3;
  u8 ata66;

  /*Do the extra channel work */
  pci_read_config_word(pdev,
                   mcrbase+2,&mcr3);
  /*Set bit 15 of 0x52 to enable ..*/
  pci_write_config_word(pdev,
                    mcrbase + 2,..);
  pci_read_config_byte(pdev,
                   0x5A,&ata66);
  /*Reset TCBLID/FCBLID to output */
  pci_write_config_word(pdev,
                   mcrbase+2,mcr3);
  if (ata66 & (2 >> ap->port_no))
      return ATA_CBL_PATA40;
  else
      return ATA_CBL_PATA80;
}
```

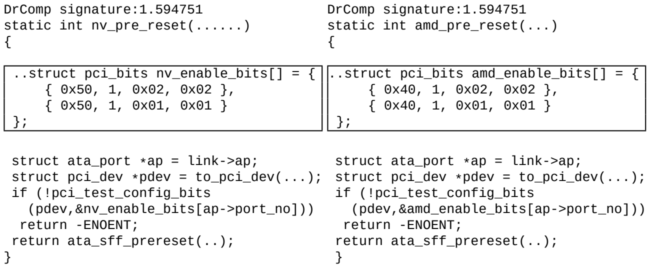
```
DrComp signature: 1.8
static int hpt37x_cable_detect(...)
{
  struct pci_dev *pdev = to_pci_dev(...);

  u8 scr2, ata66;

  pci_read_config_byte(pdev,
                         0x5B, &scr2);
  pci_write_config_byte(pdev,
                         0x5B,...);

   udelay(10); /* debounce */

  /* Cable register now active */
  pci_read_config_byte(pdev,
                         0x5A,&ata66);
  /* Restore state */
  pci_write_config_byte(pdev,
                         0x5B,scr2);
  if (ata66 & (2 >> ap->port_no))
      return ATA_CBL_PATA40;
  else
      return ATA_CBL_PATA80;
}
```

Figure 10. Similar code between two different HPT ATA controller drivers essentially performing the same action. These are among the least-similar functions that DrComp is able to detect these functions as related. The boxes show differentiating statements in the two functions that account for the close signature values.

To address this question, we developed a scalable, code similarity tool for discovering similar code patterns across related drivers and applied it to Linux drivers. The goal of this work is to find driver functions with substantially similar code, indicating that the common code could be abstracted and removed from all drivers to reduce driver code size and complexity.

5.1 Methodology

We developed a new code-similarity tool to handle the number of Linux drivers to find similarities rather than exact copies. We



```
DrComp signature:1.594751
static int nv_pre_reset(......)
{

 ..struct pci_bits nv_enable_bits[] = {
      { 0x50, 1, 0x02, 0x02 },
      { 0x50, 1, 0x01, 0x01 }
 };

 struct ata_port *ap = link->ap;
 struct pci_dev *pdev = to_pci_dev(...);
 if (!pci_test_config_bits
   (pdev,&nv_enable_bits[ap->port_no]))
  return -ENOENT;
 return ata_sff_prereset(..);
}
```

```
DrComp signature:1.594751
static int amd_pre_reset(...)
{

..struct pci_bits amd_enable_bits[] = {
      { 0x40, 1, 0x02, 0x02 },
      { 0x40, 1, 0x01, 0x01 }
 };

 struct ata_port *ap = link->ap;
 struct pci_dev *pdev = to_pci_dev(...);
 if (!pci_test_config_bits
   (pdev,&amd_enable_bits[ap->port_no]))
  return -ENOENT;
 return ata_sff_prereset(..);
}
```

Figure 11. **The above figure shows identical code that consumes different register values. Such code is present in drivers where multiple chipsets are supported as well as across drivers of different devices. The functions are copies except for the constants as shown in the boxes.**

needed to parse through the entire driver source tree consisting of 5 million lines of code, with close to a million lines of code in large classes like network drivers. Most existing clone-detection tools develop a stream of tokens or tree/graphs and perform an *n x n* comparison of all code snippets to detect clones, which given the size of the driver classes, is not possible. In addition, we needed a *similarity* detector for finding code that that is closely related but not identical. With more control over how similar regions are detected using information from the semantics of drivers, we are able to detect more useful similar code. For example, while parsing function calls, we treat calls to the device and kernel differently, improving the accuracy of our similarity-detection tool.

Our similarity tool, *DrComp*, is based on *shape analysis*[3] [11]. This is a method to determine whether clusters of points have a similar shape and variants of these technique are often used to cluster data, to determine nature of data distribution, and to detect identical shapes in computer vision [3].

DrComp generates a set of multidimensional coordinates for every function in every driver. It then detects as similar two functions whose coordinate sets (*shape*) are similar. DrComp processes a driver function and adds a point to the function's shape for every statement or action in statement for loop, kernel interaction, conditionals, device interaction, variable assignment, break and return statements. The coordinates of the point are the offset into the function (line number) and the statement type. To improve accuracy, it is important that the generated shape of the code emphasizes the important characteristics of the function. Hence, we also reinforce the shape of the function by weighting statements that are important yet sparse, such as a function returns and calls to kernel functions. The shape of each driver function is a cloud of points on plane representing the structure of the program. While we consider only two dimensions of the code, the statement type and edit distance to generate the points, our tool can easily be extended to include additional dimensions based on code features (such as nesting depth) or driver features (such as interrupt frequency).

To eliminate complex 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 Euclidean distance between all the points in the code cluster obtained above. The output of DrComp is a signature for every function in every driver. Thus, two functions with identical code will have identical signatures. Furthermore, code that is similar, in that it has a similar structure of loops and I/O operations, will have similar signatures.

Figure 10 shows an example of some of the *least similar* related code in drivers we found. These two functions have signatures within 0.05% of each other. DrComp only looks for the code structure from statement types (also distinguishing kernel and device invocations) and edit distance, so functions may use different arguments (register values), compare against different values or loop on different conditions, and still be grouped as similar code.

5.2 Redundancy Results

DrComp detected that 8% of all driver code is very similar to other driver code. The results of our similarity study are shown in Table 4. For classes with many similarities, we show the number of fragment clusters (sets of similar code), as well as the total number of functions that are similar to another function. For the results in above table, we show results within individual driver classes and not across classes, as they are less likely to benefit from a shared abstraction.

Overall, we identified similarities within a single driver, across a subset of drivers in a class, and in some cases across most drivers in a class. Within a single driver, we found that the most common form of repeated code was wrappers around device I/O, driver library or kernel functions. These wrappers either convert data into the appropriate format or perform an associated support operation that is required before calling the routines but differ from one another because they lie on a different code path. These wrappers could be removed if the kernel interface supported the same data types as the device or if drivers provided appropriate abstractions to avoid such repeated code.

We also find swaths of similar functions across entire classes of drivers. The major difference between drivers for different chipsets of the same device are often constant values, such as device registers or flag values. For example, ATA disk drivers abstract most of the code into a core library, `libata`, and each driver implements a small set of a device-specific functionality. Commonly, these functions are short and perform one or two memory-mapped I/O reads or writes, but with different values for every driver. Figure 5 shows two functions from different ATA drivers with substantially similar code. This practice generates large bodies of very similar drivers with small differences. Further abstraction could additionally simplify these drivers, for example, replacing these routines with tables encoding the different constants. Similarly, a hardware specification language [25] may be able to abstract the differences between related devices into a machine-generated library.

Finally, we note similarities across subsets of drivers in a class. For example, another common class of similarities is wrappers around kernel functions and driver libraries for that class: the `release` method for frame buffers is virtually identical across many of the drivers, in that it checks a reference count and restores the VGA graphics mode. There are a few small differences, but refactoring this interface to pull common functionality into a library could again simplify these drivers.

Implications: Overall, these results demonstrate that there are many opportunities for reducing the volume of driver code by abstracting similar code into libraries or new abstractions. We visually inspected all function clusters to determine how a programmer could leverage the similarity by having a single version of the code. We see three methods for achieving this reduction: (i) procedural abstractions for driver sub-classes, (ii) better multiple chipset support and (iii) table driven programming.

The most useful approach is *procedural abstraction*, which means to move the shared code to a library and provide parameters covering the differences in implementation. There is significant code in single drivers or families of drivers with routines performing similar functions on different code paths. Creating driver-class or sub-class libraries will significantly reduce this code. Second, existing driver libraries can be enhanced with new abstractions that cover the similar behavior. There are many families of drivers that replicate code heavily, as pointed out in Table 4. Abstracting more code out these families by creating new driver abstractions that

[3] We perform geometric shape analysis, not the program analysis technique of the same name.

Driver class	Driver subclass	Similar code fragments	Fragment clusters	Fragment size (Avg. LOC)	Redundancy results and action items to remove redundant code
char	acpi	64	32	15.1	Procedural abstraction for centralized access to kernel resources and passing get/set configuration information as arguments for large function pairs.
	gpu	234	108	16.9	Procedural abstractions for device access. Code replicated across drivers, like in DMA buffer code for savage, radeon, rage drivers, can be removed by supporting more devices per driver.
	isdn	277	118	21.0	Procedural abstraction for kernel wrappers. Driver abstraction/common library for ISDN cards in hisax directories.
	input	125	48	17.23	Procedural abstraction for kernel wrappers. Driver abstraction/common driver for all touchscreen drivers. Procedural abstraction in Aiptek tablet driver.
	media	1116	445	16.5	Class libraries for all Micron image sensor drivers. Procedural abstraction in saa 7164 A/V decoder driver and ALI 5602 webcam driver.
	video	201	88	20	Class libraries for ARK2000PV, S3Trio, VIA VT8623drivers in init/cleanup, power management and frame buffer operations. Procedural abstraction in VESA VGA drivers for all driver information functions.
	sound	1149	459	15.1	Single driver for ICE1712 and ICE1724 ALSA drivers. Procedural abstraction for invoking sound libraries, instead of repeated code with different flags. Procedural abstraction for AC97 driver and ALSA driver for RME HDSPM audio interface.
block	ata	68	29	13.3	Common power management library for ALI 15x3, CMD640 PCI, Highpoint ATA controllers, Ninja32, CIL 680, ARTOP 867X, HPT3x3, NS87415 PATA drivers and SIS ATA driver. Table driven programming for device access in these drivers.
	ide	18	9	15.3	Procedural abstraction for the few wrappers around power management routines.
	scsi	789	332	25.6	Shared library for kernel/scsi wrappers for Qlogic HBA drivers; pmc sierra and marvell mvsas drivers. Large redundant wrappers in mp2sas firmware, Brocade FC port access code.
net	Ethernet/ wireless	1906	807	25.1	Shared library for wireless drivers for talking to device/kernel and wireless routines. Lot of NICs share code for most routines like configuration, resource allocation and can be moved to a single driver with support for multiple chipsets. A driver sub-class library for all or vendor specific Ethernet drivers.
	infiniband	138	60	15.0	Procedural abstraction for Intel nes driver.

Table 4. The total number of similar code fragments and fragment clusters across driver classes and action items that can be taken to reduce them.

support multiple chipsets can simplify driver code significantly. Finally, functions that differ only by constant values can be replaced by table-driven code. This may also be applicable to drivers with larger differences but fundamentally similar structures, such as network drivers that use ring buffers to send and receive packets. By providing these abstractions, we believe there is an opportunity to reduce the amount of driver code, consequently reducing the incidence of bugs and improving the driver development process by producing concise drivers in the future.

6. Conclusions

The purpose of this study is to investigate the complete set of drivers in Linux, to avoid generalizing from the small set of drivers commonly used for research, and to form new generalizations.

Overall, we find several results that are significant to future research on drivers. First, a substantial number of assumptions about drivers, such as class behavior, lack of computation, are true for many drivers but by no means all drivers. For example, instead of request handling, the bulk of driver code is dedicated to initialization/cleanup and configuration, together accounting for 51% of driver code. A substantial fraction (44%) of drivers have behavior outside the class definition, and 15% perform significant computations over data. Thus, relying on a generic frontend network driver, as in Xen virtualization, conceals the unique features of different devices. Similarly, synthesizing driver code may be difficult, as this processing code may not be possible to synthesize. Tools for automatic synthesis of driver code should also consider driver support for multiple chipset as we find that Linux supports over 14,000 devices with just 3,217 bus and device drivers.

Second, our study of driver/device/kernel interactions showed wide variation in how drivers interact with devices and the kernel. At one end, miniport drivers contain almost exclusively device-specific code that talks to the device, leaving kernel interactions to a shared library. At the other end, some drivers make extensive calls to the kernel and very few into shared device libraries. This latter category may be a good candidate for investigation, as there may be shared functionality that can be removed. Overall, these results also show that the cost of isolating drivers may not be constant across all driver classes.

Third, our investigation of driver/device interaction showed that USB and XenBus drivers provide more efficient device access than PCI drivers, in that a smaller amount of driver code supports access to many more devices, and that coarse-grained access may support moving more driver functionality out of the kernel, even on the device itself. Furthermore, many drivers require very little access to hardware and instead interact almost exclusively with the bus. As a result, such drivers can effectively be run without privileges, as they need no special hardware access. We find that USB and Xenbus provide the opportunity to utilize the extra cycles on devices by executing drivers on them and can effectively be used to remove drivers from the kernel leaving only standardized bus code in the kernel.

Finally, we find strong evidence that there are substantial opportunities to reduce the amount of driver code. The similarity analysis shows that there are many instances of similar code patterns that could be replaced with better library abstractions, or in some cases with tables. Furthermore, the driver function breakdown in Section 3 shows that huge amounts of code are devoted to initialization; this code often detects the feature set of different chipsets. Again, this code is ripe for improvement through better abstractions, such as object-oriented programing technique and inheritance [31].

While this study was performed for Linux only, we believe many similar patterns, although for different classes of devices, will show in other operating systems. It may also be interesting to compare the differences in driver code across operating systems, which may demonstrate subtle differences in efficiency or complexity. Our study is orthogonal to most studies on bug detection. However, correlating bugs with different driver architectures can provide insight on the reliability of these architectures in real life.

Acknowledgements

This work is supported in part by National Science Foundation (NSF) grants CNS-0915363 and CNS-0745517 and a grant from Google. We also thank Matt Renzelmann and the anonymous reviewers for their invaluable feedback. Swift has a significant financial interest in Microsoft.

References

[1] J. Andrus, C. Dall, A. Vant Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *SOSP*, 2011.

[2] 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 *Eurosys*, 2006.

[3] S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *Transactions on Pattern Analysis and Machine Intelligence*, 2002.

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

[5] 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.

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

[7] T.-c. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. *Operating Systems Review*, 33, 1999.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *SOSP*, 2001.

[9] F. M. David et al. CuriOS: Improving reliability through operating system structure. In *OSDI*, 2008.

[10] Digi International. AnywhereUSB. `http://www.digi.com/products/usb/anywhereusb.jsp`.

[11] I. Dryden and K. Mardia. *Statistical shape analysis*, volume 4. Wiley New York, 1998.

[12] K. Fraser et al. Safe hardware access with the Xen virtual machine monitor. In *OASIS Workhop*, 2004.

[13] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *OOPSLA*, 2010.

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

[15] V. Ganapathy, A. Balakrishnan, M. M. Swift, and S. Jha. Microdrivers: A new architecture for device drivers. In *HOTOS11*, 2007.

[16] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *ASPLOS*, Mar. 2008.

[17] A. Hari, M. Jaitly, Y.-J. Chang, and A. Francini. The swiss army smartphone: Cloud-based delivery of usb services. In *Mobiheld*, 2011.

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

[19] A. Kadav and M. Swift. Live migration of direct-access devices. *Operating Systems Review*, 43(3):95–104, 2009.

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

[21] B. Leslie et al. User-level device drivers: Achieved performance. *Jour. Comp. Sci. and Tech.*, 2005.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, 2004.

[23] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *SOSP*, 2011.

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

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

[26] Microsoft Corp. Web services on devices. `http://msdn.microsoft.com/en-us/library/aa826001%28v=vs.85%29.aspx`.

[27] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, 2002.

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

[29] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Eurosys*, 2008.

[30] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *VEE*, 2009.

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

[32] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Eurosys*, 2009.

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

[34] L. Ryzhyk, Y. Zhu, and G. Heiser. The case for active device drivers. In *APSys*, Aug. 2010.

[35] A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter. A declarative language approach to device configuration. In *ASPLOS*, 2011.

[36] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. John Wiley and Sons, eighth edition, 2009.

[37] M. Spear et al. Solving the starting problem: Device drivers as self-describing artifacts. In *Eurosys*, 2006.

[38] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. In *OSDI*, 2004.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *In TOCS*, 2005.

[40] Úlfar Erlingsson et al. Xfi: software guards for system address spaces. In *OSDI*, 2006.

[41] USB Implementors Forum. Universal serial bus mass storage class specification overview. `http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf`, 2003.

[42] USB Implementors Forum. Universal serial bus 3.0 specification. http://www.usb.org/developers/docs/usb_30_spec_071311.zip, 2011.

[43] D. A. Wheeler. Sloccount. `http://www.dwheeler.com/sloccount/`.

[44] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, 2008.

[45] L. Wittie. Laddie: The language for automated device drivers (ver 1). Technical Report 08-2, Bucknell CS-TR, 2008.

[46] Xen.org. Writing xen drivers: Using xenbus and xenstore. `http://wiki.xensource.com/xenwiki/XenBus`, 2006.

[47] F. Zhou et al. SafeDrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.

Fine-Grained Fault Tolerance using Device Checkpoints

Accepted at ASPLOS 2013 (Camera Ready Pending).

Asim Kadav, Matthew J. Renzelmann, Michael M. Swift

Computer Sciences Department, University of Wisconsin-Madison
{kadav, mjr, swift} @cs.wisc.edu

Abstract

Recovering faults in drivers is difficult compared to other code because their state is spread across both memory and a device. Existing driver fault-tolerance mechanisms either restart the driver and discard its state, which can break applications, or require an extensive logging mechanism to replay requests and recreate driver state. Even logging may be insufficient, though, if the semantics of requests are ambiguous. In addition, these systems either require large subsystems that must be kept up-to-date as the kernel changes, or require substantial rewriting of drivers.

We present a new driver fault-tolerance mechanism that provides fine-grained control over the code protected. Fine-Grained Fault Tolerance (FGFT) isolates driver code at the granularity of a single entry point. It executes driver code as a transaction, allowing roll back if the driver fails. We develop a novel checkpointing mechanism to save and restore device state using existing power-management code. Unlike past systems, FGFT can be incrementally deployed in a single driver without the need for a large kernel subsystem, but at the cost of small modifications to the driver.

In the evaluation, we show that FGFT can have almost zero runtime cost in many cases, and that checkpoint-based recovery can reduce the duration of a failure by 79% compared to restarting the driver. Finally, we show that applying FGFT to a driver requires little effort, and the majority of drivers in common classes already contain the power-management code needed for checkpoint/restore.

1. Introduction

In most commodity operating systems, third-party driver code executes in privileged mode. Faulty device drivers cause many reliability issues in these systems [8, 36]. Hence, there has been significant research to tolerate driver failures using programming-language and hardware-protection techniques [3, 6, 15, 16, 22, 25, 44]. These systems execute the entire driver as a single isolated component. However, much of this work focuses on *detecting* failures and *isolating* drivers from the rest of the system. Few of these systems address how to *restore* driver functionality beyond simply reloading the driver, which may leave applications non-functioning.

Most driver-reliability systems do not try to restore device state and instead completely restart failed drivers [18, 41, 45], effectively resetting device state to a known-good configuration. The state-of-the-art mechanism for restoring driver functionality, shadow drivers [40], logs state-changing operations at the driver/kernel interface. Following a failure, shadow drivers restart the driver and replay the log in order to restore internal driver and device state. This resets the driver and device to a state functionally equivalent to its pre-failure state.

This approach, complete driver isolation and logging for recovery, poses four problems:

1. *Too hard:* Shadow drivers must be written for every class of driver and must be updated when the interface changes. This adds a large body of code to the kernel requiring constant maintenance, which is a high barrier to adoption. Other systems require substantially rewriting drivers, which is also a barrier.
2. *Not enough:* Shadow drivers must encode the semantics of the kernel/driver interface. However, many drivers have proprietary commands that cannot be captured by a generic shadow driver, leading to incomplete recovery. Recent work showed that up to 44% of drivers have non-class behavior [20].
3. *Too expensive:* Shadow drivers must interpose on and log *all* invocations of a driver. Continuous monitoring imposes a performance cost, particularly on high-performance devices such as SSDs and NICs even when the critical I/O path is bug-free.
4. *Too slow:* Restarting a driver, the first step of log replay, can be slow (multiple seconds) due to complex initialization code and therefore may not be useful in latency-sensitive environments.

A key source of these problems is that prior systems seek *completeness*: applying to *all* driver code at *all* times. While this reduces the per-driver cost, it pushes up both development and runtime costs.

We developed a new driver fault tolerance mechanism to address these shortcomings called *Fine-grained Fault Tolerance* (FGFT). Rather than isolating and recovering from the failure of an entire driver, FGFT executes a driver *entry point* as a transaction and uses software-fault isolation to prevent corruption and detect failures. If the call faults, FGFT rolls back driver state and fails the call. On entry to a driver, a stub copies parameters to the driver code. Only if the driver executes correctly are the results copied back; otherwise, the copy is destroyed.

In order to restore device state modified by a driver before faulting, we developed a novel *device state checkpointing* mechanism that can capture the device state. The stub captures a checkpoint before invoking the driver, and restores the checkpoint on failure. This mechanism leverages existing power-management code present in most drivers, which greatly reduces the development cost of adopting FGFT.

FGFT shifts the cost of driver fault tolerance to the faulty code. While shadow drivers and whole-driver isolation require up-front code for any instance of a class of drivers, FGFT instead requires small changes to the driver itself to support isolation and implement checkpointing. Where past isolation mechanisms interpose on *all* driver code and reduce its performance uniformly, FGFT *only* imposes a cost on entry points selected for isolation. Thus, the cost of executing a single call with fault tolerance may be higher with FGFT than other systems, but when applied only to code off the critical path it has much *lower* overhead because the critical code is left unchanged. Thus, one possible use for FGFT is to apply it selectively to vulnerable code suspected or known to have bugs.

The contributions of our work are:

- We build fine-grained fault tolerance, a system consisting of a static analysis and code generation tool that provides isolation by executing each driver request on a minimal copy of required driver state. Our system can be used to isolate specific requests and we show from a study of published bugs that fine-grained isolation is practical since bugs only affect 14% of all entry points in buggy drivers.
- We demonstrate a *novel* mechanism to create device checkpoints on a running system. In a study of six drivers, we show that taking a checkpoint is fast, averaging only 20 μs.
- We show how to use checkpoints and transactional execution of driver code to provide fast recovery and remove the permanent overhead of monitoring *all* requests.
- We show that the implementation effort of FGFT is small: we added 38 lines of code to the kernel to trap processor exceptions, and found that device checkpoint code can be constructed with little effort from power-management code present in 76% of drivers in common driver classes.

We begin with an overview of the FGFT design.

2. Design Overview

Fine-grained Fault Tolerance is a system to tolerate faults in drivers using a *pay-as-you-go* model based on checkpoints for recovery. This system protects code from faults at the granularity of a single thread executing a single entry point. FGFT recovers from any failures that occur during the function. This can greatly reduce the cost of isolating and tolerating faults, because far less code is affected.

We list four goals of providing fine-grained fault tolerance:

1. *Class Independent.* Isolation and recovery should be independent of the driver-kernel interface and should be able to recover driver actions from proprietary commands.
2. *Low infrastructure.* Little new code should be added to the kernel in support of FGFT.
3. *Pay-as-you-go.* FGFT should not have a permanent overhead of isolation or monitoring driver behavior. Furthermore, programmer effort should only be required only when fault tolerance is desired.
4. *Fast recovery.* FGFT should restore driver functions quickly after a failure without affecting other threads concurrently executing in the driver.

The first goal enables FGFT to apply to a broad range of drivers, and the second reduces the adoption cost for an operating system. Pay-as-you-go ensures that for high-performance drivers, tolerating faults in code off the critical path has little cost. Fast recovery enables its use in latency-sensitive environments.

The two major components of FGFT are an isolation mechanism to prevent a faulty driver from corrupting the OS and to detect failures, and a recovery mechanism to restore the driver to a functioning state after a failure. We begin a discussion of our fault model to motivate our design choices.

2.1 Fault Model

A driver entry point is a driver function invoked by the kernel or applications to access specific driver functionality. Each driver registers a set of functions as entry points. FGFT provides fault tolerance at the granularity of a single entry point into a driver. In contrast, past systems treat the entire driver as a component with internal state. As the driver executes, the FGFT isolation mechanism enforces fine-grained memory safety. It ensures that the driver is only allowed to access its stack and data passed to the driver; access to anything else will be treated as a fault. FGFT detects memory failures (such as null pointer dereferences) and reading/writing unintended kernel and driver structures. Furthermore, FGFT also catches processor exceptions (NULL pointer exception, general protection fault, alignment fault, divide error (divide by zero), missing segments, and stack faults) and triggers recovery if they arise out of isolated portions of driver code.

We design for an open-source environment, and therefore trust the compiler to produce code that correctly accesses the stack. We also assume that the driver is unable to hang or damage the device, although it may misconfigure the device.

In addition to enforcing memory safety, FGFT uses marshaling to copy data in and out of the driver. Type errors that cause the marshaling to fail will be detected, although errors with compatible types (such as treating an array of bytes as an array of longs), will not be.

FGFT on its own does not provide any semantic checks to enforce driver invariants. The major consequence of our decision is that driver faults must be detected within the entry point where they occur. Otherwise, failures that begin with one entry point improperly setting a flag that is read by another cannot be tolerated.

A key benefit of FGFT is that by operating on specific entry points it can be selective about what code should be hardened against faults. We call the entry points to be isolated *suspect*. The suspect code can be executed in isolation while the remainder of the driver executes in the kernel at full speed. Hence, FGFT is useful when specific driver code is known to have problems, such as just-patched code or code with known but un-patched vulnerabilities and can execute these requests in isolation. We identify at least three cases where a fine-grained model is useful:

1. *Untested code*: Device drivers often contain untested code that are only invoked by specific chipsets or during recovery can be invoked using FGFT.
2. *Statically found bugs*: There are many static analysis tools that identify hard to find/trigger driver bugs with substantial false positive rates. FGFT can be integrated with existing static analysis tools until a fix is issued, which often takes considerable time. This approach limits failures when such code is triggered under buggy situations, while limiting the overhead at other times.
3. *Runtime monitoring tools*: Runtime monitoring tools flag incoming requests based on their parameters, such as a specific `ioctl` command code, or enabled at run time through module parameters using run-time monitoring [23] or security tools [31]. FGFT can dynamically decide whether to execute code in isolation or at full speeds.

We next describe the two major components of FGFT: isolation and recovery.

2.2 Fine-Grained Isolation

FGFT provides isolation by forcing suspect code to operate on a *copy* of driver and kernel data. This ensures that anything the entry point does will not be seen by other threads or the kernel until it successfully completes, and allows quick recovery after a failure by deleting the copy. Thus, FGFT creates a clean copy of data needed for a driver entry point on every invocation, which consists of all data referenced by the entry point: parameters, global driver variables, and global kernel variables.

We use entry points as the granularity of isolation because it closely matches internal driver structure: they provide a natural

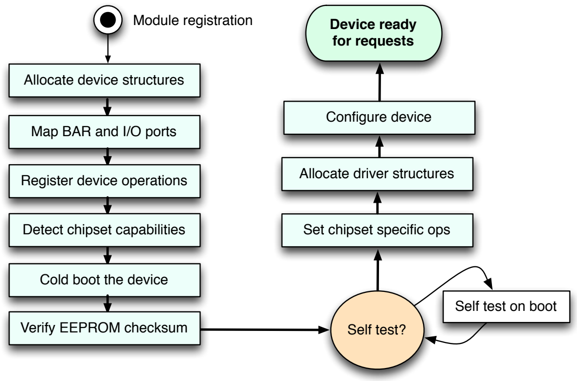


Figure 1. **Modern devices perform many operations during initialization such as setting up kernel and device structures based on chipset and device features, checksumming device ROM data, various device tests followed by driver initialization and configuration.**

boundary for returning errors after a fault, and drivers already synchronize concurrent invocation of entry points. If two driver threads cannot run concurrently in the driver, then driver synchronization ensures that one of them blocks until the other successfully completes. Thus, FGFT reuses existing synchronization mechanisms to ensure that when suspect code runs, no other threads are active in the driver. This ensures that any changes to device state will not be seen until the entry point completes successfully or it fails and recovery completes.

FGFT provides entry-point isolation with a copy-in/out model of driver and kernel state when suspicious entry-points are invoked. FGFT uses static analysis and code-generation to generate another kernel module that contains suspect entry points instrumented for memory safety. Furthermore, FGFT generates communication code containing marshaling routines to copy driver and kernel state necessary for executing these entry points in isolation. Since the static analysis to marshal the data structures required by the isolated copy can be imprecise, FGFT requires a programmer to annotate ambiguous data types in the driver code. In order to provide even finer control over when to provide fault tolerance, FGFT automatically inserts taps, which are predicates that can decide at runtime whether to invoke the normal or fault-tolerant version of an entry point.

FGFT detects faults through run-time memory safety checks that detect access to unreachable addresses – memory not passed as a parameter or allocated by the entry point. Since, FGFT generates the code for copy-in/out, it is able to provide fine-grained memory safety (base and bound validation [27]). Furthermore, for failure detection, FGFT also interposes on kernel trap handlers and detects if the faults originate from the suspicious entry-points and if FGFT needs to provide recovery.

2.3 Checkpoint based Recovery

FGFT relies on checkpoints prior to a driver entry point for recovery. Unlike log-based recovery, which requires knowing how to replay requests, checkpoints can restore state independent of how a function modifies driver state. For example, a checkpoint of a device prior to an `ioctl` call allows its state to be recovered no matter what the call does.

Log-based recovery is also slow enough that the technique may not be useful in latency-sensitive environments. The primary delay comes from probing the device all over again which cold boots the device and performs the initialization steps as shown in Figure 1. For example, during initialization a network driver probes for the device, verifies EEPROM contents, tests the device, and registers the device with the kernel.

The checkpoint of the driver's state in memory is captured automatically through the copy-in/out model of invocation. Suspect code always executes on a copy of the driver state, so the original data is unmodified and need not be restored. The major challenge, though, is the *device state*, which may be modified unpredicatbly by the driver. We therefore require that drivers provide a facility for capturing and restoring device state. Prior to invoking suspect code, FGFT can take a checkpoint, and following a failure, it can restore the checkpoint.

An appealing approach is to treat devices like memory and copy memory-mapped I/O regions. However, reading registers may have side effects such as clearing counters. In addition, some devices overlay two logical registers, one for read and one for write, at the same address. Instead, we take inspiration from code *already present* in many drivers that must perform *nearly the same* task as checkpoint/restore: power management.

The functionality provided by power management, to suspend a device before entering the low power mode and restoring it when transitioning to high power mode, is similar to what is required to support device checkpoints. We reuse the suspend/resume code by identifying code that supports saving state to memory from the code that actually suspends the device. Similarly, we identify code required for restoring this state. In Section 4, we describe in detail how power management code can be re-factored to support checkpoint/ restore in device drivers and how existing driver synchronization can be used to arbitrate device access.

2.4 Design Summary

FGFT improves the state of art in driver recovery and meets its goals. FGFT provides class-independent driver recovery with checkpoints as opposed to restarting the driver. Hence, FGFT discards failed requests and retains proprietary driver state such as *ioctls* that were issued before the failure.

FGFT requires very little kernel code, as the code for isolation is generated automatically and the recovery code requires only small modifications to existing driver code. The annotation cost for isolation and recovery is only required when a driver needs fault tolerance. Only when a suspicious request executes does FGFT execute it in isolation, thus limiting isolation overhead to these requests. Compared to FGFT, Nooks [41] and SUD [3] require a new kernel subsystem and writing and maintaining wrappers around the driver-kernel interface.

There is also no recovery overhead of monitoring the correctly executing requests at *all* times since driver recovery is based on checkpoints. Finally, FGFT provides fast recovery since it does not restart the driver and re-execute the complicated device probe routines. Since the device state is restored from a checkpoint, the recovery times are an order of magnitude shorter as we demonstrate in our evaluation in Section 5.

3. Fine-Grained Isolation

Isolation ensures that the driver and kernel changes made by a request are not propagated if the request fails. We need the following properties from an isolation mechanism:

1. *Transactional execution*: We need to execute the driver entry points in a transactional fashion that keeps a clean copy of all data modified by the driver.
2. *Memory safety and fault detection*: We need to ensure a driver cannot corrupt the kernel or other threads in the driver and provide mechanisms that detect when a driver has failed.
3. *Synchronization*: Threads executing in the driver need to synchronize with other threads to ensure they do not corrupt shared state in the kernel, driver, or device.

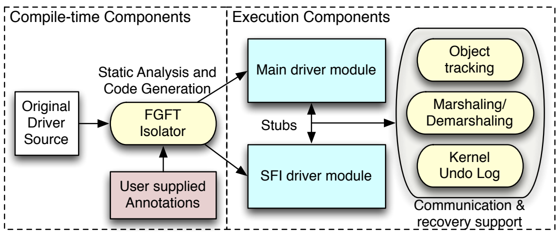


Figure 2. Fine-grained fault tolerance replicates driver entry points into a normal driver and an *SFI driver module*. A runtime support module provides communication and recovery support.

To achieve these goals, we rely on well-understood compile-time software fault isolation (SFI) [43]. As a driver entry point operates on data shared with the rest of the driver, the SFI mechanism must allow access to such data but prevent its corruption. FGFT therefore executes isolated code on a *minimal copy* of the driver and kernel, which is a copy of data referenced from an entry point but not entire structures. If the entry point does not fail, FGFT merges the copy back into the real driver and kernel structures. On a failure, the copy is discarded. This in effect executes the suspect entry point as a transaction using lazy version management [21].

However, not all data can be copied. Structures shared with the device, such as network transmit and receive rings, cannot be copied because the device will not share the copied structure. Instead, FGFT grants suspect code direct read and/or write access to these structures and relies on device-state checkpointing to restore these structures following a failure. Furthermore, driver code used for recovery cannot be isolated and must be trusted.

Since FGFT must copy all data accessed by a driver, its isolation will incur a performance overhead when invoked frequently, such as by high-bandwidth devices that handle thousands of requests per second. For such scenarios, using hardware support such as transactional memory [33] can reduce the overhead of copying for every request.

We implemented FGFT for the Linux 2.6.29 kernel. Figure 2 shows the components of FGFT. We next describe how FGFT provides isolation, communicates with isolated code, and detects failures.

3.1 Software Fault Isolation

As FGFT targets open-source Linux device drivers, we implement SFI using a source-code rewriting tool called *FGFT Isolator* written using CIL [30]. It generates isolation code into the driver and produces communication code, described below, for communicating with the isolated code.

Isolator generates an additional driver module called the *SFI module* that contains a copy of the suspect entry points and all functions transitively called from those functions, instrumented for SFI. In addition, Isolator generates a new version of the driver that invokes the SFI module entry points. At the top of the existing entry points, Isolator inserts a test to see whether to execute normally or in isolation, and if so invokes the SFI module.

The decision to invoke a given entry point in isolation can also be made in one of three ways. First, a developer can use the attribute `__attribute__((isolate))` to manually specify which functions to isolate. This causes the function to always execute with isolation. Second, FGFT can *automatically* use any static analysis tool to identify buggy code and which entry points are affected. These entry points are then always executed with isolation. Finally, the decision can be made at run time. A fault management system, such as the Solaris Fault management Daemon [38], can call into the SFI module and specify which functions to execute with isolation. Furthermore, a fault management system can register a function pointer at run time that takes the same arguments as the suspicious function and returns a decision of whether to isolate or not.

In addition to producing the SFI driver code, Isolator also produces communication code that invokes the SFI driver and copies in the minimal driver and kernel state needed by the suspect entry points, copies out any changes made by the SFI driver, and initiates recovery following a detected failure. Isolator manages resource allocation, synchronization and I/O across the two copies. Isolator only detects memory failures. For other failures, such as arithmetic exceptions, we trap processor exceptions and check if they originate from SFI module.

Isolator uses CIL's memory tracking module [30] to instrument all memory references in the driver. It inserts a call to our `memcheck` function that verifies the target of a load/store is valid. If not, it detects a failure and invokes the recovery mechanism. The `memcheck` routine consults a *range table* to verify memory references and provide fine-grained memory protection by only allowing access to driver and kernel data as identified at compile time. This table contains the addresses and lengths of copied data structures and buffers shared with the device. The range table is created on every invocation of a suspect entry point and flushed on return.

We do not add all local variables to the range table because we trust the compiler to generate correct code for moving variables between registers and the stack. However, if the driver ever takes the address of a local variable, or it creates an array as a local variable, then Isolator adds a call in the instrumented SFI driver to add the variable's address and length to the range table and remove it from the range table when the variable goes out of scope. Similarly, we trust the compiler to produce valid control transfers and do not instrument branch or call instructions.

3.1.1 Communication Code for Entry Points

FGFT Isolator generates *stub code* to invoke suspect entry points that copies into and out of the driver. Similar to RPC stubs, these stubs create a copy of the parameters passed to the suspect code, but also copy any driver or kernel global variables it uses. When the suspect entry point completes, stub code copies modified data structures and return values back to the regular driver and kernel in the current thread. An alternative approach would be to rely on transactional-memory techniques to dynamically create a copy of data as it is referenced, which may have lower copying costs but higher run-time costs [2].

Isolator automatically identifies the minimal data needed for an entry point through static analysis. This includes the structure fields from parameters referenced by the entry point or functions it calls plus fields of global variables referenced. As they copy data, stubs update the range table with the address and length of each object. For objects that cannot be copied (such as those shared with the device), stubs fill in the existing address of the field, its length, and whether the entry point needs read, write, or read/write access.

If suspect code callbacks invoke the kernel, Isolator generates stubs for kernel functions that copy parameters to the kernel and copies kernel return values back to suspect code. The SFI driver may pass in fields from its parameters to the kernel as arguments. To avoid creating a new copy of these fields, as would be done by RPC, FGFT maintains an *object tracker* that maps the address of kernel and regular driver objects to the address of objects in the SFI driver. Stub code consults the object tracker when calling into the kernel to determine whether arguments refer to a new or existing object. If an object exists, stubs copy the argument back to the existing object and otherwise temporarily allocate a new object. To support recovery, stubs may generate a *compensation entry* in a *kernel undo log*. This log records operations that must be reversed on failure, such as freeing memory allocations.

The stub code must know the layout of data structures and whether data is shared with the driver in order to correctly copy data. As driver code often contains ambiguous data structures such as `void *` pointers or list pointers (***e.g.***, `struct list_head`), we rely on programmer-provided annotations to disambiguate such code [45]. These annotations also declare which structure fields or parameters are shared with the device and should not be copied. In Section 5, we evaluate the difficulty of providing annotations.

Some driver functions trigger synchronous callbacks. For example, the `pci_register_driver` function causes a callback on the same thread to the driver's `probe` function. FGFT treats the callback as a *nested transaction*: it causes another isolated call operating on a second copy of the data.

3.1.2 Resource Access from SFI module

Some resources cannot be copied into the driver because they attach additional semantics or behavior to memory.

I/O memory. Driver entry points may communicate with the device by writing to I/O memory. Stubs grant the SFI driver read/write access to memory-mapped I/O regions and memory shared with the device via `dma_alloc_coherent`. Isolator identifies these regions with annotations and creates stubs that grant drivers direct read/write access. SFI threads also re-synchronize with device (usually accomplished by sleeping for the device) since the SFI thread accesses the device directly and any changes are immediately visible.

Locks. Drivers may synchronize with other threads using spin locks and mutexes. Stubs pass locks through without copying but only allow read access. Suspect code must call back into the kernel to acquire a lock. After acquiring a lock, the SFI driver copies its parameters back from the kernel. The stub code for kernel locking routines add a compensation entry to the kernel undo log to release the lock after a failure. To ensure that changes made by suspect code are not seen by the rest of the kernel, the lock stubs defer releasing locks until after the entry point returns to the kernel.

The above mechanism protects shared structures across different driver threads. However, the suspicious thread can also block waiting for data to arrive on shared structures that have been copied over from other driver threads. In such cases, FGFT requires extra annotations to provide read access on the shared structures/fields which can be cumbersome for the developer.

Memory allocation. Stubs for allocators invoke the kernel allocator, add the returned memory region to the range table and generate a compensation entry to free the memory on failure. The newly allocated memory is not copied into the driver because its contents do not need to be preserved.

3.2 Failure Detection

In addition to protecting the kernel and regular driver code from corruption, isolation provides the primary means to detect failures. FGFT's SFI mechanism implements *spatial memory safety* [27]: every memory reference must be within an object made accessible during the copy process. Thus, references outside the range table indicate a failure.

Stubs can detect additional failures when copying data back to the kernel. For example, if the driver writes an invalid address into a data structure, the copying code will dereference that address and generate an exception.

We also modified the Linux kernel exception handlers to detect unexpected traps from the SFI driver as failures. If one occurs, the trap handler sets the instruction pointer to the recovery routine. This is the only change to the Linux kernel, and required only 38 lines of code.

The detection mechanisms may miss several categories of failures. First, if the driver violates its own data structure invariants, stubs may not detect the problem. Recent work on identifying and verifying data structure invariants could detect these faults [5]. For example, if a suspect entry point sets a flag indicating that a field is valid but does not set the field, then corruption will leak out of the SFI driver. Second, the driver does not provide strong type safety, so the driver may assign a structure field to the wrong type of data. While this may be detected when stub copies data, it is not guaranteed. Finally, FGFT does not enforce kernel restrictions on the range of scalar values, such as valid packet lengths.

4. Checkpoint-based recovery

The FGFT system is built around checkpoint-based recovery. While checkpointing and restoring memory state is simple using techniques such as transactional memory or copy-on-write, it has not previously been possible to capture the state of a device. Without this, restoring memory state will lead to a driver that is inconsistent with respect to its device, believing incorrectly that it has performed an action or is operating in a different mode. We first describe device state checkpointing, which is the basis of FGFT's recovery mechanism. We then describe how FGFT uses device checkpoints to recover in case of a failure.

4.1 Device state checkpointing

To be useful, a device checkpoint mechanism should fulfill the following goals:

1. *Lightweight.* There should be no continuous monitoring or long-latency operations.
2. *Broad.* The mechanism must work with a wide range of devices/drivers, including those with unique behavior.
3. *Consistent.* Drivers are often invoked on multiple threads, and checkpoints must be a consistent view of device state.

We identified the suspend/resume code already present in many drivers as having much of the functionality needed to implement checkpoint and restore. We next describe how power management for drivers works, and then describe how to reuse the functionality for driver recovery with checkpoints.

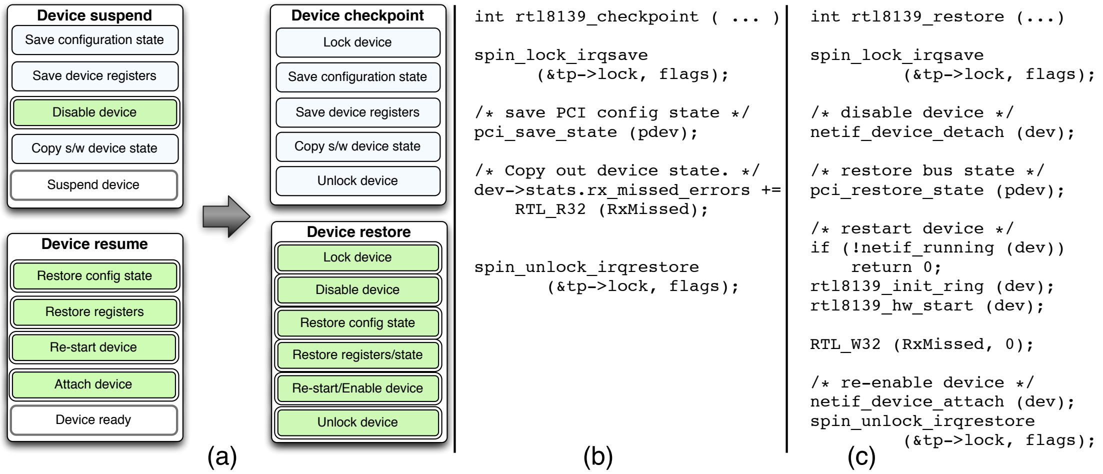
4.1.1 Suspend/Resume Background

Modern operating systems can dynamically reduce their power consumption to provide a hot *standby* mode, also called *suspend to RAM*, which disables processors and devices but leaves state in memory. One major component of reducing power is to disable devices. Thus, operating systems direct devices to switch to a low-power state when the system goes into standby mode. The behavior of devices is specified by the ACPI specification for the platform and by buses, such as PCI and USB.

In order to transition quickly between standby and full-power mode, drivers implement a power-management interface to save device state before entering standby mode, and to restore device state when leaving standby mode [12, 13]. These operations must be quick to allow fast transitions. The system-wide suspend-to-RAM mechanism saves the memory state of the driver, and the driver is responsible for saving and restoring any volatile device state.

Drivers implement a power management interface with methods to save and restore state. For example, Linux PCI devices implement these two methods:

```
int (*suspend) (struct pci_dev *dev,
     pm_message_t state);
int (*resume) (struct pci_dev *dev);
```



```
int rtl8139_checkpoint ( ... )

spin_lock_irqsave
       (&tp->lock, flags);

/* save PCI config state */
pci_save_state (pdev);

/* Copy out device state. */
dev->stats.rx_missed_errors +=
    RTL_R32 (RxMissed);


spin_unlock_irqrestore
       (&tp->lock, flags);
```

```
int rtl8139_restore (...)

spin_lock_irqsave
         (&tp->lock, flags);

/* disable device */
netif_device_detach (dev);

/* restore bus state */
pci_restore_state (pdev);

/* restart device */
if (!netif_running (dev))
    return 0;
rtl8139_init_ring (dev);
rtl8139_hw_start (dev);

RTL_W32 (RxMissed, 0);

/* re-enable device */
netif_device_attach (dev);
spin_unlock_irqrestore
         (&tp->lock, flags);
```

Figure 3. Our device state checkpointing mechanism refactors code from existing suspend-resume routines to create checkpoint and restore for drivers as shown in Figure (a). The checkpoint routine only stores a consistent device snapshot to memory while the restore loads the saved state and re-initializes the device. Figures (b) and (c) show checkpoint and restore routines in the rtl8139 driver.

When saving device state to memory, the driver may invoke the bus to save bus configuration data, as well as explicitly save the contents of select device registers that are not captured by the configuration state. The driver then instructs the device to suspend itself. Simple devices that have no state may simply disable the device.

Upon resume, drivers wake the device, optionally perform a soft reset, restore their saved state. Since the latency of a system to respond post-resume is critical, the initialization is lightweight compared to restarting the driver, as it assumes the device has not changed. Similar to suspend, simple devices may just re-enable the device without restoring state.

For a system to support standby mode, all drivers must support power management. While not all drivers do (Linux is notorious for incomplete support [26]), it is widely implemented by Windows and MacOS drivers, and support in Linux drivers is improving.

The functionality provided by driver power management is very similar to what is needed for device state checkpointing. First, it provides the ability to save device state to memory in a way that allows applications to continue functioning. Second, even though the device may continue to receive power, the soft reset that occurs when re-enabling a device ensures that any previous state is replaced by the restored state. Finally, power management is implemented by most commonly used drivers. However, it is not directly usable for checkpointing: power management routines lack the ability to continue executing after suspending a device because the device has been disabled.

4.1.2 Checkpoint

Device state checkpointing is constructed from a subset of the device suspend support already present in drivers. A device may have many distinct forms of state, each of which require a different mechanism for checkpoint:

1. Device configuration information published through the bus configuration space.
2. Device registers with configuration data specific to the device.
3. Counters and statistics exported by the device and aggregated by the driver.
4. Addresses of memory buffers shared with the driver, such as the DMA ring buffers used by network drivers to send or receive packets.

We note that a checkpoint may not actually contain the full state of the device. Rather, it must contain *enough* information that functionality can later be restored without affecting applications. Thus, device state that can be recreated or recomputed need not be saved. Furthermore, the checkpoint only contains the device state. To be restored properly, it requires a consistent copy of the driver state taken at the same time. Thus, it must be paired with mechanisms such as transactional memory or copy-on-write to save the driver's state.

The configuration state is the easiest to save. Most buses provide a method to save configuration information. For example, PCI drivers in Linux use pci_save_state, that saves a set of standard registers and the base address registers (BARs) to memory. Each driver, though, must handle the remaining state, separately.

The driver explicitly saves register contents and counters in an internal driver structure. The difference between registers and counters arises during recovery, described below, because counter values cannot be written back to the device.

Memory buffers shared with the device can be recreated. As a result, most device drivers do not include the address of these buffers in a checkpoint. Instead, they free buffers during suspend and re-allocate them during resume.

Figure 3(a) diagrams the tasks performed by suspend and resume, and shows how that code is shuffled to create checkpoint and restore functionality. Of the suspend code, checkpointing reuses all the functionality except detaching the device with the kernel and suspending the device. As an example, Figure 3(b) shows the code to checkpoint the 8139too driver.

It may be necessary to checkpoint a driver while it is in use. Existing suspend routines assume the device is quiescent when the device state is saved. Checkpoint, though, may be called at any time. Thus, it must be synchronized with other threads using the driver. Because device state checkpointing must be coordinated with other mechanisms for capturing driver state, we do not put our own synchronization code in the checkpoint routine but re-use existing device locks in the driver. Device locks protect against conflicting configuration operations, or operations like resetting the device when I/O operations are in progress. This ensures that we do not corrupt device state assumed by another thread in progress when we reset device state in case of a failure.

4.1.3 Restore

The restore operation can be constructed from a mix of suspend and resume code. Normally the resume function is invoked when the device returns to full power and needs to be reconfigured. In the case of a checkpoint, though, the device is already running at full power. Thus, resume invokes the bottom half of the suspend routine to disable the device before restoring state.

The restore operation proceeds in four steps:

1. Disable the device to put it in a quiescent, known state.
2. Restore bus configuration state
3. Re-enable the device
4. Restore device-specific state

Figure 3(c) shows the code to restore state for a simple network driver.

Of the four categories of driver state, only configuration state and saved device registers can be reloaded. Counters, which cannot be written back to the device, are restored by adjusting the driver's version of the counter. Typically, the driver will read the device counter and update a copy in memory, and reset the device's counter. To restore the device's counter state, the driver only resets the device's counter; the in-memory copy of the counter must be saved as part of the driver's memory state.

To restore shared buffers, the driver releases existing shared buffers after disabling the device. As part of re-enabling the device, it recreates shared buffers and notifies the device of their new addresses. While this slows restore, it makes checkpoint very efficient because only irretrievable state is saved.

Unlike suspend-resume, it may be useful to use device state checkpointing from interrupt contexts, where sleeping is not allowed. As a result, checkpoint and restore code must convert sleeps to busy waits (`udelay` in Linux) and use memory allocation flags safe for interrupt context (`GFP_ATOMIC` in Linux)

Compared to full-driver restart, resume improves performance because it does not re-invoke device probe, often the lengthiest part of starting a device normally. Furthermore, drivers for newer buses such as USB and IEEE394 do not restart the device because the bus handles this operation. This further reduces restore times. For PCI devices, a further optimization is to avoid changing the power mode of the device. However, we observed that many drivers do not require actually powering down the device before performing restore. For these drivers, restore can be sped up by skipping these unnecessary power mode changes.

4.1.4 Discussion

Device state checkpointing provides several benefits compared with a logging approach to capturing driver/device state. First, it can be invoked at any time and has no cost until invoked. Thus, it has no overhead for infrequent uses. Second, it handles state unique to a device, such as configuration options. Correct standby operation demands that devices remain correctly configured across standby, and hence drivers must already save and restore any required state. However, device state checkpointing relies on power management code, which may not be present in all devices. It also requires a programmer to implement checkpoint/restore for every driver. We evaluate these concerns in Section 5.

Limitations. There is a risk in utilizing a mechanism for an unintended purpose: the driver continues running following a checkpoint and may thus further modify the device state. In contrast, devices are normally idle between suspend and resume. Thus, it is possible that the state saved is insufficient to fully restore the device to correct operation. However, the power management specifications require drivers to fully capture device state in software since devices can transition to an even lower power state where the device is powered off. In such cases, drivers must be able to restore their original state, following a full reset. Thus, suspend must store enough information to restore from any state.

In the case of drivers with persistent internal state, such as disks and other storage devices, restore will only restore the transient device state and not the persistent state, such as the contents of files. As a result, use of checkpoints must be coordinated with higher-level recovery mechanisms, such as Membrane [39], to keep persistent data consistent.

Fault Tolerance
Device recovery: Current recovery mechanisms require writing wrappers to track all device state and full device restart results in long latency.
OS functionality
Fast reboot: Restarting system requires probing all bus and device drivers.
NVM Operating systems: Providing persistent state of a running system requires ability to checkpoint a running device.
I/O Virtualization
Device consolidation: Re-assignment of passthrough devices across different VMs needs to wait for device initialization.
Live migration: Live migration of pass-through devices converts the millisecond latency of migrations to multiple seconds due to device initialization.
Clone VMs: Ability to launch many cloned VMs very quickly is limited by device initialization.

Table 1. Other uses for fast device state checkpointing.

Other uses for device state checkpointing. In addition to fault tolerance, device state checkpoints have other uses. Table 1 lists five possible uses. Within an operating system, checkpoints support fast reboot after upgrading system software by restoring device state from a checkpoint rather than reinitializing the driver. Similarly, operating systems using non-volatile memory to survive power failures [1, 29] can restart drivers from a checkpoint rather than reinitializing the device. In virtualized settings, pass-through and virtualization-aware devices [32] allow drivers in guest operating systems to interact with physical hardware. Device state checkpoints enable virtual-machine checkpoints to a passthrough device [24] and live migration, because the device state from the source can be extracted and restored on identical hardware at the destination. With virtual devices, the latency of live migration can be as low as 60ms [10], so a 2 second delay to initialize a device adds significant downtime. Finally, device state checkpointing enables dynamic fault tolerance at fine granularity as demonstrated by FGFT.

4.2 Recovery with checkpoints

We now describe how FGFT uses isolation and device checkpoints to perform recovery from failures. When a failure is detected, communication stubs call a recovery routine that is responsible for restoring correct driver operation.

Failure anticipation. To prepare for an eventual recovery, generated stubs create a device checkpoint before invoking a suspect entry point. They invoke the `checkpoint` routine. In addition, stubs for kernel functions log compensation entries to undo their effects in the kernel undo log. Driver state is not explicitly checkpointed; instead, suspect code operates on a copy of driver state as described in Section 3. In addition, the stub saves its processor register state, allowing a jump right into the stub if the driver fails.

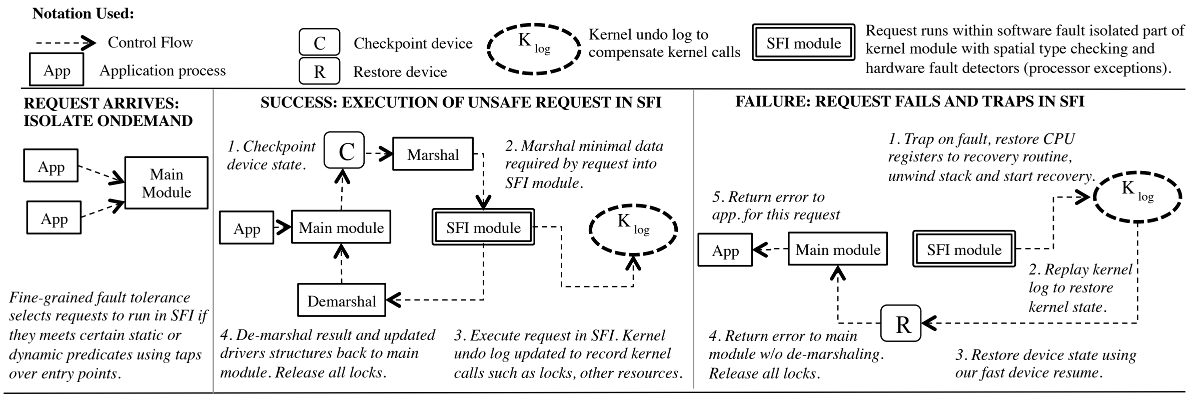

Notation Used:
Control Flow
App
Application process
C
Checkpoint device
R
Restore device
K log
Kernel undo log to compensate kernel calls
SFI module
Request runs within software fault isolated part of kernel module with spatial type checking and hardware fault detectors (processor exceptions).
REQUEST ARRIVES: ISOLATE ONDEMAND
App
App
Main Module
Fine-grained fault tolerance selects requests to run in SFI if they meets certain static or dynamic predicates using taps over entry points.
SUCCESS: EXECUTION OF UNSAFE REQUEST IN SFI
1. Checkpoint device state.
C
Marshal
2. Marshal minimal data required by request into SFI module.
App
Main module
SFI module
K log
Demarshal
4. De-marshal result and updated drivers structures back to main module. Release all locks.
3. Execute request in SFI. Kernel undo log updated to record kernel calls such as locks, other resources.
FAILURE: REQUEST FAILS AND TRAPS IN SFI
1. Trap on fault, restore CPU registers to recovery routine, unwind stack and start recovery.
K log
5. Return error to app. for this request
App
Main module
SFI module
2. Replay kernel log to restore kernel state.
R
4. Return error to main module w/o de-marshaling. Release all locks.
3. Restore device state using our fast device resume.

Figure 4. FGFT behavior during successful and failed entry point executions.

Recovery steps. In case a failure is detected by SFI or processor exceptions originating from suspect module, the recovery routine restores driver operation through a sequence of steps:

1. *Unwind thread.* If not already in the stub, the instruction pointer is set to the address of the recovery code in the entry point's stub, which reloads the saved registers. As noted above, nested calls to drivers are logically handled as separate transactions, so there is no need to unwind the thread to the outermost entry point.
2. *Restore device state checkpoint.* The stub recovery code calls the driver's `restore` routine to restore the device state.
3. *Free call state.* All temporary structures created for the suspect entry point call such as the range table, object tracker, and copies of kernel/driver structures are released.
4. *Release locks.* Any locks acquired before or during the call to the SFI driver are released, allowing other threads to execute.

If a driver entry point fails, the stub returns an appropriate error indicator, such as a NULL pointer or an error code, and relies on higher-level code to handle the failure. As only the single entry point fails, this failure has little impact on applications. All application state relating to the device, such as open handles, remain valid. Furthermore, other threads in the driver continue to run as soon as the recovery process completes and releases all acquired locks.

Compared to other driver isolation systems, the recovery process is much simpler because only one thread is affected, so other threads are not unwound. In addition, the driver state is left unmodified, so it is not saved and restored. Finally, device state is restored quickly from a checkpoint rather than by replaying a log. Hence, we see that checkpointing device state results in quicker and simpler recovery semantics for driver recovery.

4.3 Implementation effort

A key goal for FGFT is to reduce the implementation effort to isolate a driver. FGFT consists of minimal modifications to the kernel exception handlers (38 lines of code), a kernel module containing the object tracker, range table, and recovery support, and the Isolator tool in OCaml. The module is 1,200 lines of C code, and Isolator is 9700 lines of OCaml that implement: SFI isolation (400 lines), stub generation (7,800 lines), and static analysis of references to parameter fields (1,500 lines). In comparison, Nooks adds 23,000 lines of *kernel code* (85x more than FGFT) to isolate and reload device drivers and shadow drivers add another 1,100 lines of code for recovery. FGFT also does not require any wrappers around the driver interface. Nooks required 14,000 lines of manually written wrappers, which are hard to maintain as the kernel interface changes. FGFT's isolator tool automatically generates similar code for stubs.

5. Evaluation

We implemented device state checkpoint and fine-grained fault tolerance for the Linux 2.6.29 kernel for six drivers across three buses. All experiments were performed on a dual-core 3 GHz Intel Pentium D with 1GB of memory and an Intel 82541PI gigabit NIC. We note this is a somewhat old machine and a more recent machine would likely show lower performance costs. The evaluation examines the following aspects of FGFT:

1. *Fault Resilience.* What failures can fine-grained fault tolerance can handle? We evaluate FGFT using a series of fault injection experiments and report our results.
2. *Performance.* What is the performance loss of fine-grained fault tolerance on steady-state operation? We report the performance cost for applying FGFT on support and core I/O routines.
3. *Recovery Time.* What is the downtime caused by a driver failure? We compare the time taken by FGFT to restore the device and cleanup the failed driver thread state with the time taken to unload and reload a driver.
4. *Usefulness of FGFT.* Is selectively isolating entry points useful? We evaluate whether suspect entry points can be identified in drivers and whether they reduce the amount of code isolated.
5. *Device Checkpoint Support.* Is re-using existing power management functionality reasonable? We examine the frequency of power management support in existing drivers that facilitates device checkpoints.
6. *Developer Effort.* What is the overhead to the developer to enforce isolation in the system? We measure the effort needed to annotate a driver for isolation and add checkpoint/recovery code.

Unless otherwise specified, we compare FGFT against unmodified drivers running on an unmodified 2.6.29 Linux kernel.

5.1 Fault Resilience

We first evaluate how well FGFT can handle driver bugs using a combination of dynamic and static fault injection over six drivers. These tests evaluate both the ability of fine-grained fault tolerance to isolate and recover driver state as well as the ability of device state checkpointing to restore device functionality. Table 2 describes the types of faults inserted in the SFI module. Static fault injection modifies the driver source code to emulate programming

Fault Type	Description of fault
Corrupt pointers	Dynamic: Corrupt all pointers referenced in a function to random values.
Corrupt stack	Dynamic: corrupt execution stack by copying large chunks of data over stack variable addresses.
Corrupt expressions	Static: corrupt arithmetic instructions by adding invalid operations (like divide by zero).
Skip assignment	Static: remove assignment operations in a function.
Skip parameters	Dynamic: zero incoming parameters in a function.

Table 2. Faults injected to test failure resilience represent runtime and programming errors. Dynamic faults are inserted by invoking an `ioctl`**, and static faults by making an additional pass to inject faults while converting the driver to support FGFT.**

Driver	Injected Faults	Benign Faults	Native crashes	FGFT crashes
8139too	43	0	43	NONE
e1000	47	0	47	NONE
ens1371	36	0	36	NONE
pegasus	34	1	33	NONE
psmouse	22	1	21	NONE
r8169	46	0	46	NONE
Total	**258**	**2**	**256**	**NONE**

Table 3. Fault injection table with number of unique faults injected per driver. FGFT is able to correctly restore the driver state and device state in every case.

bugs, while dynamic fault injection modifies driver data while running to emulate run time errors. We perform a sequence of trials that test each fault site separately.

During each experiment, we run applications that use the driver to detect whether a driver failure causes the application to fail. For network, we use `ssh`, and `netperf`; for sound we use `aplay`, `arecord` from the ALSA suite. We tested the mouse by scrolling the device manually as we performed the fault injection experiments. After each injection experiment, we determine if there is an OS/driver crash or the application malfunctions. We re-invoke the failed entry point without the fault to ensure that it continues to work, and that resources such as locks have been appropriately released.

We injected a total of 258 unique faults in the native and FGFT drivers. Table 3 shows the number of faults injected for every driver and the outcome. For the native driver, all but two faults crashed the driver or resulted in kernel panics. The two benign faults were missing assignments.

In contrast, when we injected faults into driver entry points protected by FGFT, the driver and the OS remain available for every single fault. Furthermore, in every case, applications continue to execute correctly following the fault. For example, the sound application `aplay` skips for a few milliseconds during driver recovery but continued to play normally. The shell notes this disruption with the message "`ALSA buffer underrun`."

We also verify that internal driver and device state is correctly recovered using the `ethtool` interface for network drivers. We find that when failures happen during a call to change configuration settings, re-reading settings after a crash always returns the correct values.

Finally, we verify that changes to drivers made using non-class interfaces, such as the `proc` and `sys` file systems, before any failures persist. In contrast, shadow drivers cannot replay these actions since they cannot capture non-class driver interactions.

5.2 Performance

The primary cost of using FGFT is the time spent copying data in and out of the SFI module and creating device checkpoints. We measure performance with a gigabit Ethernet driver, as it may send or receive receive more than 75,000 packets per second. Thus, the overhead of FGFT will show up more clearly than on low-bandwidth devices.

We evaluate the runtime costs of using FGFT and regular versions of drivers in six configurations:

1. *Native:* Unmodified e1000 driver.
2. *FGFT* $_{static}$: Statically choose 75% of code (all off I/O-path) to isolate.
3. *FGFT* $_{dynamic}$: Dynamically choose at runtime whether to isolate off-I/O-path code.
4. *FGFT* $_{1/10}$: Isolate I/O path code for every 10th packet.
5. *FGFT* $_{all}$: Isolate I/O path code for every packet

The *dynamic* experiment measures the additional cost of choosing at runtime whether to invoke the regular or SFI version of a function. The *1/10* test measures the performance benefit of dynamically choosing whether to isolate based on parameters, such as packets from an untrusted process or destined for an untrusted network. Finally, the *all* test represents the worst case of invoking the SFI module on the I/O path for a high-bandwidth device.

We measure performance with netperf [19] by connecting our test machine to another machine with 1.2 GHz Intel Celeron processor and a Belkin NIC with a crossover cable. Table 4 reports the average of 5 runs.

In the *static* and *dynamic* tests where code off the I/O-path code is isolated, performance and CPU utilization are unaffected. These results demonstrate that FGFT achieves the goal of only imposing a cost on isolated code. The added cost of dynamically choosing to isolate is in the noise. For the *1/10* test that isolates entry points on the I/O path (the packet send routine) for every tenth packet, bandwidth dropped 0.6% and CPU utilization increased negligibly. Thus, selectively applying isolation, even on critical I/O paths, can have a small impact.

The performance only drops appreciably when we isolate critical path code on every request since we copy shared driver and kernel data across modules for *each* packet being transmitted. In the *all* test, the system consumes 100% of the CPU on one core (50% total CPU utilization), and bandwidth drops by 39%. This CPU utilization occurs because interrupts on other processors must spin until the copying and call completes, which drives up CPU utilization. FGFT is designed to limit isolation costs to specific requests and hence pays a cost of isolation because it needs to setup isolation (create copies) as each packet requests isolation. Isolation techniques that isolate the entire driver, such as Nooks [41] and SUD [3], may prove better for isolating I/O path code for high-performance devices because they copy less data per request. However, if the buggy code is off the I/O path, those techniques impose a much higher cost than FGFT does on I/O path.

Overall, these results show that FGFT performs well for the vast majority of devices that are low bandwidth, and can also be applied at low cost to high bandwidth devices off the I/O path. In addition, if the driver can dynamically detect which calls may be error prone to reduce the frequency of calls to the SFI driver, it can also be applied on the I/O path.

5.3 Recovery Time

A major benefit of checkpoint-based recovery is the speed of restoring service. Table 5 lists the time taken by the driver to recover using FGFT and by unloading and reloading the driver. We measure

Intel Pro/1000 gigabit NIC (E1000)

System	Throughput	CPU Utilization
Native	751.5 Mb/s	5.1%
FGFT $_{static}$	751.1 Mb/s	5.0%
FGFT $_{dynamic}$	751.1 Mb/s	5.2%
FGFT $_{1/10}$	745.3 Mb/s	5.4%
FGFT $_{all}$	454.0 Mb/s	50.0%

Table 4. TCP streaming send performance with netperf for regular and FGFT drivers with checkpoint based recovery.

Driver	Class	Bus	Restart recovery	FGFT recovery	Speedup
8139too	net	PCI	0.31s	70μs	4400
e1000	net	PCI	1.80s	295ms	6
r8169	net	PCI	0.12s	40μs	3000
pegasus	net	USB	0.15s	5ms	30
ens1371	sound	PCI	1.03s	115ms	9
psmouse	input	serio	0.68s	410ms	1.65

Table 5. Comparison of FGFT and restart based recovery times. Restart-based recovery requires additional time to replay logs running over the lifetime of the driver. FGFT does not affect concurrently executing threads in the system.

Driver	Class	Bus	Checkpoint times	Restore times
8139too	net	PCI	33μs	62μs
e1000	net	PCI	32μs	280ms
r8169	net	PCI	26μs	30μs
pegasus	net	USB	0μs	4ms
ens1371	sound	PCI	33μs	111ms
psmouse	input	serio	0μs	390ms

Table 6. Latency for device state checkpoint/restore.

recovery times by recording the time from detection of failure to completion of the restore routine. Overall, FGFT is between 1.6 and 4,400 times faster than restart recovery, and between 145ms and 1.5s faster. The drivers with the largest speedup have complicated probe routines that are avoided by restoring from a checkpoint. Hence, FGFT provides low-latency recovery and frequently offers an order-of-magnitude lower recovery latencies.

Checkpoint/restore latency. We examine the device checkpoint latencies to understand the source of our recovery performance in the previous section. Table 6 shows the latency of a checkpoint or restore for the same six drivers. Checkpointing is very fast, taking only 20μs on average and 33μs at worst. Thus, it is fast enough to be called frequently, such as before the invocation of most driver entry points. Restore times are longer, with a range from 30μs for the r8169 network driver to 390ms for psmouse. USB drivers store less state because the USB bus controller stores configuration information instead of the driver. Thus, during a normal resume, the bus restores configuration state before calling the driver to resume. The psmouse driver represents a legacy device and does not support suspend/resume. Instead, we re-use existing device code to reset the mouse.

5.4 Usefulness of being fine-grained

We evaluate whether selectively isolating specific entry points is useful by looking for evidence that driver bugs are confined to one or a few entry points. If the functions with bugs are reachable through a large number of entry points, then full driver isolation is more useful than per-entry point isolation. For example, if a bug occurs in a low level read routine, then the bug will affect a large number of entry points.

In order to have a large data set, we use a published list of hardware dependence bugs that represent one of the larger number of unfixed bugs in the drivers [11]. We were able to map these bugs in 210 drivers (541 total bugs) to our kernel under analysis. For each driver, we calculate the number of entry points and the fraction of code in the driver that must be isolated.

Class	Bus	Drivers reviewed	Drivers with PM	
net	PCI	104	68	65%
net	USB	32	27	84%
net	PCMCIA	4	4	100%
sound	PCI	72	63	88%
sound	USB	3	1	33%
sound	PCMCIA	2	2	100%
ATA	PCI	61	45	74%
SCSI	USB	1	1	100%
SCSI	PCMCIA	5	5	100%
Total	**-**	**284**	**216**	**76%**

Table 7. List of drivers with and without power management as analyzed with static analysis tools. USB devices (audio and storage) support hundreds of devices with a common driver, and provide support for suspend and resume.

We find that the bugs are reachable through 643 entry points, for an average of 3 per driver. As a comparison, these drivers have a total of 4,460 entry points (21 per driver), so only 14% of entry points must be isolated. The code reachable from these entry points comprises only 18% of the code in these drivers. These results indicate that at least some classes of driver bugs are confined to a single entry point, and therefore fine-grained fault tolerance can reduce the cost of fault tolerance as compared to isolating the entire driver.

5.5 Device Checkpoint Support

Device state checkpointing relies on existing power-management code. We measure how broadly it applies to Linux drivers by counting the number of drivers with power-management support. While modern ACPI-compliant systems require that all devices support power management, many legacy drivers do not.

We perform a simple static analysis over all network, sound, and storage drivers using the PCI, USB, and PCMCIA bus in Linux 2.6.37.6. The analysis scans driver entry points and identifies power management callbacks. Table 7 shows the number of drivers scanned by class and bus and the number that support power management.

Overall, we found that 76% of the drivers support power management. Of the drivers that do not support power management, most were either very old, from before Linux supported power management, or worked with very simple devices. Only two modern devices, both Intel 10gb Ethernet cards, did not provide suspend/resume. Thus, we find that nearly all modern devices support power management and can therefore support device state checkpointing.

5.6 Developer effort

The primary development cost in using FGFT is adding annotations, which describe how to copy data between the kernel and the SFI module. Table 8 shows the number of annotations needed to apply FGFT to every function in each of the tested drivers. We separate annotations into *driver annotations*, which are made to the driver code, and *kernel-header annotations*, which are a one-time effort common to all drivers in the class. These annotations are the incremental cost of making a driver fault tolerant, and the implementation effort of Isolator and the kernel code described in Section 4 are the up-front cost.

Overall, drivers averaged 20 annotations, with more annotations for drivers with more complex data structures. Most driver classes

Driver	Driver LoC	Isolation annotations	
		Driver Annotations	Kernel Annotations
8139too	1,904	15	20
e1000	13,973	32	
r8169	2,993	10	
pegasus	1,541	26	12
ens1371	2,110	23	66
psmouse	2,448	11	19

Table 8. Annotations required in FGFT isolation mechanisms for correct marshaling. Kernel annotations are common to a class, and driver annotations are specific to a single driver.

Driver	Recovery additions	
	LOC Moved	LOC Added
8139too	26	4
e1000	32	10
r8169	17	5
pegasus	22	4
ens1371	16	6
psmouse	19	6

Table 9. Developer effort for checkpoint/restore driver callbacks.

required 20 or fewer kernel-header annotations except for sound drivers, which have a more complex interface and required 66 annotations. Thus, the effort to annotate a driver is only modest, as annotations touch only a small fraction of driver code. In comparison, SafeDrive [45] changed 260 lines of code in the e1000 driver for isolation and another 270 lines for recovery. Nooks [41] required 23,000 lines of code to isolate and reload drivers. Thus, these small annotations to drivers may be much simpler than adding a large new subsystem to the kernel.

Checkpointing implementation. We evaluated the ease of implementing device state checkpointing by adding support to the six drivers listed in Table 9. For each driver we show the amount of code we copied from suspend/resume to create checkpoint/restore as well as the number of new lines added. On average, we moved 22 lines code and added six lines. The new code adds support for checkpoint/restore in interrupt contexts and avoids nested locks when the routines are invoked with a lock held. Even though device state checkpointing requires adding new code, the effort is mostly moving existing code. In comparison, implementing a shadow driver requires (i) building a model of driver behavior and (ii) writing a wrapper for every function in the driver/kernel interface to log state changes.

6. Related work

Fine-grained fault tolerance draws inspiration from past projects on driver reliability, program partitioning and software fault isolation systems.

Device driver recovery. Prior driver-recovery systems, including Nooks [41], Shadow drivers [40], SafeDrive [45] and Minix 3 [18] all unload and restart a failed driver. In contrast, FGFT takes a checkpoint prior to invoking the driver, and then rolls back to the most recent checkpoint, which is much faster. CuriOS provides transparent recovery and further ensures that client session state can be recovered [14]. However, CuriOS is a new operating system and requires specially written code to take advantage of its recovery system, while FGFT works with existing driver code in existing operating systems. ReViveI/O [28], and similar systems [34] provide whole-system checkpoint/restore by buffering I/O and only letting it reach the device after the next memory checkpoint. However, this approach does not work with polling, where I/O operations cannot be buffered and applied later.

Driver isolation systems. Driver isolation systems rely on hardware protection (Nooks [41] and Xen [15]) or strong in-situ detection mechanisms (BGI [6], LXFI [25] and XFI [42]) to detect failures in driver execution. However, in latter systems if the failure is detected *after* any state shared with the kernel has been modified then these systems cannot rollback to a last good state. Other driver isolation systems such as SUD [3] and Linux user-mode drivers [22], require writing class-specific wrappers in the kernel that are hard to maintain as the kernel evolves. FGFT differs from existing isolation systems by providing fine-grained isolation semantics and limits the runtime overheads only to suspect code. Prior work on fast software fault isolation, such as BGI [6], would benefit FGFT by improving its performance on I/O-intensive workloads.

Software fault tolerance. Existing SFI techniques use programmer annotations (SafeDrive [45] or API contracts (LXFI [25]) to provide type safety. XFI [42] transforms code binaries to provide inline software guards and stack protection. In contrast, FGFT operates on source code and allows drivers to operate on a copy of shared data. FGFT marshals the minimum required data and uses range hash to provide spatial safety.

Transactional kernels. FGFT executes drivers as a transaction by buffering their state changes until they complete. VINO [37] similarly encapsulated extensions with SFI and used a compensation log to undo kernel changes. However, VINO applied to an entire extension and did not address recovering device state. In addition, it terminated faulty extensions, while most users want to continue using devices following a failure. FGFT is complementary to other transactional systems such as TxLinux [35], that provide transactional semantics for system calls. These techniques could be applied to driver calls into the kernel instead of using a kernel undo log of compensation records.

Program Partitioning Program partitioning has been used for security [4, 7] and remote code execution [9]. Existing program partitioning tools statically partition user mode code or move driver code to user mode [17]. FGFT is the first system to partition programs within the kernel and is hence able to provide partitioning benefits to kernel specific components such as interrupt delivery and critical I/O path code. Furthermore, instead of partitioning code in any one domain, FGFT replicates its entry points and decides on a runtime basis whether a particular thread should run in isolation.

7. Conclusions

The performance and development costs of existing driver fault-tolerance mechanisms have restricted their adoption. In this paper, we presented fine-grained fault tolerance, a *pay-as-you-go* model for tolerating driver failures that can be dynamically invoked for specific requests. Fine-grained fault tolerance is made possible due to device checkpoints. This functionality is often considered to require a significant re-engineering of device drivers. However, we demonstrate that checkpoint functionality is already provided by existing suspend/resume code. While we only applied checkpoints to fault tolerance, there are more opportunities to use device checkpoint/restore, such as OS migration, fast reboot, and persistent operating systems that should be explored.

References

[1] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proc. of the Thirteenth IEEE HOTOS*, 2011.

[2] A. Birgisson, U. E. Mohan Dhawan, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspec-

tion. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, Oct. 2008.

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

[4] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. of the 13th USENIX Security Symposium*, 2004.

[5] S. Butt, V. Ganapathy, M. Swift, and C.-C. Chang. Protecting commodity OS kernels from vulnerable device drivers. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, Dec. 2009.

[6] 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.

[7] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. of the 21st ACM SOSP*, 2007.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proc. of the 18th ACM SOSP*, 2001.

[9] B. Chun and P. Maniatis. Augmented smartphone applications through clone cloud execution. In *Proceedings of the 12th conference on Hot topics in operating systems*. USENIX Association, 2009.

[10] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005.

[11] J. Corbet. Trusting the hardware too much. `http://lwn.net/Articles/479653/`. LWN February 2012.

[12] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Associates, Feb. 2005.

[13] M. Corp. Power management and ACPI - architecture and driver support. `msdn.microsoft.com/en-us/windows/hardware/gg463220`.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] R. Jones. Netperf: A network performance benchmark, version 2.1, 1995. Available at `http://www.netperf.org`.

[20] A. Kadav and M. M. Swift. Understanding modern device drivers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 3-7 2012.

[21] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[22] 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.

[23] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the conference on Programming language design and implementation*, PLDI, 2005.

[24] M. Mahalingam and R. Brunner. I/O Virtualization (IOV) For Dummies. `labs.vmware.com/download/80/`, 2007. VMworld 2007.

[25] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *SOSP*, 2011.

[26] P. Mochel. The Linux power management summit. `lwn.net/Articles/181888/`, 2006.

[27] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[28] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient handling of i/o in highly-available rollback-recovery servers. In *Proceedings of The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*

[29] D. Narayanan and O. Hodson. Whole-system persistence. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 3-7 2012.

[30] 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.

[31] V. Paxson. Bro: a system for detecting network intruders in real-time. In *Proceedings of the 7th conference on USENIX Security Symposium*, 1998.

[32] PCI-SIG. I/O virtualization. `http://www.pcisig.com/specifications/iov/`, 2007.

[33] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[34] P. Ramachandran. *Detecting and Recovering from In-Core Hardware Faults Through Software Anomaly Treatment*. PhD thesis, University of Illinois, Urbana-Champaign, 2011.

[35] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *Proc. of the 21st ACM Symp. on Operating System Principles*, Oct. 2007.

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

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

[38] Sun Microsystems. Opensolaris community: Fault management. `http://opensolaris.org/os/community/fm/`.

[39] Swaminathan Sundararaman et. al. Membrane: Operating system support for restartable file systems. In *Proceedings of the 8th Conference on File and Storage Technologies (FAST '10)*, 2010.

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

[41] 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.

[42] 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.

[43] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM SOSP*, Dec. 1993.

[44] 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.

[45] 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.