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

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.

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.

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

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.

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.

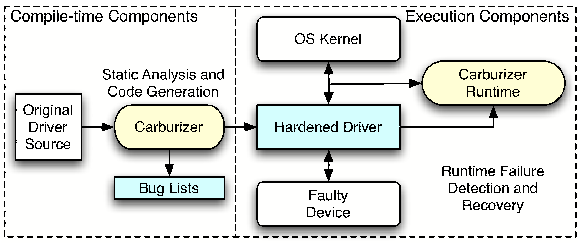

Figure 1: The Carburizer architecture. Existing kernel drivers are converted to hardened drivers and execute with runtime support for failure detection and recovery.

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 preprocessed 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 or aliased to the tainted variables. Carburizer considers the

```
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.

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.

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

```
1 static int amd8111e_read_phy(.......)
2 {
3   .
4   unsigned long long delta = (cpu/khz/HZ)*2;
5   unsigned long long _start = 0;
6   unsigned long long _cur = 0;
7   timeout = rdstcll(start) + delta ;
8   reg_val = readl(mmio + PHY_ACCESS);
9   while (reg_val & PHY_CMD_ACTIVE)  {
10    reg_val = readl( mmio + PHY_ACCESS );
11
12    if (_cur < timeout) {
13        rdstcll(_cur);
14    } else {
15      __shadow_recover();
16    }
17  .
18 }
```

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

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.

```
1 static void __init attach_pas_card(...)
2 {
3   .
4   if ((pas_model = pas_read(0xFF88)))
5   {
6     char  temp[100];
7
8   if ((int )pas_model < 0 ||
9       (int )pas_model >= 5) {
10      __shadow_recover();
11  }
12    sprintf(temp,
13      "%s rev %d",
14       pas_model_names[(int) pas_model],
15       pas_read(0x2789));
16    .
17 }
```

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 37 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 a wide variety of 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% $\pm$ 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 the 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 slight overhead of an unnecessary bounds

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)

check. More sophisticated analysis could remove these false positives.

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,

```
1 static int phy_reset(...) {
2   .
3   while (miicontrol & BMCR_RESET) {
4     msleep(10);
5     miicontrol = mii_rw(...);
6     if (tries++ > 100)
7       return -1;
8   }
9   .
10 }
```

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.

```
1 static int phy_reset(...) {
2   .
3   while (miicontrol & BMCR_RESET) {
4     msleep(10);
5     miicontrol = mii_rw(...);
6     if (tries++ > 100) {
7       printk("...");
8       return -1;
9   }
10  }
11  .
12 }
```

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.

then we assume its report is sufficient and Carburizer does not introduce additional reporting.

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.

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

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.

(according to our manual analysis), whether reported or not, compared with the number of errors reported by 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.