Understanding and Improving Device Access Complexity

Asim Kadav
(with Prof. Michael M. Swift)
University of Wisconsin-Madison
Devices enrich computers

- Keyboard
- Sound
- Printer
- Network
- Storage

- Keyboard
- Flash storage
- Graphics
- WiFi
- Headphones
- SD card
- Camera
- Accelerometers
- GPS
- Touch display
- NFC
Huge growth in number of devices

New I/O devices: accelerometers, GPUS, GPS, touch

Many buses: USB, PCI-e, thunderbolt

Heterogeneous OS support: 10G ethernet vs card readers
Device drivers: OS interface to devices

- Applications
- OS
- Buses
- Devices

Exposure: device abstractions and hide device complexity

Exposure: kernel abstractions and hide OS complexity

Allow diverse set of applications and OS services to access diverse set of devices
Evolution of devices hurts device access

Tools and mechanisms to address increasing device complexity

- Growth in number and diversity
- Run in challenging environments
- Hardware failures (like CMOS issues)
- Complex firmware and configuration modes

Simplicity
Reliability
Low latency
Cost effective

Efficient device support in OS
Growth in drivers hurts understanding of drivers.

Understand the software complexity and improve driver code.
Last decade: Focus on the driver-kernel interface

Recipe for disaster

3rd party developers + device drivers = OS kernel
### Improvement System Validation

<table>
<thead>
<tr>
<th>Improvement</th>
<th>System</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Drivers</td>
</tr>
<tr>
<td>New functionality</td>
<td>Shadow driver migration [OSR09]</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RevNIC [Eurosys 10]</td>
<td>1</td>
</tr>
<tr>
<td>Reliability</td>
<td>Nooks [SOSP 03]</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>XFI [OSDI 06]</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CuriOS [OSDI 08]</td>
<td>2</td>
</tr>
<tr>
<td>Type Safety</td>
<td>SafeDrive [OSDI 06]</td>
<td>6</td>
</tr>
</tbody>
</table>

**Limited kernel changes + Applicable to lots of drivers => Real Impact**

**Design goal: Complete solution that limits kernel changes and applies to all drivers**
Goal: Address software and hardware complexity

★ Understand and improve device access in the face of rising hardware and software complexity

1. Increasing hardware complexity
   - Reliability against hardware failures

2. Increasing hardware complexity
   - Low latency device availability

3. Increasing software complexity
   - Better understanding of driver code
My approach

Narrow approach and solve specific problems in all drivers

Tolerate device failures

Broad approach and have a holistic view of all drivers

Understand drivers and potential opportunities

Known approach and apply to all drivers

Transactional approach for low latency recovery

Minimize kernel changes and apply to all drivers
Contributions/Outline

First research consideration of hardware failures in drivers
- Tolerate device failures
  - SOSP '09

Largest study of drivers to understand their behavior and verify research assumptions
- Understand drivers and potential opportunities
  - ASPLOS '12

Introduce checkpoint/restore in drivers for low latency fault tolerance
- Transactional approach for low latency recovery
  - ASPLOS '13
What happens when devices misbehave?

- Drivers make it better
  - Provide error detection and recovery
- Drivers make it worse
  - Assume perfect hardware or panic when anything bad happens

What assumptions do modern drivers make about hardware?
Current state of OS-hardware interaction

- Many device drivers often assume device perfection
  - Common Linux network driver: 3c59x.c

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

Hardware dependence bug: Device malfunction can crash the system
Sources of hardware misbehavior

- Firmware/Design bugs
- Device wear-out, insufficient burn-in
- Bridging faults
- Electromagnetic interference, radiation, heat
Sources of hardware misbehavior

- Firmware/Design bugs
- Device wear-out, insufficient burn-in
- Bridging faults
- Electromagnetic interference, radiation, heat

Results of misbehavior

- Corrupted/stuck-at inputs
- Timing errors
- Interrupt storms/missing interrupts
- Incorrect memory access
Transient hardware failures caused 8% of all crashes and 9% of all unplanned reboots [1]
★ Systems work fine after reboots
★ Vendors report returned device was faultless

Existing solution is hand-coded hardened drivers
★ Crashes reduce from 8% to 3%

How do hardware dependence bugs manifest?

1. Drivers use device data in critical control and data paths
   ```c
   printk("%s", msg[inb(regA)]);
   ```

2. Drivers do not report device malfunction to system log
   ```c
   if (inb(regA)! = 5) {
       return -22; // do nothing
   }
   ```

3. Drivers do not detect or recover from device failures
   ```c
   if (inb(regA)! = 5) {
       panic();
   }
   ```
## Vendor recommendations for driver developers

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Recommended by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Intel</td>
</tr>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>⬤</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>⬤</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>⬤</td>
</tr>
<tr>
<td>Timing</td>
<td>Infinite polling</td>
<td>⬤</td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td>⬤</td>
</tr>
<tr>
<td>Reporting</td>
<td>Report all failures</td>
<td>⬤</td>
</tr>
<tr>
<td>Recovery</td>
<td>Handle all failures</td>
<td></td>
</tr>
</tbody>
</table>

**Goal:** Automatically implement as many recommendations as possible in commodity drivers.
Goal: Tolerate hardware device failures in software through hardware failure detection and recovery

**Static analysis component**
- Detect and fix hardware dependence bugs
- Detect and generate missing error reporting information

**Runtime component**
- Detect interrupt failures
- Provide automatic recovery
Hardening drivers

- **Goal: Remove hardware dependence bugs**
  - Find driver code that uses data from device
  - Ensure driver performs validity checks

- **Carburizer detects and fixes hardware bugs:**
  - Infinite polling
  - Unsafe array reference
  - Unsafe pointer reference
  - System panic calls
Finding sensitive code

★ First pass: Identify tainted variables that contain data from device

```c
int test() {
    a = readl();
    b = inb();
    c = b;
    d = c + 2;
    return d;
}
```

```c
int set() {
    e = test();
}
```

Types of device I/O

★ Port I/O: inb/inw
★ Memory-mapped I/O: readl/readw
★ DMA buffers
★ Data from USB packets

Tainted Variables

OS

network card
Detecting risky uses of tainted variables

★ Second pass: Identify risky uses of tainted variables

★ Example: Infinite polling
  ★ Driver waiting for device to enter particular state
  ★ Solution: Detect loops where all terminating conditions depend on tainted variables
  ★ Extra analyses to existing timeouts
Infinite polling

★ Infinite polling of devices can cause system lockups

```c
static int amd8111e_read_phy(........)
{
    ...
    reg_val = readl(mmio + PHY_ACCESS);
    while (reg_val & PHY_CMD_ACTIVE)
        reg_val = readl(mmio + PHY_ACCESS);
    ...
}
```

AMD 8111e network driver(amd8111e.c)
Hardware data used in array reference

- Tainted variables used as array indexes
- Detect existing range/not NULL checks

```c
static void __init attach_pas_card(...) {
    if ((pas_model = pas_read(0xFF88))) {
        ...
        sprintf(temp, "%s rev %d",
            pas_model_names[(int) pas_model], pas_read(0x2789));
        ...
    }
}
```

Pro Audio Sound driver (pas2_card.c)
Analysis results over the Linux kernel

<table>
<thead>
<tr>
<th>Driver class</th>
<th>Infinite polling</th>
<th>Static array</th>
<th>Dynamic array</th>
<th>Panic calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>117</td>
<td>2</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>scsi</td>
<td>298</td>
<td>31</td>
<td>22</td>
<td>121</td>
</tr>
<tr>
<td>sound</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>video</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td>381</td>
<td>9</td>
<td>57</td>
<td>32</td>
</tr>
<tr>
<td>Total</td>
<td>860</td>
<td>43</td>
<td>89</td>
<td>179</td>
</tr>
</tbody>
</table>

Lightweight and usable technique to find hardware dependence bugs

- Analyzed/Built 6300 driver files (2.8 million LOC) in 37 min
- Found **992** hardware dependence bugs in driver code
- False positive rate: 7.4% (manual sampling of 190 bugs)
Repairing drivers

Call recovery service

- Timeout checks
- Array bounds check
- Not null checks

- Infinite polling
- Unsafe array reference
- Unsafe pointer reference
- System panic calls
Runtime fault recovery: Shadow drivers

- Carburizer calls generic recovery service if check fails
- Low cost transparent recovery
  * Based on shadow drivers
  * Records state of driver at all times
  * Transparently restarts and replays recorded state on failure
- No isolation required (like Nooks)

Swift [OSDI ’04]
Carburizer automatically fixes infinite loops

```c
timeout = rdtsc11(start) + (cpu/khz/HZ)*2;
reg_val = readl(mmio + PHY_ACCESS);
while (reg_val & PHY_CMD_ACTIVE) {
    reg_val = readl(mmio + PHY_ACCESS);
    if (_cur < timeout)
        rdtsc11(_cur);
    else
        __recover_driver();
}
```

*Code simplified for presentation purposes*
Carburizer automatically adds bounds checks

```c
static void __init attach_pas_card(...) {
    if ((pas_model = pas_read(0xFF88))) {
        ... 
        if ((pas_model< 0)) || (pas_model>= 5))
            __recover_driver();
        ...
        sprintf(temp, "%s rev %d", 
            pas_model_names[(int) pas_model], pas_read(0x2789));
    }
}
```

*Code simplified for presentation purposes*
## Fault injection and performance

**★ Synthetic fault injection on network drivers**

<table>
<thead>
<tr>
<th>Device/Driver</th>
<th>Original Driver</th>
<th>Carburizer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Behavior</td>
<td>Detection</td>
</tr>
<tr>
<td>3COM 3C905</td>
<td>CRASH</td>
<td>None</td>
</tr>
<tr>
<td>DEC DC 21x4x</td>
<td>CRASH</td>
<td>None</td>
</tr>
</tbody>
</table>

**★ < 0.5% throughput overhead and no CPU overhead with network drivers**

Carburizer failure detection and transparent recovery works and has very low overhead
## Summary

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Intel</th>
<th>Sun</th>
<th>MS</th>
<th>Linux</th>
<th>Carburizer Ensures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Validation</strong></td>
<td>Input validation</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td>Infinite polling</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td></td>
<td>✅</td>
<td></td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
<td></td>
<td>✅</td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td></td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reporting</strong></td>
<td>Report all failures</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Carburizer improves system reliability by automatically ensuring that hardware failures are tolerated in software**
Contributions beyond research

- Linux Plumbers Conference [Sep ‘11]
- LWN Article with paper & list of bugs [Feb ‘12]
- Released patches to the Linux kernel
Recovery performance: device initialization is slow

★ **Multi-second device probe**
  ★ Identify device
  ★ Cold boot device
  ★ Setup device/driver structures
  ★ Configuration/Self-test

★ **What does slow device re-initialization hurt?**
  ★ Fault tolerance: Driver recovery
  ★ Virtualization: Live migration, cloning
  ★ OS functions: Boot, upgrade
Recovery functionality: assumes drivers follow class behavior

- Kernel exports standard entry points for every class (like “packet send” for network class)
- Shadow drivers records state by interposing class defined entry points
- Recovery = Restart and replay of captured state
- Do drivers have additional state?

How many drivers obey class behavior?
Tolerate device failures

Understand drivers and potential opportunities

Transactional approach for cheap recovery

Overview
Recovery specific results
Our view of drivers is narrow

Drivers
6.7 million LOC in Linux

Necessary to review driver code in modern settings

Driver Research (avg. 2.2 drivers/system)

Bugs
Understanding Modern Device Drivers

Study source of all Linux drivers for x86 (~3200 drivers)

- **Driver properties**
  - Code properties
  - Verify research assumptions

- **Driver interaction**
  - Driver kernel & device interaction
  - Driver architecture

- **Driver similarity**
  - 7 million lines of code needed?
Study methodology

★ Static source analysis of 3200 drivers in Linux 2.6.37.6 (May 2011)

★ Identify driver entry points, kernel and bus callouts
  ★ Device class, sub-class, chipsets
  ★ Bus properties & other properties (like module params)
  ★ Driver functions registered as entry points (purpose)

For every driver

Driver entry points

xmit
open
probe
close
Study methodology

★ Static source analysis of 3200 drivers in Linux 2.6.37.6 (May 2011)

★ Identify driver entry points, kernel and bus callouts

★ Reverse propagate information to aggregate bus, device and kernel behavior
Study methodology

★ **Static source analysis of 3200 drivers in Linux 2.6.37.6 (May 2011)**

- Identify driver wide and function specific properties of all drivers

- Reverse propagate information to aggregate bus, device and kernel behavior

- Use statistical clustering techniques and static analysis to identify similar code
Contributions/Outline

- Tolerate device failures
- Understand drivers and potential opportunities
- Transactional approach for cheap recovery

Overview
- Recovery specific results
### Driver Code Characteristics

- **Initialization/cleanup** – 36%
- **Core I/O & interrupts** – 23%
- **Device configuration** – 15%
- **Power management** – 7.4%
- **Device ioctl** – 6.2%

Initialization code dominates driver LOC and adds to complexity.
Problem 2: Shadow drivers assume drivers follow class behavior

Class definition includes:
- Callbacks registered with the bus, device and kernel subsystem

How many drivers follow class behavior and how much code does this add?
Problem 2(a): Drivers do behave outside class definitions

★ Non-class behavior in device drivers:
- module parameters, unique ioctl, procfs/sysfs interactions

Overall 44% of drivers have non-class behavior and research making this assumption will not apply

$ echo 1 > /sys/class/sound/mixer/device/enable

Windows WLAN card config via private ioctl

Linux sound card config via sysfs

Overall 44% of drivers have non-class behavior and research making this assumption will not apply
Class-specific driver recovery leads to a large kernel recovery subsystem
Few other results

**Driver properties**

- Many assumptions made by driver research does not hold:
  - 44% of drivers do not obey class behavior
  - 15% drivers perform significant processing
  - 28% drivers support multiple chipsets

**Driver interactions**

- USB bus offers efficient access (as compared to PCI, Xen)
  - Supports high # devices/driver (standardized code)
  - Coarse-grained access

**Driver similarity**

- 400,000 lines of code similar to code elsewhere and ripe for improvement via:
  - Procedural abstractions
  - Better multiple chipset support
  - Table driver programming

★ More results in “Understanding Modern Device Drivers” ASPLOS 2012
Limitations of restart/replay recovery

- Device save/restore limited to restart/replay
  - Slow: Device initialization is complex (multiple seconds)
  - Incomplete: Unique device semantics not captured
  - Hard: Need to be written for every class of drivers
  - Large changes: Introduces new, large kernel subsystem

Checkpoint/restore of device and driver state removes the need to reboot device and replay state
Checkpointing drivers is hard

- Easy to capture *memory* state

Intuition: Operating systems already capture device state during power management

- Device state is not captured
  - Device configuration space
  - Internal device registers and counters
  - Memory buffer addresses used for DMA
- Unique for every device
Intuition with power management

- Refactor power management code for device checkpoints
  - Correct: Developer captures unique device semantics
  - Fast: Avoids probe and latency critical for applications

- Ask developers to export checkpoint/restore in their drivers
Device checkpoint/restore from PM code

Checkpoint

- Save config state
- Save register state
- Disable device
- Save DMA state
- Suspend

Restore

- Restore config state
- Restore register state
- Restore or reset DMA state
- Re-attach/Enable device
- Device Ready

Suspend/resume code provides device checkpoint functionality
Fine-Grained Fault Tolerance

- **Goal:** Improve driver recovery with minor changes to drivers
- **Solution:** Run drivers as transactions using device checkpoints

### Device state
- Developers export checkpoint/restore in drivers

### Driver state
- Run drivers invocations as memory transactions
- Use source transformation to copy parameters and run on separate stack

### Execution model
- Checkpoint device
- Execute driver code as memory transactions
- On failure, rollback and restore device
- Re-use existing device locks in the driver
Adding transactional support to drivers

Driver with checkpoint support

Source transformation (adds driver transactions)

User supplied annotations

Main driver module

SFI driver module

Object tracking

Marshaling/Demarshaling

Kernel undo log

Communication and recovery support

SFI = software fault isolated

Static modifications

Run-time support
Transaction execution of drivers

- Detects and recovers from:
  - Memory errors like invalid pointer accesses
  - Structural errors like malformed structures
  - Processor exceptions like divide by zero, stack corruption

<table>
<thead>
<tr>
<th>Address</th>
<th>Access rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffa000</td>
<td>Read</td>
</tr>
<tr>
<td>0xffffa008</td>
<td>Write</td>
</tr>
<tr>
<td>0xffffa00a</td>
<td>Read</td>
</tr>
</tbody>
</table>

Kernel Log alloc

C

get ringparam
probe
xmit
get config

netdev
network driver

netdev->priv->tx_ring
netdev->priv->rx_ring

result

SFI network driver

Range Table

★ Detects and recovers from:
★ Memory errors like invalid pointer accesses
★ Structural errors like malformed structures
★ Processor exceptions like divide by zero, stack corruption
FGFT: Failed transactions

FGFT provides transactional execution of driver entry points

SFI network driver

netdev->priv->tx_ring

netdev->priv->rx_ring

Address Access rights
0xffffa000 Read
0xffffa008 Write
0xffffa00a Read

Kernel Log alloc

Range Table

get config

probe

xmit

get ringparam

C

R
How does this give us transactional execution?

- **Atomicity: All or nothing execution**
  - Driver state: Run code in SFI module
  - Device state: Explicitly checkpoint/restore state

- **Isolation: Serialization to hide incomplete transactions**
  - Re-use existing device locks to lock driver
  - Two phase locking

- **Consistency: Only valid (kernel, driver and device) states**
  - Higher level mechanisms to rollback external actions
  - At most once device action guarantee to applications
Evaluation platform

★ **Criterion:**
- Latency of recovery: How fast is it?
- Correctness of recovery: How well does it work?
- Incremental effort: How much work is it?
- Performance: How much does it cost?

★ **Operating system:**
- Linux 2.6.29
- Six drivers across three buses

★ **Hardware:**
- Results on 2.5 GHz Intel Core 2 Quad system configured with 4 GB DDR2 DRAM
FGFT provides significant speedup in driver recovery and improves system availability.
## Static and dynamic fault injection

<table>
<thead>
<tr>
<th>Driver</th>
<th>Injected Faults</th>
<th>Native Crashes</th>
<th>FGFT Crashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8139too</td>
<td>43</td>
<td>43</td>
<td>NONE</td>
</tr>
<tr>
<td>e1000</td>
<td>47</td>
<td>47</td>
<td>NONE</td>
</tr>
<tr>
<td>r8169</td>
<td>36</td>
<td>36</td>
<td>NONE</td>
</tr>
<tr>
<td>pegasus</td>
<td>34</td>
<td>33</td>
<td>NONE</td>
</tr>
<tr>
<td>ens1371</td>
<td>22</td>
<td>21</td>
<td>NONE</td>
</tr>
<tr>
<td>psmouse</td>
<td>46</td>
<td>46</td>
<td>NONE</td>
</tr>
<tr>
<td>TOTAL</td>
<td>258</td>
<td>256</td>
<td>NONE</td>
</tr>
</tbody>
</table>

FGFT survives multiple static and dynamic faults without aborting other threads in the driver.
FGFT requires limited programmer effort and needs only 38 lines of new kernel code.
Throughput with isolation and recovery

FGFT can isolate and recover high bandwidth devices at low overhead without adding kernel subsystems

netperf on Intel quad-core machines

Throughput %age (Baseline 844 Mbps)

CPU: 2.4%  3.4%  2.4%  2.9%

Native
FGFT-I/O-all
FGFT-off-I/O
FGFT-I/O-1/2

0  25  50  75  100

0  50  75  100

netperf on Intel quad-core machines
Talk summary

First research consideration of hardware failures in drivers
- Released tool, patches & informed developers

Largest study of drivers to understand their behavior and verify research assumptions
-Measured driver behavior & identified new directions

Introduced checkpoint/restore in drivers for low latency fault tolerance
- Fast & correct recovery with incremental changes to drivers
Thank you

Asim Kadav
★ http://cs.wisc.edu/~kadav
★ kadav@cs.wisc.edu