Why study device drivers?

» Linux drivers constitute ~5 million LOC and 70% of kernel
   » Little exposure to this breadth of driver code from research
   » Better understanding of drivers can lead to better driver model

» Large code base discourages major changes
   » Hard to generalize about driver properties
   » Slow architectural innovation in driver subsystems

» Existing architecture: Error prone drivers
   » Many developers, privileged execution, C language
   » Recipe for complex system with reliability problems

Our view of drivers is narrow

» Driver research is focused on reliability
   » Focus limited to fault/bug detection and tolerance
   » Little attention to architecture/structure

» Driver research only explores a small set of drivers
   » Systems evaluate with mature drivers
   » Volume of driver code limits breadth

» Necessary to review current drivers in modern settings

Difficult to validate research on all drivers
Difficult to validate research on all drivers

“...Please do not misuse these tools! (Coverity).... If you focus too much on fixing the problems quickly rather than fixing them cleanly, then we forever lose the opportunity to clean our code, because the problems will then be hidden.”

Understanding Modern Device Drivers

- Study source of all Linux drivers for x86 (~3200 drivers)
- Understand properties of driver code
  - What are common code characteristics?
  - Do driver research assumptions generalize?
- Understand driver interactions with outside world
  - Can drivers be easily re-architected or migrated?
  - Can we develop more efficient fault-isolation mechanisms?
- Understand driver code similarity
  - Do we really need all 5 million lines of code?
  - Can we build better abstractions?

Methodology of our study

- Target Linux 2.6.37.6 (May 2011) kernel
- Use static source analyses to gather information
- Perform multiple dataflow/control-flow analyses
  - Detect driver properties of the code
  - Detect driver code interactions with environment
  - Detect driver code similarities within classes
Extract driver wide properties for individual drivers

Determine code characteristics of each driver function

Determining interactions of each driver function

Outline

- Methodology
- Driver code characteristics
- Driver interactions
- Driver redundancy
Part 1: Driver Code Behavior

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

Driver code complexity and size is assumed to be a result of its I/O function.

1-a) Driver Code Characteristics

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

Only 23% of driver code is dedicated to I/O and interrupts

Driver code complexity stems mostly from initialization/cleanup code.
1-b) Do drivers belong to classes?

- Drivers registers a class interface with kernel
  - Example: Ethernet drivers register with bus and net device library

- Class definition includes:
  - Callbacks registered with the bus, device and kernel subsystem
  - Exported APIs of the kernel to use kernel resources and services

- Most research assumes drivers obey class behavior

---

### Class definition used to record state

- Modern research assumes drivers conform to class behavior
- Example: Driver recovery (Shadow drivers [6])

- Driver state is recorded based on interfaces defined by class
- State is replayed upon restart after failure to restore state

---

### Class definition used to infer driver behavior

- Example: Reverse engineering of drivers - RevNIC [Eurosys 10]

- Driver behavior is reverse engineered based on interfaces defined by class
- Driver entry points are invoked to record driver operations
- Code is synthesized for another OS based on this behavior

---

Driver Code Characteristics

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

Better ways needed to manage device configuration code

---

Non-class behavior can lead to incomplete restore after failure

Non-class behavior can lead to incomplete reverse engineering of device driver behavior
Do drivers belong to classes?

- Non-class behavior stems from:
  - Load time parameters, unique ioctls, procfs and sysfs interactions

```c
... qlnic_sysfs_write_esw_config (...) {
  ...
  switch (esw_cfg[i].op_mode) {
    case QLCNIC_PORT_DEFAULTS:
      qlnic_set_eswitch(..., &esw_cfg[i]);
    ...
    case QLCNIC_ADD_VLAN:
      qlnic_set_vlan_config(..., &esw_cfg[i]);
    ...
    case QLCNIC_DEL_VLAN:
      esw_cfg[i].vlan_id = 0;
      qlnic_set_vlan_config(..., &esw_cfg[i]);
    ...
  }

Drivers/net/qlnic/qlnic_main.c: QLogic driver (network class)
```

Many drivers do not conform to class definition

- Results as measured by our analyses:
  - 16% of drivers use proc/sysfs support
  - 36% of drivers use load time parameters
  - 16% of drivers use ioctl that may include non-standard behavior

- Breaks systems that assume driver semantics can be completely determined from class behavior

Overall, 44% of drivers do not conform to class behavior.
Systems based on class definitions may not work properly when such non-class extensions are used.

1-c) Do drivers perform significant processing?

- Drivers are considered only a conduit of data
- Example: Synthesis of drivers (Termitel\textsuperscript{SOSP09})
  - State machine model only allows passing of data
  - Does not support transformations/processing
- But: drivers perform checksums for RAID, networking, or calculate display geometry data in VMs

Instances of processing loops in drivers

- Detect loops in driver code that:
  - do no I/O
  - do not interact with kernel
  - lie on the core I/O path

```c
static u8 e1000e_calculate_checksum(...) {
  u32 i;
  u8 sum = 0;
  ...
  for (i = 0; i < length; i++)
    sum += buffer[i];
  return (u8) (0 - sum);

Drivers/net/e1000e/lib.c: e1000e network driver
```
Many instances of processing across classes

```c
static void _cx18_process_vbi_data(...) {
  // Process header & check endianness
  // Obtain RAW and sliced VBI data
  // Compress data, remove spaces, insert mpg info.
}
```

Drivers do perform processing of data

- Processing results from our analyses:
  - 15% of all drivers perform processing
  - 28% of sound and network drivers perform processing

- Driver behavior models should include processing semantics
  - Implications in automatic generation of driver code
  - Implications in accounting for CPU time in virtualized environment

Driver behavior models should consider processing

Part 2: Driver interactions

- a) What are the opportunities to redesign drivers?
  - Can we learn from drivers that communicate efficiently?
  - Can driver code be moved to user mode, a VM, or the device for improved performance/reliability?

- b) How portable are modern device drivers?
  - What are the kernel services drivers most rely on?

- c) Can we develop more efficient fault-tolerance mechanisms?
  - Study drivers interaction with kernel, bus, device, concurrency
2-a) Driver kernel interaction

Common drivers invoking device specific routines reduces driver code significantly (and more classes can benefit)

2-b) Driver-bus interaction

Many classes are portable: Limited interaction with device library and kernel services

Driver kernel interaction

- Compare driver structure across buses
- Look for lessons in driver simplicity and performance
- Can they support new architectures to move drivers out of kernel?
  - Efficiency of bus interfaces (higher devices/driver)
    - Interface standardization helps move code away from kernel
  - Granularity of interaction with kernel/device when using a bus
    - Coarse grained interface helps move code away from kernel
### PCI drivers: Fine grained & few devices/driver

<table>
<thead>
<tr>
<th>BUS</th>
<th>Kernel Interactions (network drivers)</th>
<th>Device Interactions (network drivers)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mem</td>
<td>sync</td>
</tr>
<tr>
<td>PCI</td>
<td>29.3</td>
<td>35.4</td>
</tr>
</tbody>
</table>

- PCI drivers have fine grained access to kernel and device
  - Support low number of devices per driver (same vendor)
  - Support performance sensitive devices
  - Provide little isolation due to heavy interaction with kernel
  - Extend support for a device with a completely new driver

### USB: Coarse grained & higher devices/driver

<table>
<thead>
<tr>
<th>BUS</th>
<th>Kernel Interactions (network drivers)</th>
<th>Device Interactions (network drivers)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mem</td>
<td>sync</td>
</tr>
<tr>
<td>PCI</td>
<td>29.3</td>
<td>35.4</td>
</tr>
<tr>
<td>USB</td>
<td>24.5</td>
<td>37.7</td>
</tr>
</tbody>
</table>

- USB devices support far more devices/driver
  - Bus offers significant functionality enabling standardization
  - Simpler drivers (like, DMA via bus) with coarse grained access
  - Extend device specific functionality for most drivers by only providing code for extra features

* accessed via bus

### Xen: Extreme standardization, limit device features

<table>
<thead>
<tr>
<th>BUS</th>
<th>Kernel Interactions (network drivers)</th>
<th>Device Interactions (network drivers)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mem</td>
<td>sync</td>
</tr>
<tr>
<td>PCI</td>
<td>29.3</td>
<td>35.4</td>
</tr>
<tr>
<td>USB</td>
<td>24.5</td>
<td>37.7</td>
</tr>
<tr>
<td>Xen</td>
<td>21.0</td>
<td>2.7</td>
</tr>
</tbody>
</table>

- Xen represents extreme in device standardization
  - Xen can support very high number of devices/driver
  - Device functionality limited to a set of standard features
  - Non-standard device features accessed from domain executing the driver

**Efficient remote access to devices and efficient device driver support offered by USB and Xen**

---

### Outline

- Methodology
- Driver code characteristics
- Driver interactions
- Driver redundancy
Part 3: Can we reduce the amount of driver code?

- Are 5 million lines of code needed to support all devices?
- Are there opportunities for better abstractions?
- Better abstractions reduce incidence of bugs
- Better abstractions improve software composability

Goal: Identify the missing abstraction types in drivers

- Quantify the savings by using better abstractions
- Identify opportunities for improving abstractions/interfaces

Finding out similar code in drivers

Drivers within subclasses often differ by reg values

```
... nv_mcp55_thaw(...) {
    void __lomes *min_base = ap->host->iomap[NV_MMIO_BAR];
    int shift = ap->port_no * NV_INT_PORT_SHIFT_MCP55;
    ... writel(NV_INT_ALL << shift, min_base+NV_INT_STATUS_MCP55);
    mask = readl(min_base + NV_INT_ENABLE_MCP55);
    mask |= (NV_INT_MASK_MCP55 << shift);
    writel(mask, min_base + NV_INT_ENABLE_MCP55);
}

... nv_ck804_thaw(...) {
    void __lomes *min_base = ap->host->iomap[NV_MMIO_BAR];
    int shift = ap->port_no * NV_INT_PORT_SHIFT;
    ... writel(NV_INT_ALL << shift, min_base+NV_INT_STATUS_CK804);
    mask = readb(min_base + NV_INT_ENABLE_CK804);
    mask |= (NV_INT_MASK_CK804 << shift);
    writeb(mask, min_base + NV_INT_ENABLE_CK804);
}
```

Drivers/ata/sata_nv.c

Wrappers around device/bus functions

```
static int nv_pre_reset(...) {
    struct pci_bits
    nv_enable_bits[] = {
        { 0x50, 1, 0x02, 0x02 },
        { 0x50, 1, 0x01, 0x01 }
    };
    struct ata_port *
    ap = link->ap;
    struct pci_dev *
    pdev = to_pci_dev(...);
    if (!pci_test_config_bits(pdev, &nv_enable_bits[ap->port_no]))
        return -ENODS;
    return ata_sff_prereset(...);
}
```

Drivers/ata/pata_amd.c

```
static int amd_pre_reset(...) {
    struct pci_bits
    amd_enable_bits[] = {
        { 0x40, 1, 0x02, 0x02 },
        { 0x40, 1, 0x01, 0x01 }
    };
    struct ata_port *
    ap = link->ap;
    struct pci_dev *
    pdev = to_pci_dev(...);
    if (!pci_test_config_bits(pdev, &amd_enable_bits[ap->port_no]))
        return -ENODS;
    return ata_sff_prereset(...);
}
```

Drivers/ata/pata_amd.c
### Significant opportunities to improve abstractions

- At least 8% of all driver code is similar to other code

<table>
<thead>
<tr>
<th>Sources of redundancy</th>
<th>Potential applicable solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls to device/bus with different register values</td>
<td>Table/data driven programming models</td>
</tr>
<tr>
<td>Wrappers around kernel/device library calls</td>
<td>Procedural abstraction for device classes</td>
</tr>
<tr>
<td>Code in family of devices from one vendor</td>
<td>Layered design/subclass libraries</td>
</tr>
</tbody>
</table>

### Conclusions

- Many driver assumptions do not hold
- Bulk of driver code dedicated to initialization/cleanup
- 44% of drivers have behavior outside class definition
- 43% of drivers perform computation over drivers
- USB/Xen drivers can be offered as services away from kernel
- 8% of driver code can be reduced by better abstractions
- More results in the paper!

### Thank You

- **Contact**
  - Email: kadav@cs.wisc.edu
  - Driver research webpage: http://cs.wisc.edu/sonar

### Extra slides

Taxonomy of Linux drivers developed using static analysis to find out important classes for all our results (details in the paper).
Drivers repeat functionality around kernel wrappers

```c
... delkin_cb_resume(...) {
    struct ide_host *host =
        pci_get_drvdata(dev);
    int rc;
    pci_set_power_state(dev, PCI_D0);
    rc = pci_enable_device(dev);
    if (rc) return rc;
    pci_restore_state(dev);
    pci_set_master(dev);
    if (host->init_chipset)
        host->init_chipset(dev);
    return 0;
}
```

```c
... ide_pci_resume(...) {
    struct ide_host *host =
        pci_get_drvdata(dev);
    int rc;
    pci_set_power_state(dev, PCI_D0);
    rc = pci_enable_device(dev);
    if (rc) return rc;
    pci_restore_state(dev);
    pci_set_master(dev);
    if (host->init_chipset)
        host->init_chipset(dev);
    return 0;
}
```

Drivers covered by our analysis

- All drivers that compile on x86 platform in Linux 2.6.37.6
- Consider driver, bus and virtual drivers
- Skip drivers/staging directory
  - Incomplete/buggy drivers may skew analysis
- Non x86 drivers may have similar kernel interactions
- Windows drivers may have similar device interactions
  - New driver model introduced (WDM), improvement over vxd

Limitations of our analyses

- Hard to be sound/complete over ALL Linux drivers
- Examples of incomplete/unsound behavior
  - Driver maintains private structures to perform tasks and exposes opaque operations to the kernel

Repeated code in family of devices (e.g. initialization)

```c
... asd_aic9405_setup(...) {
    int err = asd_common_setup(...);
    if (err) return err;
    asd_ha_hw_prof.addr_range = 4;
    asd_ha_hw_prof.port_name = 8;
    asd_ha_hw_prof.dev_name = 4;
    asd_ha_hw_prof.sata_name = 8;
    return 0;
}
```

```c
... asd_aic9410_setup(...) {
    int err = asd_common_setup(...);
    if (err) return err;
    asd_ha_hw_prof.addr_range = 8;
    asd_ha_hw_prof.port_name = 8;
    asd_ha_hw_prof.dev_name = 8;
    asd_ha_hw_prof.sata_name = 16;
    return 0;
}
```
How many devices does a driver support?

- Many research projects generate code for specific device/driver
- Example, safety specifications for a specific driver

```
static int __devinit cy_pci_probe(...) {
  if (device_id == PCI_DEVICE_ID_CYCLOM_Y_Lo) { ...
  if (pci_resource_flags(pdev,2)&IOMEMORY_IO) { ...
  if (device_id == PCI_DEVICE_ID_CYCLOM_Y_lo ||
      device_id == PCI_DEVICE_ID_CYCLOM_Y_HI) {...
    } else if (device_id==PCI_DEVICE_ID_CYCLOM_Z_HI) {
    ...
  if (device_id == PCI_DEVICE_ID_CYCLOM_Y_lo ||
     device_id == PCI_DEVICE_ID_CYCLOM_Y_HI) {
    switch (plx_ver) {
      case PLX_9050: ...
      default: /* Old boards, use PLX_9060 */
    }
drivers/char/cyclades.c: Cyclades character driver
```

How many devices does a driver support?

- 28% of drivers support more than one chipset
- 83% of the total devices are supported by these drivers

- Linux drivers support ~14000 devices with 3200 drivers
- Number of chipsets weakly correlated to the size of the driver (not just initialization code)
- Introduces complexity in driver code
- Any system that generates unique drivers/specs per chipset will lead in expansion in code
### Driver device interaction

- Portiommio: Access to memory mapped I/O or x86 ports
- DMA: When pages are mapped
- Bus: When bus actions are invoked
- Varying style of interactions
- Varying frequency of operations

### Class definition used to record state

- Modern research assumes drivers conform to class behavior
- Driver state is recorded based on interfaces defined by class
- State is replayed upon restart after failure to restore state
- Driver behavior is reverse engineered based on interfaces defined by class
- Code is synthesized for another OS based on this behavior

![High-level architecture of RevNIC](image)