SymDrive: Testing Drivers without Devices

Matthew Renzelmann, Asim Kadav, and Michael Swift
University of Wisconsin–Madison
#1: One Driver, Many Devices

```c
static DEFINE_PCI_DEVICE_TABLE(e1000_pci_tbl) = {
    INTEL_E1000_ETHERNET_DEVICE(0x1000),
    INTEL_E1000_ETHERNET_DEVICE(0x1001),
    INTEL_E1000_ETHERNET_DEVICE(0x1004),
    INTEL_E1000_ETHERNET_DEVICE(0x1008),
    INTEL_E1000_ETHERNET_DEVICE(0x1009),
    INTEL_E1000_ETHERNET_DEVICE(0x100C),
    INTEL_E1000_ETHERNET_DEVICE(0x100D),
    INTEL_E1000_ETHERNET_DEVICE(0x100E),
    INTEL_E1000_ETHERNET_DEVICE(0x100F),
    INTEL_E1000_ETHERNET_DEVICE(0x1010),
    INTEL_E1000_ETHERNET_DEVICE(0x1011),
    INTEL_E1000_ETHERNET_DEVICE(0x1012),
    INTEL_E1000_ETHERNET_DEVICE(0x1013),
    INTEL_E1000_ETHERNET_DEVICE(0x1014),
    INTEL_E1000_ETHERNET_DEVICE(0x1015),
    INTEL_E1000_ETHERNET_DEVICE(0x1016),
    INTEL_E1000_ETHERNET_DEVICE(0x1017),
    INTEL_E1000_ETHERNET_DEVICE(0x1018),
    INTEL_E1000_ETHERNET_DEVICE(0x1019),
    INTEL_E1000_ETHERNET_DEVICE(0x101A),
    INTEL_E1000_ETHERNET_DEVICE(0x101D),
    INTEL_E1000_ETHERNET_DEVICE(0x101E),
    INTEL_E1000_ETHERNET_DEVICE(0x1026),
    INTEL_E1000_ETHERNET_DEVICE(0x1027),
    INTEL_E1000_ETHERNET_DEVICE(0x1028),
    INTEL_E1000_ETHERNET_DEVICE(0x1075),
    INTEL_E1000_ETHERNET_DEVICE(0x1076),
    INTEL_E1000_ETHERNET_DEVICE(0x1077),
    INTEL_E1000_ETHERNET_DEVICE(0x1078),
    INTEL_E1000_ETHERNET_DEVICE(0x1079),
    INTEL_E1000_ETHERNET_DEVICE(0x107A),
    INTEL_E1000_ETHERNET_DEVICE(0x107B),
    INTEL_E1000_ETHERNET_DEVICE(0x107C),
    INTEL_E1000_ETHERNET_DEVICE(0x107D),
    INTEL_E1000_ETHERNET_DEVICE(0x108A),
    INTEL_E1000_ETHERNET_DEVICE(0x1099),
    INTEL_E1000_ETHERNET_DEVICE(0x10B5),
    /* required last entry */
    {0,}
};

switch (hw->device_id) {
    case E1000_DEV_ID_82545GM_SERDES:
    case E1000_DEV_ID_82546GB_SERDES:
        hw->media_type = e1000_media_type_internal_serdes;
        break;
    default:
        switch (hw->mac_type) {
            case e1000_82542_rev2_0:
            case e1000_82542_rev2_1:
                hw->media_type = e1000_media_type_fiber;
                break;
            case e1000_ce4100:
                hw->media_type = e1000_media_type_copper;
                break;
            default:
                status = er32(STATUS);
                if (status & E1000_STATUS_TBIMODE) {
                    hw->media_type = e1000_media_type_fiber;
                    hw->tbi_compatibility_en = false;
                } else {
                    hw->media_type = e1000_media_type_copper;
                }
                break;
        }
    break;
}
```

Driver behavior depends on hardware-specific details

Testing with 37 network cards is expensive and inconvenient

The E1000 driver supports 37 distinct pieces of hardware

More Hardware → Expensive Testing
#2: Error Handling

```
ret = lp5523_init_led(&chip->leds[led], &client->dev, i, pdata);
if (ret) {
    goto fail3; // Suppose we take this path
}
```

```
INIT_WORK(&(chip->leds[led].brightness_work), lp5523_led_brightness_work);
```

```
fail3:
for (i = 0; i < chip->num_leds; i++) {
    led_classdev_unregister(&chip->leds[i].cdev);
    ...
}
```

Example from the Linux 3.1.1 lp5523 LED driver

Handling device failures
+ Handling kernel failures
+ Cross-file dependencies
+ Reentrancy
...

→ Difficult Code Review + More Bugs
#3: Kernel Evolution

Patch:
flush_scheduled_work() is on its way out.

The conversions are mostly straight forward and **the only dangers are**, *

Signed-off-by: Tejun Heo <tj@kernel.org>
Cc: "David S. Miller" <davem@davemloft.net>
Cc: Jay Cliburn <jcliburn@gmail.com>
Cc: Michael Chan <mchan@broadcom.com>
Cc: Divy Le Ray <divy@chelsio.com>
Cc: e1000-devel@lists.sourceforge.net
Cc: Vasanthy Kolluri <vkolluri@cisco.com>
Cc: Samuel Ortiz <samuel@sortiz.org>
Cc: Lennert Buytenhek <buytenh@wantstofly.org>
Cc: Andrew Gallatin <gallatin@myri.com>
Cc: Francois Romieu <romieu@fr.zoreil.com>
Cc: Ramkrishna Vepa <ramkrishna.vepa@exar.com>
Cc: Matt Carlson <mcarlson@broadcom.com>
Cc: David Brownell <dbrownell@users.sourceforge.net>
Cc: Shreya

Affected Drivers:
/8139too.c
/atlx/atlx2.c
/bcm63xx_enet.c
/bnx2.c
/cassini.c
/cxgb3/cxgb3_main.c
/e1000e_net.c
/enic.c
/ibm_r8169.c
/irda.c
/ixgb.c
/ixgbevf/ixgbevf_main.c
/mv643xx_eth.c
/myri10ge/myri10ge.c
/niu.c
/pxa168_eth.c
/r8169.c
/s2io.c
/sis190.c
/tg3.c
/usb/sierra_net.c
/usb/usbnet.c
/vmxnet3/vmxnet3_drv.c

---

Patch checking often requires considerable human effort

Hardware costs add up: > 200 devices required here

Patch testing often requires considerable hardware

Kernel Evolution → Expensive Testing
Existing Approaches

- Formal Specification
- Symbolic Execution
- Standard Testing
- Code Review
- Static Analysis
- SymDrive

Developer Effort →

← Finds Fewer Bug Types  Finds Many Bug Types →
SymDrive

- Driver testing and validation tool
  - Runs drivers using symbolic execution: no devices
  - Reduces effort via static analysis/code generation
  - Interposes on driver/kernel interface with test framework
  - Provides broad driver, bus, and platform compatibility

Benefits
- Facilitates writing better quality code
- Allows many people to test and develop drivers
SymDrive Outline

• Introduction
• Goals and Background
• Overview of SymDrive
• Problems and Solutions
  – What problems does SymDrive solve and how?
• Results
  – Does SymDrive find deep bugs? Is it easy to use?
• Conclusion
Goals of SymDrive

- Find deep bugs
- Be easy to use
- Enable broader patch testing

[Kadav2012]
SymDrive must:

- Analyze multiple paths through the driver
  - … to find deep bugs
- Provide a model for device behavior
  - … to be easy to use
- Check driver runtime properties
  - … to enable broader patch testing

Symbolic execution can achieve these goals
Background: Symbolic Execution

```c
void f(void) {
    int x = inb(...); // x is symbolic
    if(x < 10) return -EIO;
    else return 0;
}
```

Path explosion

```
void f(void) {
    int x = inb(...);
    if(x < 10) return -EIO;
    else return 0;
}
void g() {
    if (x) { ... }
    else { ... }
    while (x) { ... }
}

void h() {
    if (x) { ... }
    else { ... }
    while (x) { ... }
}
```
Symbolic Execution Challenges

How should SymDrive…
1. Avoid path explosion and execute useful paths?
2. Check driver runtime properties?
3. Achieve high coverage?
SymDrive Outline

• Introduction
• Goals and Background
• Overview of SymDrive
• Problems and Solutions
  – Path Explosion
  – Driver Testing
• Results
  – Does SymDrive find deep bugs? Is it easy to use?
• Conclusion
Baseline System

Compile Time

- Driver Source
- GCC
- Driver Module

Runtime (VM)

- Test Programs
- OS Kernel
- Driver Module
- Symbolic Bus
- Symbolic Device

S²E Symbolic Execution Engine

How should SymDrive avoid path explosion?

How can the developer check runtime properties using SymDrive?
Challenge #1: Path Explosion

Example #1 (e1000):

```c
ret_val = e1000_read_phy_reg(hw, ..., &phy_data);
if (ret_val) return ret_val;
phy_data |= M88E1000_EPSR_TX_CLK_25;
ret_val = e1000_write_phy_reg(hw, ..., phy_data);
if (ret_val) return ret_val;
ret_val = e1000_read_phy_reg(hw, ..., &phy_data);
if (ret_val) return ret_val;
```

Symbolic hardware may induce path explosion

Example #2 (8139too):

```c
for (i = 1000; i > 0; i--) {
    barrier();
    if (((read (ChipCmd) & CmdReset) == 0)
        break;
    udelay (10);
}
```

Loops may induce path explosion
Solution #1: Inform Runtime

• Inform SymDrive of relevant driver activity
  – Whether functions are executing successfully
  – When loops start/iterate/end

• Use source code hints

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>symdrive_prioritize</td>
<td>Prioritize current path</td>
</tr>
<tr>
<td>symdrive_deprioritize</td>
<td>Deprioritize current path</td>
</tr>
<tr>
<td>symdrive_loop_before</td>
<td>Current path is about to enter a loop</td>
</tr>
<tr>
<td>symdrive_loop_body</td>
<td>Current path is about to iterate the loop</td>
</tr>
<tr>
<td>symdrive_loop_after</td>
<td>Current path exited the loop</td>
</tr>
</tbody>
</table>
Solution #1A: Favor-Success Scheduling

• Prioritize execution of paths that return success
• Generate code when functions begin/end
  – Inform symbolic execution engine of failure

Example (all drivers):

```c
// This code executes when the driver function returns
if (IS_ERR_VALUE(retval)) { // failure
  // …
  symdrive_deprioritize(line);
} else { // success
  // …
}
```

Deprioritize failing paths
Solution #1B: Loop Elision

- Inform SymDrive of loops to improve scheduling
- Prioritize execution of paths that exit loops fast
- Generate instrumentation for loops statically

Example (8139too):

```c
symdrive_loop_before(...);
while (i > 0) {
    ...
    if ((ioread8(ioaddr + 55) & 16U) == 0U) break;
    i--;
    symdrive_loop_body(...);
} symdrive_loop_after(...);
```

Instrument loops; prioritize paths that exit loops quickly
Solution #1C: High-Coverage Mode

- Provides additional symbolic data
- Favors paths that touch unexecuted code
- Makes kernel functions return symbolic values
  - Forks execution: once for success, once for failure

```c
static int __devinit lp5523_init_led(struct lp5523_led *led, 
   struct device *dev, int ch, struct lp5523_platform_data *dat) {

   ... 
   res = led_classdev_register(dev, &led->cdev); 
   if (res < 0) return res; 
   res = sysfs_create_group(&led->cdev.dev->kobj, 
                           &lp5523_led_attribute_group); 
   ... 
   return 0; 
}
```

Fork failure

Fork success

High-coverage mode improves coverage
Implementation: SymGen

Compile Time

Driver Source \rightarrow \text{SymGen} \rightarrow \text{Transformed Driver} \rightarrow \text{GCC} \rightarrow \text{Driver Module}

Runtime (VM)

Static analysis and code generation tool

\text{Instrumentation notifies SymDrive Plugin of path quality}

\text{SymDrive Plugin}

\text{SymDrive Plugin}

\text{S}^2\text{E Symbolic Execution Engine}

\text{Symbolic Device}

\text{Symbolic Bus}

\text{Driver Module}

\text{Test Programs}

\text{Instrumented C source code}
Challenge #2: Testing

• Why?
  – Linux/FreeBSD do not test driver interfaces well
  – Developers need a way to specify correct behavior

• S²E can already report hangs/crashes:

```c
int f(int *p) { ... *p = 0; ... } // If p is NULL, S²E reports error
```

• What about semantic bugs? Example (e1000):

```c
/* tell stack to leave us alone until e1000_open() is called */
netif_carrier_off(netdev);
netif_stop_queue(netdev);
...
err = register_netdev(netdev);
```

Semantic bugs are hard to find
Solution #2A: Checkers

- Verify kernel/driver pre- and postconditions
- Complement symbolic execution

```c
static int netif_carrier_off_check(const char *fn, int prepost, struct net_device **dev) {
    if (prepost == PRECONDITION) {
        tfassert (ndo_state.register_netdev == CALLED_OK);
        ...
    }
    return 0;
}
```

Checkers verify correct behavior
Solution #2B: Interface Interposition

- Determine driver/kernel interface automatically
- Find the appropriate checker
  - Use the test framework
- Example (8139too):

```c
// First, look up the checker for this driver function type:
check_fn_rtl8139_init_one = lookup_checker(&rtl8139_init_one);

// Then, invoke the checker, either as a pre- or postcondition
check_fn_rtl8139_init_one(...);
```

SymGen interposes on the driver/kernel interface automatically
Implementation: Checkers

**Compile Time**
- Driver Source
- SymGen
- Transformed Driver
- GCC
- Driver Module

**Runtime (VM)**
- Support library simplifies checker development
- SymGen interposes on driver / kernel interface automatically
- Test Framework finds correct checker to execute

- Checkers
- Driver Module
- Symbolic Bus
- SymGen
- Support Library
- Test Framework
- S²E Symbolic Device
- SymDrive Plugin
- OS Kernel
- Test Programs
### Example Checkers

<table>
<thead>
<tr>
<th>Checker Category</th>
<th>Assertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocator Mismatch</td>
<td>The dev_id field must match in corresponding calls to request_irq and free_irq.</td>
</tr>
<tr>
<td>Allocator Mismatch</td>
<td>Calls to pci_map_page/pci_unmap_page and pci_map_single/pci_unmap_single must correspond.</td>
</tr>
<tr>
<td>API Misuse</td>
<td>Network drivers must not call netif_stop_queue before calling register_netdev.</td>
</tr>
<tr>
<td>Driver Interface</td>
<td>Ensure that kmalloc and related functions use appropriate allocation flags for the current context.</td>
</tr>
<tr>
<td>Driver Interface</td>
<td>Network drivers: dev-&gt;trans_start field is unchanged.</td>
</tr>
<tr>
<td>Leak</td>
<td>All allocated memory resources are freed appropriately. 12 allocate/free pairs supported via generic API.</td>
</tr>
</tbody>
</table>

**Limitations:** Checkers operate at driver/kernel interface, and do not verify driver/device interaction.
SymDrive Outline

- Introduction
- Goals and Background
- Overview of SymDrive
- Problems and Solutions
  - What problems does SymDrive solve and how?
- Results
  - Does SymDrive find deep bugs? Is it easy to use?
- Conclusion
SymDrive Results

• Goal #1: Find deep bugs
  – Tested variety of drivers
• Goal #2: Be easy to use
  – Measured developer effort
• Goal #3: Enable broader patch testing
  – Measured patch testing effort
Summary of Bugs Found

- 39 bugs found across 26 drivers
- Kernel developers validated many bugs
- Full details at: http://research.cs.wisc.edu/sonar/projects/symdrive

<table>
<thead>
<tr>
<th>Driver</th>
<th>Bug Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>lps523</td>
<td>This was a particularly insidious bug that was hard to diganose despite having a reproducible crash. Without going into the details of the symptom, the bug is a missing file terminator on the $gph_d_pointer array. This array was placed next to $gph_d_isr_data in the compiled binary (and memory image) which is why the driver &quot;mostly&quot; works in any case. Greg Kroah-Hartman verified this bug is legitimate, here, though the patch remains un-merged. The bug is visible here.</td>
</tr>
<tr>
<td>lps523</td>
<td>$lp523.p2f probe calls $lp523_initlinux, $lp523.probes with a lot of work on the corresponding $brightness_work structure. But what if during a call to $lp523_initlinux the call to $sysctl_set_group fails? This call takes place right after the call to $sysctl_set_group. In this case, $lp523_set_x times calls $led_controller_unregister. But $led_controller_unregister completed successfully. So, in this case, $led_controller_unregister calls $led_brightness_set, which invokes the driver function $gph_d_set_brightness. But alas, this function calls $gph_d_set_brightness on the uninitialized work queue and the kernel panics. Bug fixed here. Bug visible here.</td>
</tr>
<tr>
<td>lps523</td>
<td>Fixed here. This patch resulted from a crash in lps523.</td>
</tr>
<tr>
<td>lps523</td>
<td>(Not counted as a bug)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver</th>
<th>Bug Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>akm8975 (Android 3.0 kernel)</td>
<td>The driver executes request_irq before it is necessarily ready to service interrupts. request_irq is called in $akm8975_init_client, but $akm8975_init_client is not set until after that call. The result is that the interrupt handler can crash because it contains this line: $disable_irq_separate($akm8975_client, $irq); There is a similar copy of the driver available here that exhibits this bug, though this is not the precise version of the driver we used. The link is just for convenience.</td>
</tr>
<tr>
<td>akm8975 (Android 3.0 kernel)</td>
<td>Does not delete a device file created during probe (via $device_create_file). This bug is a kind of memory leak. There is a similar copy of the driver available here that exhibits this bug.</td>
</tr>
<tr>
<td>akm8975 (Android 3.0 kernel)</td>
<td>The driver does not appear to stop the work queue during module unload. The driver schedules the workqueue via $schedule_worklist($work) during invocations of the interrupt handler, but there is no corresponding $unregister_work. In addition, the driver could call $schedule_worklist while executing the work queue even though the interrupt has already been freed during module unload.</td>
</tr>
</tbody>
</table>

The bugs in this table are from drivers in Linux v3.1.1. We reported the bugs to verify that they were genuine.
## Bug Categories

<table>
<thead>
<tr>
<th>Bug Type</th>
<th>Bugs</th>
<th>Kernel</th>
<th>Checker</th>
<th>Cross Entry Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Dependence</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>API Misuse</td>
<td>15</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Race</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Allocator Mismatch</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Leak</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Driver Interface</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Bad Pointer</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Totals</strong></td>
<td><strong>39</strong></td>
<td><strong>17</strong></td>
<td><strong>22</strong></td>
<td><strong>22</strong></td>
</tr>
</tbody>
</table>

- Checkers are critical to finding bugs
- Bugs span multiple entry point invocations
- Median of 1 annotation, maximum of 7
SymDrive: Testing Drivers without Devices

Testing with real hardware and gcov:
- 77% driver functions executed
- 75% of code in those functions touched

SymDrive can execute drivers with results comparable to using real hardware

Median functions touched and coverage of touched functions is greater than 80%
SymDrive is fast enough to test drivers

Driver Runtimes

<table>
<thead>
<tr>
<th>Driver</th>
<th>Time Elapsed (hh:mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8139too</td>
<td>2h:36m</td>
</tr>
<tr>
<td>a1026</td>
<td></td>
</tr>
<tr>
<td>apds9802</td>
<td></td>
</tr>
<tr>
<td>econet</td>
<td></td>
</tr>
<tr>
<td>ens1371</td>
<td>8h:23m</td>
</tr>
<tr>
<td>lp5523</td>
<td></td>
</tr>
<tr>
<td>mmc31xx</td>
<td></td>
</tr>
<tr>
<td>phantom</td>
<td></td>
</tr>
<tr>
<td>pluto2</td>
<td></td>
</tr>
<tr>
<td>tle62x0</td>
<td></td>
</tr>
<tr>
<td>es137x</td>
<td>1h:22m</td>
</tr>
<tr>
<td>rl</td>
<td></td>
</tr>
<tr>
<td>Median</td>
<td></td>
</tr>
</tbody>
</table>

System:
- 2.5Ghz Core 2 Quad
- 8GB memory
- Ubuntu 10.10 x64
Conclusions

• SymDrive:
  – Tests drivers without hardware
  – Finds bugs in real drivers that are worth patching
  – Facilitates driver patch testing and refactoring
  – Supports two OSs and many driver classes

• For more information:
  http://research.cs.wisc.edu/sonar/projects/symdrive
  <mjr, kadav, swift>@cs.wisc.edu