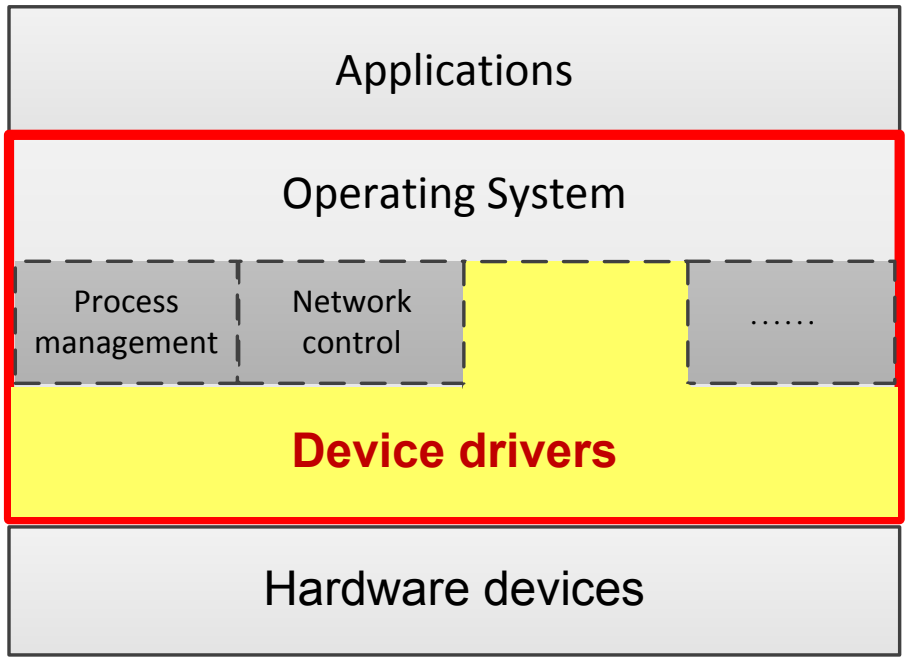
Testing Error Handling Code in Device Drivers Using Characteristic Fault Injection

Jia-Ju Bai, Yu-Ping Wang, Jie Yin, Shi-Min Hu
Department of Computer Science and Technology
Tsinghua University
Beijing, China

清華大學
Tsinghua University

DRIVER INTRODUCTION

- Role
  - Manage hardware devices
  - Support high-level programs
  - Run in kernel mode



DRIVER ERROR HANDLING

- Occasional errors
  - Kernel exceptions (-ENOMEM, -EFAULT, ……)
  - Hardware malfunctions (-EIO, -EBUSY, ……)
  - ……
- Challenges for error handling
  - Complex program logic and context
  - Many different kinds of errors
  - Infrequent to trigger
  - ……

Error handling code in drivers is necessary but hard to correctly implement

MOTIVATION

- Error handling code is incorrect in some drivers

Memory is allocated

Path: linux-3.1.1/drivers/net/bnx2.c

```
7869. static int __devinit bnx2_init_board(....)
7870. {
          ......
7885.     bp->temp_stats_blk = kzalloc(...);
          ......
7906.     rc = pci_request_regions(pdev, ...);
          ......
7937.     bp->regview = ioremap_nocache(...);
7938.     if (!bp->regview) {
7939.         dev_err("Cannot map register space, aborting\n");
7940.         rx = -ENOMEM;
7940.         goto err_out_release;
7941.     }
          ......
8247. err_out_release:
8248.     pci_release_regions(pdev);
8249. err_out_disable:
8250.     pci_disable_device(pdev);
8251.     pci_set_drvdata(pdev, NULL);
8252. err_out:
8253.     return rc;
8254. }
```

Error handling is triggered

Memory is NOT freed!

MOTIVATION

- Patch study
  - Source: Patchwork (*http://patchwork.ozlabs.org/*)
  - July 2015

Driver Class	Accepted Patches	Error Handling
I2C	29	13(44.83%)
PCI	38	13(34.21%)
PowePC	42	11(26.19%)
RTC	24	8(33.33%)
Network	598	253(42.31%)
Total	731	298(40.77%)

- Findings
  - 40% of accepted patches are related to error handling code
  - Many error handling patches are used to fix common bugs

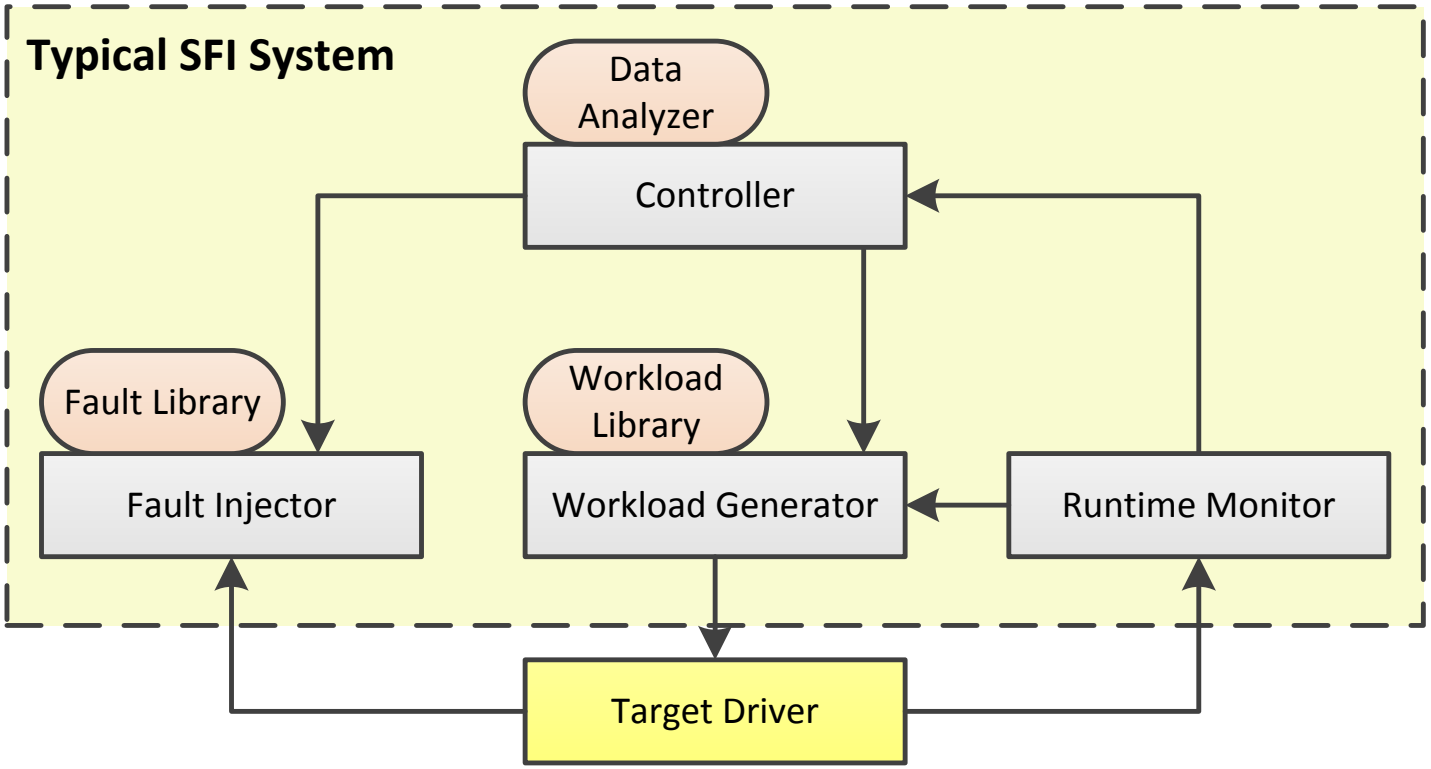
Error handling code in current drivers is not reliable enough

GOAL

- Testing error handling code in device drivers
  - Bug-detection capability
  - Error-handling-code coverage
  - Automation and efficiency
  - Scalability and generality

BASIC TECHNIQUE

- Software fault injection (SFI)
  - Good coverage for error handling code
  - Exact runtime information for bug detection
  - Support most drivers



PREVIOUS SFI APPROACHES

- Some famous approaches
  - Linux Fault Injection Capabilities Infrastructure
  - ADFI (ISSTA ’15), KEDR (ICST ’11), LFI (DSN ’09), ……
- Limitations
  - Low fault representativeness
  - Numerous redundant test cases
  - Several kinds of faults
  - Much manual effort

Our solution is to introduce driver characteristics into SFI

CHARACTERISTIC 1

- Function return value trigger
  - The error handling code is often triggered by a bad function return value
- Driver study
  - 75% of "*goto*" statements are in *if* branches of bad function return values

Driver Class	Number	"*Goto*" Statement	Return Value
Wireless	116	5109	3757(73.54%)
Ethernet	219	6749	5192(76.93%)
Block	56	1322	1005(76.02%)
Bluetooth	21	121	89(73.56%)
Clock	117	260	213(81.92%)
PCI	51	467	351(75.16%)
USB	268	4148	2971(71.62%)
Total	848	18176	13578(74.70%)

CHARACTERISTIC 2

- Few branches
  - There are few *if* branches in error handling code
- Driver study
  - 78% of error handling code is out of the *if* branches
  - Reason: fail-stop model

Driver Class	Number	Error handling	Without Branch
Wireless	116	3903	3111(79.71%)
Ethernet	219	2587	1941(75.03%)
Block	56	149	127(85.23%)
Bluetooth	21	330	239(72.42%)
Clock	117	467	422(90.36%)
PCI	51	470	371(78.94%)
USB	268	701	493(70.32%)
Total	848	8607	6704(77.89%)

CHARACTERISTIC 3

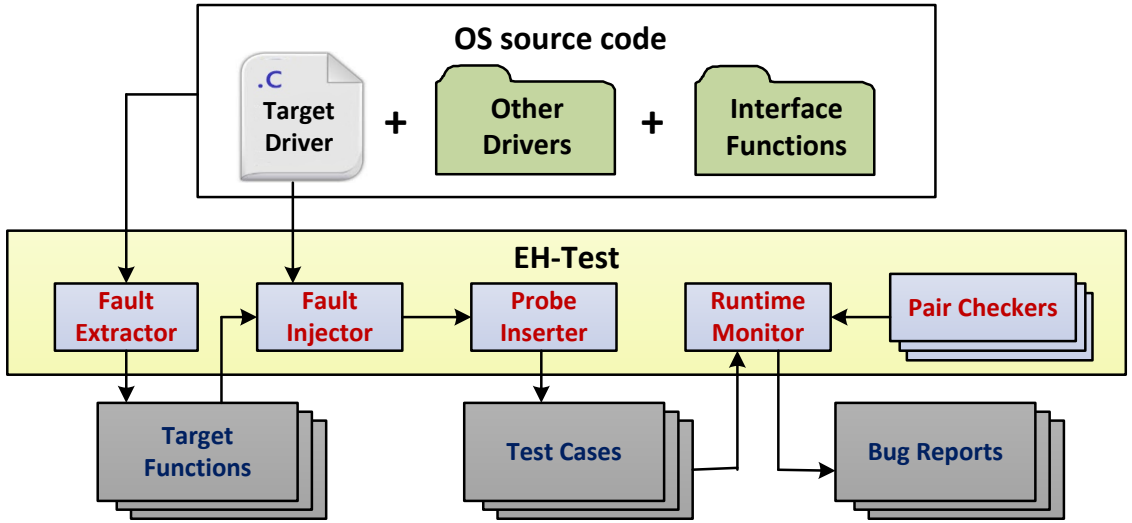
- Check decision
  - To check whether an occasional error occurs, an *if* check is often used in the source code
  - The checked data can be function return values (C1) or common variables

CHARACTERISTIC USAGE

- Function return value trigger (C1)
  - Injecting faults into function return values can cover most error handling code
- Few branches (C2)
  - Injecting single fault in each test case can cover most error handling code
- Check decision (C3)
  - The function whose return value is checked in the code should be fault-injected

EH-TEST

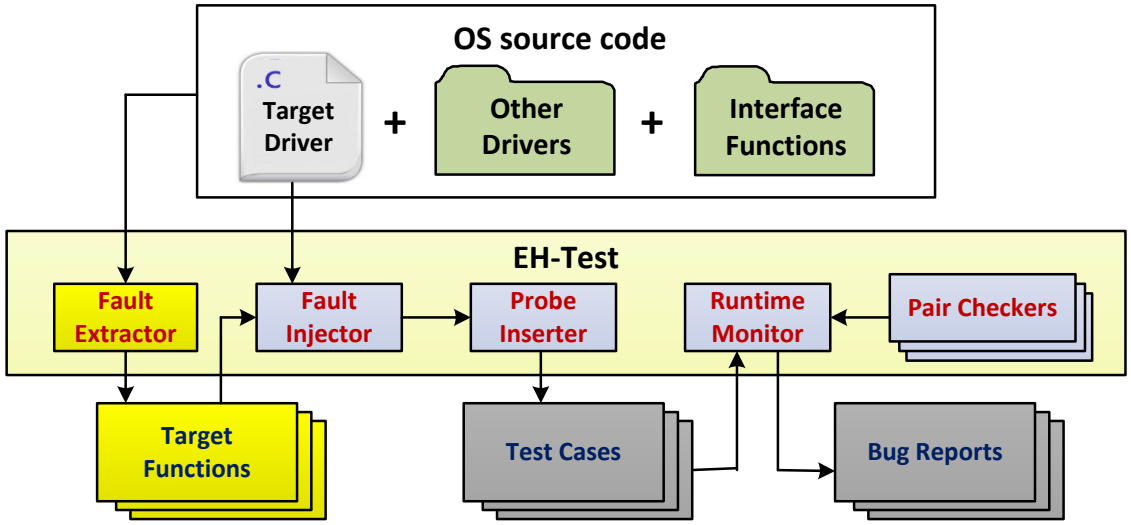
- Architecture
  - Fault extractor
  - Fault injector
  - Probe inserter
  - Runtime monitor
  - Pair checkers



- Two phases
  - Test case generation
  - Runtime testing

PHASE 1: TEST CASE GENERATION

- Task 1: Extracting target functions
  - Input: OS + driver source code
  - Output: target functions
  - Method: pattern-based extraction strategy



PATTERN-BASED EXTRACTION

- Based on C1 and C3
- Three code patterns
- Automated and accurate extraction

Collect traces:

Simple extraction: (candidate functions)

Pattern 1:

Pattern 2:

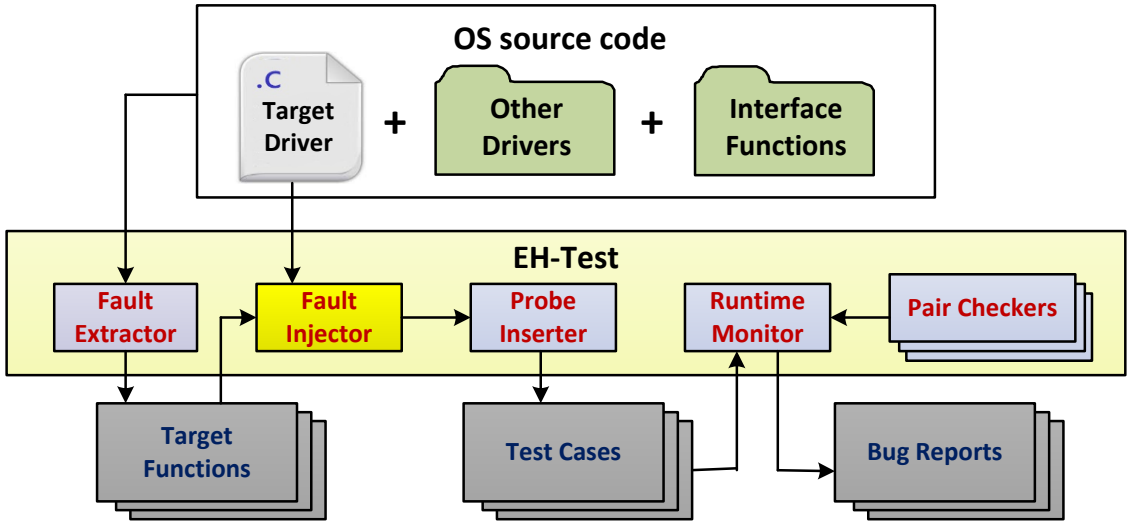
Pattern 3:

Procedure: Pattern-based extraction strategy

```
1:  func_set := ø; cand_set := ø; fault_set := ø;
2:  func_set := called functions in normal execution traces;
3:  foreach func in func_set do
4:      if GetRetType(func) == integer or pointer then
5:          AddSet(cand_set, func);
6:      end if
7   end foreach
8:  foreach func in cand_set do
9:      if func's RetVal is checked by "if" in the driver then
10:         AddSet(fault_set, func);
11:     else if func's RetVal is checked in other drivers then
12:         AddSet(fault_set, func);
13:     else if func's RetVal is specified to be checked then
14:         AddSet(fault_set, func);
15:     end if
16: end foreach
```

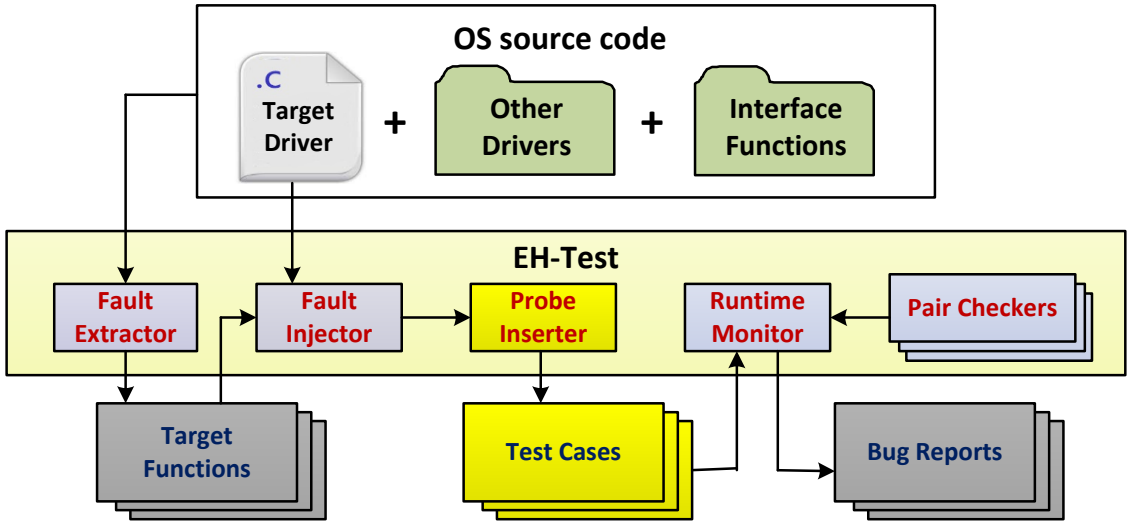
PHASE 1: TEST CASE GENERATION

- Task 2: Injecting faults into target functions
  - Input: driver code + target functions
  - Output: processed driver LLVM bytecode
  - Method: single fault injection, code instrumentation



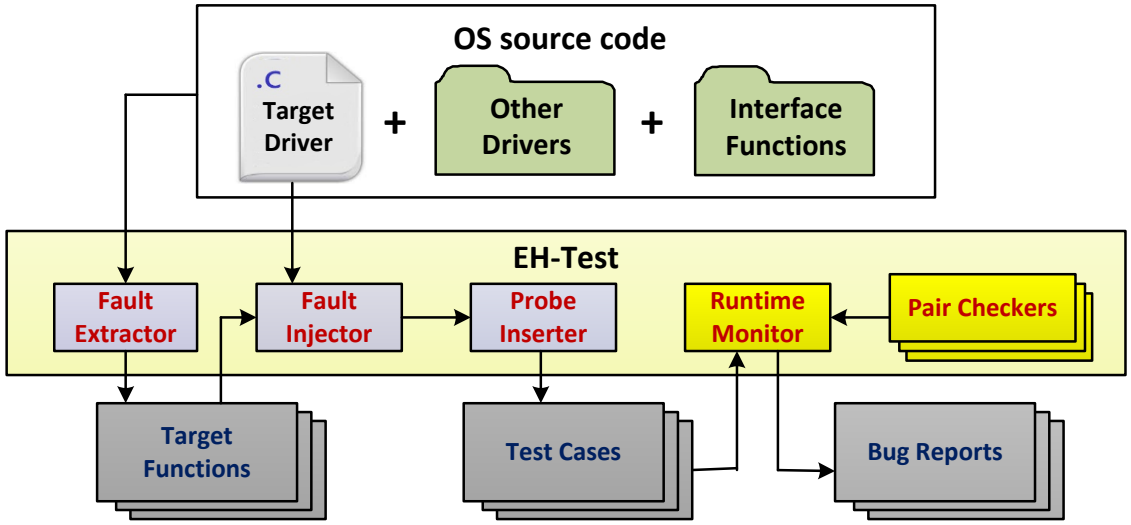
PHASE 1: TEST CASE GENERATION

- Task 3: Inserting probes for runtime monitoring
  - Input: processed driver LLVM bytecode
  - Output: driver test cases (loadable drivers)
  - Method: code instrumentation



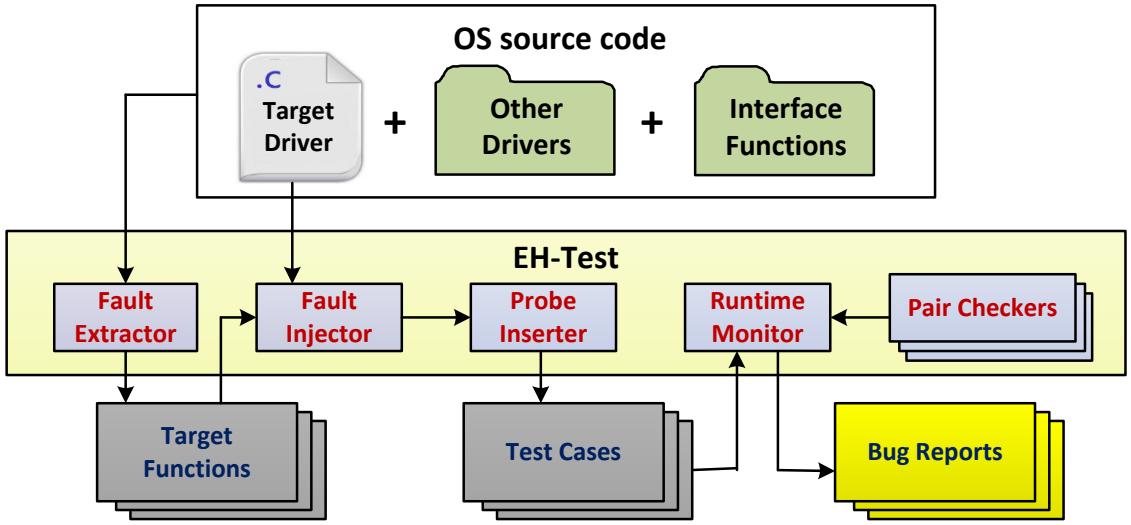
PHASE 2: RUNTIME TESTING

- Runtime monitoring
  - Record runtime information
  - Maintain a resource-usage list
  - Measuring code coverage



PHASE 2: RUNTIME TESTING

- Bug reporting
  - Driver crashes
  - Driver hangs
  - Resource-release omissions



EVALUATION

- 15 common Linux drivers (3.1.1 and 3.17.2)
  - 4 wireless drivers
  - 3 USB drivers
  - 8 Ethernet drivers

Class	Driver	Hardware	Lines
Wireless	rtl8180	Realtek RTL8180L Wireless Controller	4.6K
	b43	Broadcom BCM4322 Wireless Controller	57.5K
	iwl4965	Intel 4965AGN Wireless Controller	29.1K
	rt2800	Ralink RT3060 Wireless Controller	22.5K
USB	usb_storage	Kingston 4GB USB disk	7.6K
	uhci_hcd	Intel USB UHCI Controller	7.2K
	ehci_hcd	Intel USB2 EHCI Controller	11.2K
Ethernet	e100	Intel 82559 Ethernet Controller	3.2K
	e1000e	Intel 82572EI Ethernet Controller	28.3K
	igb	Intel 82575EB Ethernet Controller	24.9K
	r8169	Realtek RTL8169 Ethernet Controller	7.4K
	8139too	Realtek RTL8139D Ethernet Controller	2.7K
	3c59x	3Com 3c905B Ethernet Controller	3.4K
	sky2	Marvell 88E8056 Ethernet Controller	7.7K
	ipg	ICPlus IP1000 Ethernet Controller	3.0K

EVALUATION

- Target function extraction
  - 76% of candidate functions are filtered out
  - 10% false positive rate
  - 86% of target functions are called in initialization

Driver	Candidate	Target	Real
rtl8180	39	18	17 (14)
b43	260	55	55 (46)
iwl4965	497	79	74 (64)
rt2800	185	65	57 (48)
usb_storage	60	20	15 (15)
uhci_hcd	120	24	19 (10)
ehci_hcd	160	23	21 (14)
e100	80	33	27 (26)
e1000e	175	62	56 (41)
igb	247	59	51 (51)
r8169	77	15	15 (14)
8139too	64	9	8 (7)
3c59x	59	15	14 (14)
sky2	86	30	25 (25)
ipg	74	16	16 (15)
Total	2183	523	470 (404)

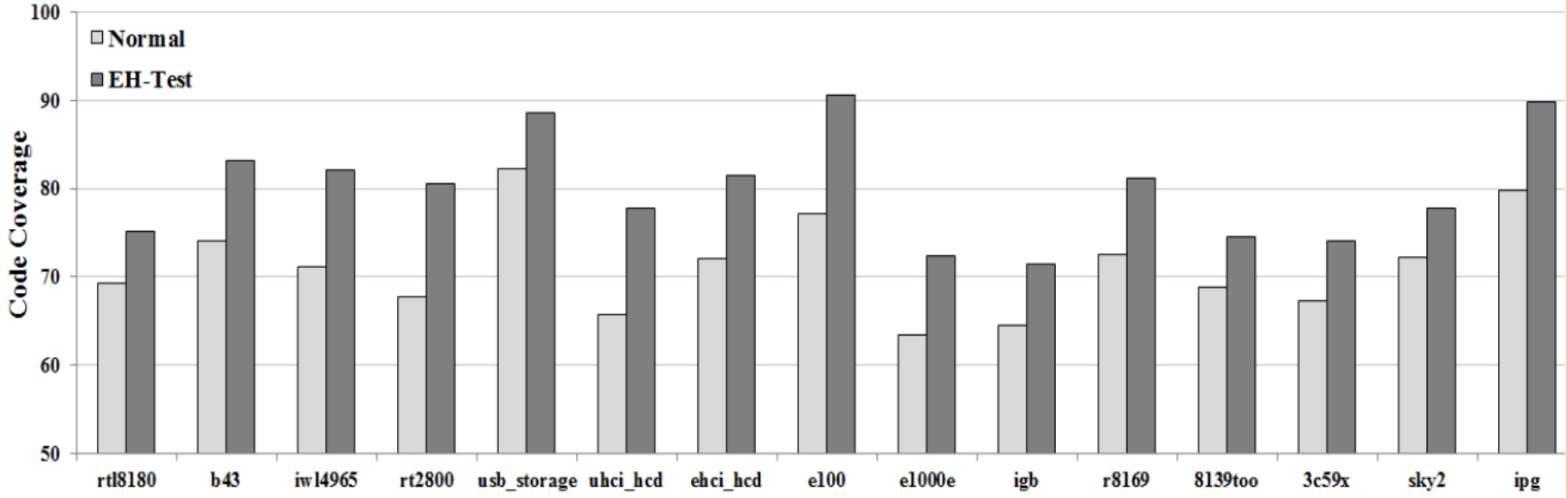
EVALUATION

- Bug detection
  - 32 real bugs in 3.1.1, 50 real bugs in 3.17.2
  - 9 bugs in 3.1.1 have been fixed in 3.17.2
  - 17 patches are sent, and 15 of them are applied
  - Many resource-release omissions

Driver	Linux 3.1.1					Linux 3.17.2				
	Test case	*Time usage*	*Crash / Hang*	*Resource*	*Bugs*	*Test case*	*Time usage*	*Crash / Hang*	*Resource*	*Bugs*
rtl8180	16	03:14	0 / 0	1 (0)	1	18	04:21	0 / 0	3 (2)	3
b43	62	23:57	0 / 0	1 (1)	1	55	26:34	0 / 0	1 (1)	1
iwl4965	100	36:42	5 / 0	7 (7)	12	79	25:18	5 / 0	8 (8)	13
rt2800	62	19:21	1 / 0	1 (0)	2	65	21:37	0 / 0	1 (0)	1
usb_storage	25	03:35	0 / 0	0 (0)	0	20	03:07	0 / 0	0 (0)	0
uhci_hcd	22	03:47	0 / 0	0 (0)	0	24	03:20	0 / 0	0 (0)	0
ehci_hcd	24	03:50	0 / 0	1 (0)	1	23	03:58	0 / 0	10 (9)	10
e100	33	03:02	1 / 0	0 (0)	1	33	02:28	1 / 0	1 (1)	2
e1000e	66	11:01	0 / 0	0 (0)	0	62	10:30	3 / 0	3 (0)	6
igb	62	10:09	0 / 0	0 (0)	0	59	12:56	0 / 1	6 (6)	7
r8169	15	01:24	0 / 0	0 (0)	0	15	01:43	0 / 0	0 (0)	0
8139too	9	00:45	0 / 0	1 (0)	1	9	00:46	0 / 0	1 (0)	1
3c59x	18	01:26	0 / 0	2 (2)	2	15	01:24	0 / 0	2 (2)	2
sky2	26	01:43	3 / 0	8 (0)	11	30	02:14	4 / 0	0 (0)	4
ipg	17	01:16	0 / 0	0 (0)	0	16	01:28	0 / 0	0 (0)	0
Total	557	125:12	10 / 0	22 (10)	32	523	121:42	13 / 1	36 (29)	50

EVALUATION

- Code coverage
  - Improve 8.8% in driver initialization
  - Not all error handling code can be covered



ADFI VS EH-TEST

- ADFI [ISSTA ’15]
  - SFI testing for drivers
  - Injecting faults into target function return values
  - Detect crashes, hangs and memory leaks
- Differences
  - Target functions are manually selected
  - Injecting multiple faults into each test case
- Bug detection
  - Find the same number of bugs in *e100* and *r8169*
  - 10 bugs in *ehci_hcd* found by EH-Test are omitted

LIMITATIONS

- Some error handling code is uncovered
  - Single fault injection
  - Only injecting faults into function return values
- Only default configuration is covered

CONCLUSION

- Driver code study and 3 useful characteristics
- Automated and accurate method: pattern-based extraction strategy
- Efficient SFI approach: EH-Test
- 50 real bugs in 15 Linux drivers
- Future work: cover more error handling code and configurations

Thanks!
Q & A