DSAC: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules

Jia-Ju Bai[1], Yu-Ping Wang[1], Julia Lawall[2], Shi-Min Hu[1]

[1]Tsinghua University, [2]Sorbonne Université/Inria/LIP6

Tsinghua University

inventors for the digital world

Background

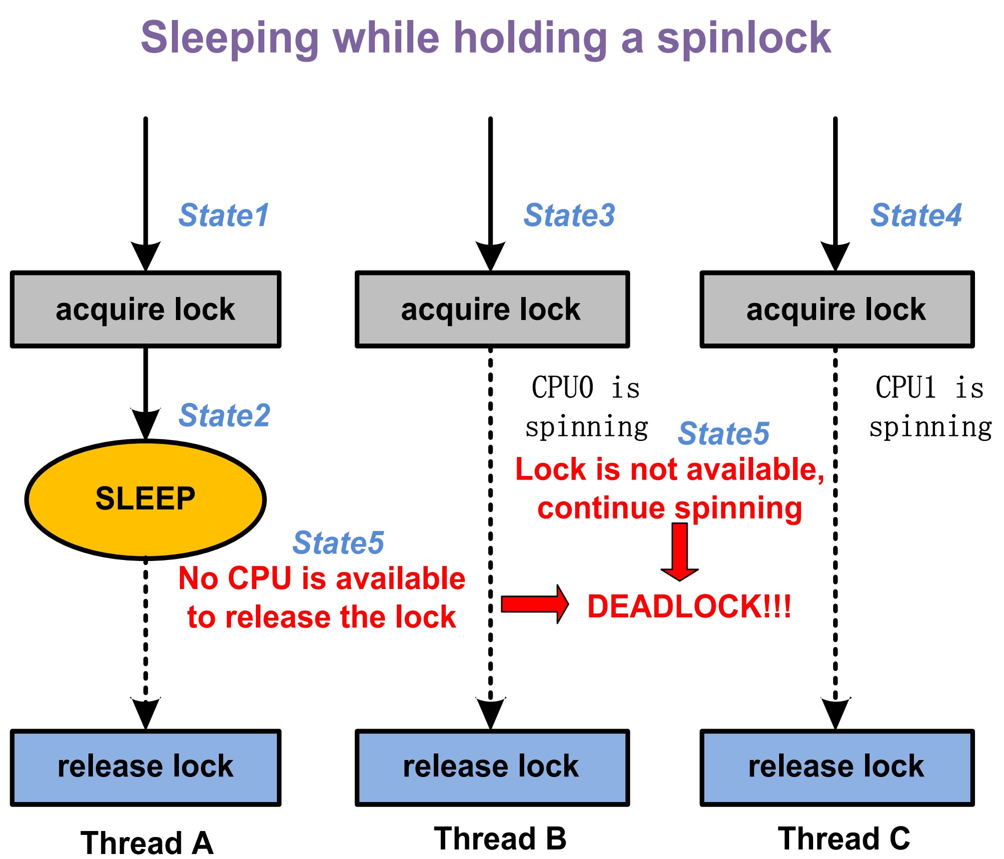
- Atomic context
  - An OS kernel state
  - A CPU core is occupied to execute the code without interruption
  - Protect resources from concurrent access
- Common examples of atomic context
  - Code is executed while holding a spinlock
  - Code is executed in an interrupt handler

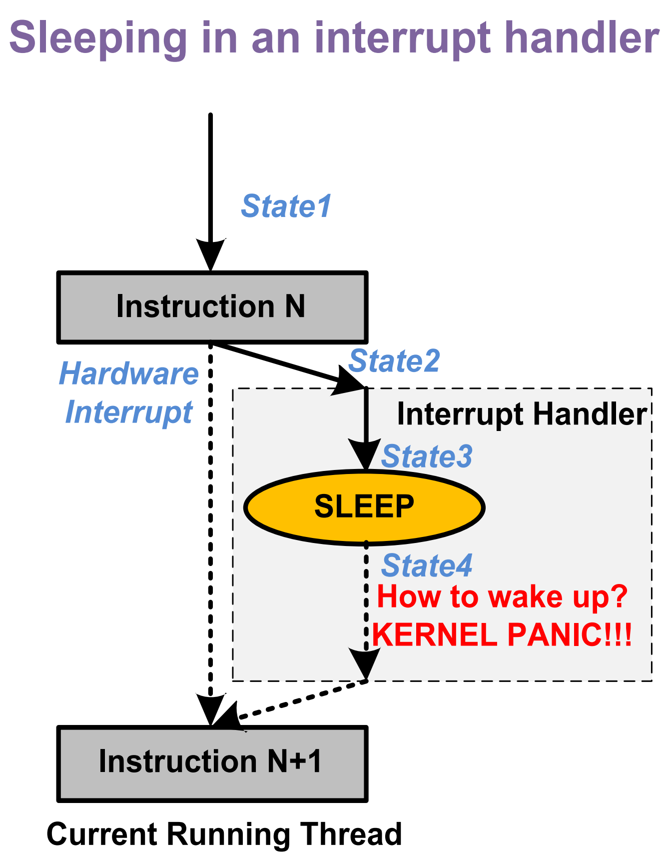
Motivation

- SAC (**S**leep in **A**tomic **C**ontext) bug
  - Sleeping in atomic context is not allowed
  - SAC bug can cause a system hang or crash at runtime
  - A kind of concurrency bugs

Motivation

- Why can a SAC bug cause a hang or crash?





Motivation

- Example fixed SAC bug

```
FILE: linux-2.6.38/drivers/usb/gadget/mv_udc_core.c
382. static struct mv_dtd *build_dtd(...) {
         ......
399.     dtd = dma_pool_alloc(udc->dta_pool, GFP_KERNEL, dma);
         ......
438. }
-----------------------------------------------------------
441. static int req_to_dtd(...) {
         ......
452.     dtd = build_dtd(...);
         ......
473. }
-----------------------------------------------------------
724. static int mv_ep_queue(...) {
         ......
774.     spin_lock_irqsave(...);
775.     req_to_dtd(...);
         ......
799. }
```

Can sleep!

Acquire a spinlock!

Motivation

- Why do SAC bugs still occur in kernel modules?
  - Determining whether an operation can sleep requires OS-specific knowledge
  - SAC bugs are only occasionally triggered at runtime
  - Multiple levels of function calls should be considered

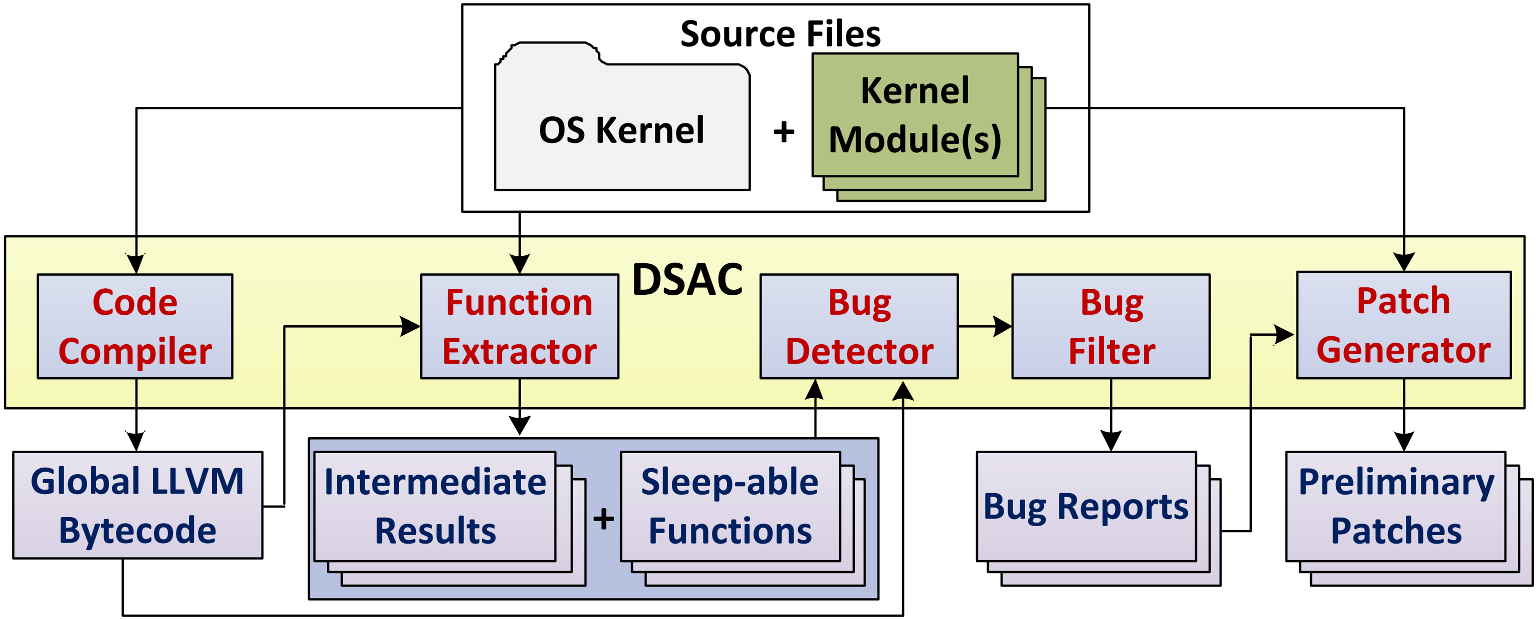
=> Most SAC bugs are manually found by code review

Goal

- Detect SAC bugs in kernel modules
  - Automation
  - Accuracy
  - Efficiency
  - Bug fixing

Approach

- DSAC
  - LLVM-based static analysis tool
  - Detect SAC bugs and recommend bug-fixing patches



Challenges

- Code coverage, accuracy and time
  - Static analysis? Runtime analysis?
- Extract sleep-able functions
  - Require OS-specific knowledge?
- Filter out repeated and false bugs
  - How to check?
- Bug fixing recommendation
  - Needs manual work?

Techniques

- Code coverage, accuracy and time
  - Hybrid flow (flow-sensitive and -insensitive) analysis
- Extract sleep-able functions
  - Heuristics-based extraction method
- Filter out repeated and false bugs
  - Path-check filtering method
- Bug fixing recommendation
  - Pattern-based method

Hybrid flow analysis

- Inter-procedural
- Context-sensitive
  - Lock stack
  - Interrupt flag
  - Executed code path (basic blocks)
- Hybrid of flow-sensitive and -insensitive
  - Flow-sensitive: contain spinlock related function calls in an interrupt handler
  - Flow-insensitive: others

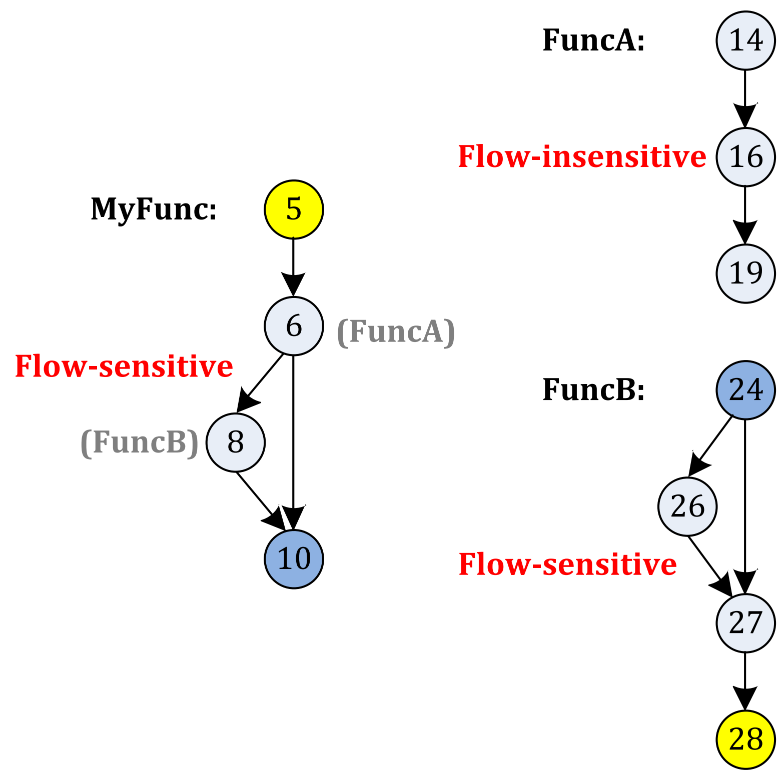
Hybrid flow analysis

- Analysis start
  - Each call to spinlock acquiring function
  - Entry of each interrupt handler function
- Analysis end
  - Lock stack is empty and interrupt flag is FALSE
- Unroll loops and recursive calls once

Hybrid flow analysis

- Example

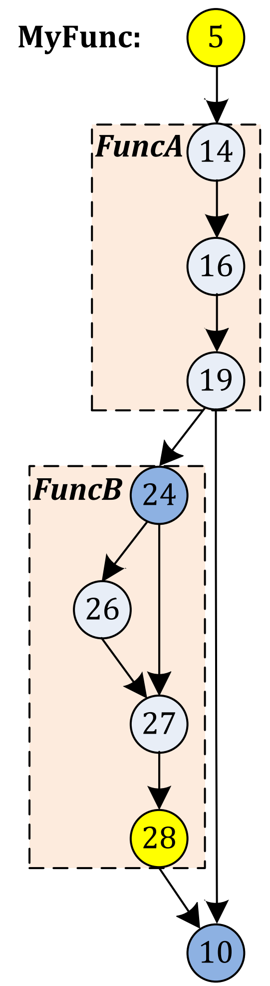
```
1: int FuncA(device *dev);
2: void FuncB(device *dev);
3:
4: void MyFunc(device *dev) {
5:     spin_lock(dev->lock);
6:     if (FuncA(dev))
7:         goto exit;
8:     FuncB(dev);
9: exit:
10:    spin_unlock(dev->lock);
11: }
12:
13: int FuncA(device *dev) {
14:    int v = reg_read(dev->reg, 0x01);
15:    if (!v) {
16:        printk("REG data error!\n");
17:        return -EIO;
18:    }
19:    msleep(1);
20:    return 0;
21: }
22:
23: void FuncB(device *dev) {
24:    spin_unlock(dev->lock);
25:    if (!dev->reply_msg)
26:        printk("No reply, wait!\n");
27:    msleep(10);
28:    spin_lock(dev->lock);
29: }
```



Hybrid flow analysis

- Example

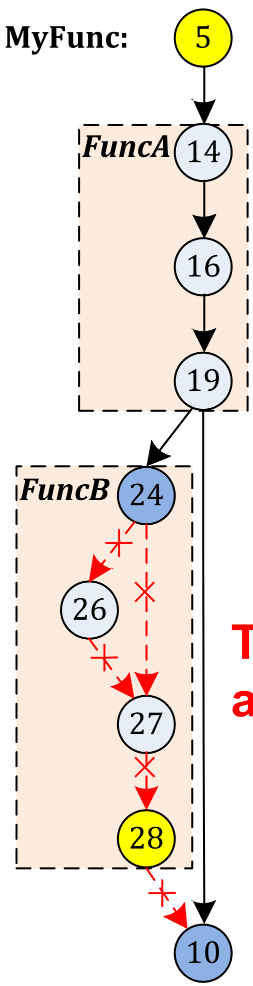
```
int FuncA(device *dev);
void FuncB(device *dev);

void MyFunc(device *dev) {
    spin_lock(dev->lock);
    if (FuncA(dev))
        goto exit;
    FuncB(dev);
exit:
    spin_unlock(dev->lock);
}

int FuncA(device *dev) {
    int v = reg_read(dev->reg, 0x01);
    if (!v) {
        printk("REG data error!\n");
        return -EIO;
    }
    msleep(1);
    return 0;
}

void FuncB(device *dev) {
    spin_unlock(dev->lock);
    if (!dev->reply_msg)
        printk("No reply, wait!\n");
    msleep(10);
    spin_lock(dev->lock);
}
```



Hybrid flow analysis

- Example

```
1: int FuncA(device *dev);
2: void FuncB(device *dev);
3:
4: void MyFunc(device *dev) {
5:     spin_lock(dev->lock);
6:     if (FuncA(dev))
7:         goto exit;
8:     FuncB(dev);
9: exit:
10:    spin_unlock(dev->lock);
11: }
12:
13: int FuncA(device *dev) {
14:    int v = reg_read(dev->reg, 0x01);
15:    if (!v) {
16:        printk("REG data error!\n");
17:        return -EIO;
18:    }
19:    msleep(1);
20:    return 0;
21: }
22:
23: void FuncB(device *dev) {
24:    spin_unlock(dev->lock);
25:    if (!dev->reply_msg)
26:        printk("No reply, wait!\n");
27:    msleep(10);
28:    spin_lock(dev->lock);
29: }
```



Two useless paths are not analyzed

Heuristics-based extraction

- Identify whether a collected function can sleep
  - Involves known sleep-able operation

    like *msleep*() call and *GFP_KERNEL* flag
  - Contains comments suggesting sleep

    like “may block” and “can sleep”
  - Call an already identified sleep-able function

Path-check filtering

- Why may repeated and false bugs occur?
  - Some code paths may be repeatedly analyzed
  - Neglect variable information and path conditions
- Check collected code path in hybrid flow analysis

Path-check filtering

- Filter out repeated bugs
  - Entry and terminal basic blocks
  - Sleep-able function name
- Filter out false bugs
  - Check a function parameter whose name contains the keyword indicating it can sleep (*“can_sleep”*)
  - Check the return value of a function like *in_interrupt* that is used to test atomic context

FILE: linux-4.11.1/drivers/scsi/ufs/ufshcd.c

```
504. static int ufshcd_wait_for_register(..., bool can_sleep) {
         ......
515.     if (can_sleep)
516.         usleep_range(...);
517.     else
518.         udelay(...);
         ......
527. }
```

Pattern-based patch generation

- Four common patterns of fixing SAC bugs
  - P1: sleep-able function ⇒ non-sleep function

    msleep(…) ⇒ mdelay(…)
  - P2: sleep-able flag ⇒ non-sleep flag

    GFP_KERNEL ⇒ GFP_ATOMIC
  - P3: move sleep-able function out of spinlock protection
  - P4: replace spinlock with sleep-able lock
- Support
  - DSAC supports P1 and P2
  - Supporting P3 and P4 is future work

Evaluation

- Linux drivers
  - Run on a common PC
  - Linux-3.17.2 (released in October 2014)
  - Linux-4.11.1 (released in May 2017)
  - Make *allyesconfig* of x86
  - Manually check the detected bugs

Evaluation

- Linux drivers

Description		3.17.2	4.11.1
Bug detection	Filtered bugs	479,912	630,354
	Final bugs	215	340
	Real bugs	200	320
Patch generation		-	43
Time usage		67m53s	84m10s

Evaluation

- Linux drivers
  - Linux-3.17.2:

    Find 215 bugs, 200 are real

    => 50 have been fixed in Linux-4.11.1
  - Linux-4.11.1:

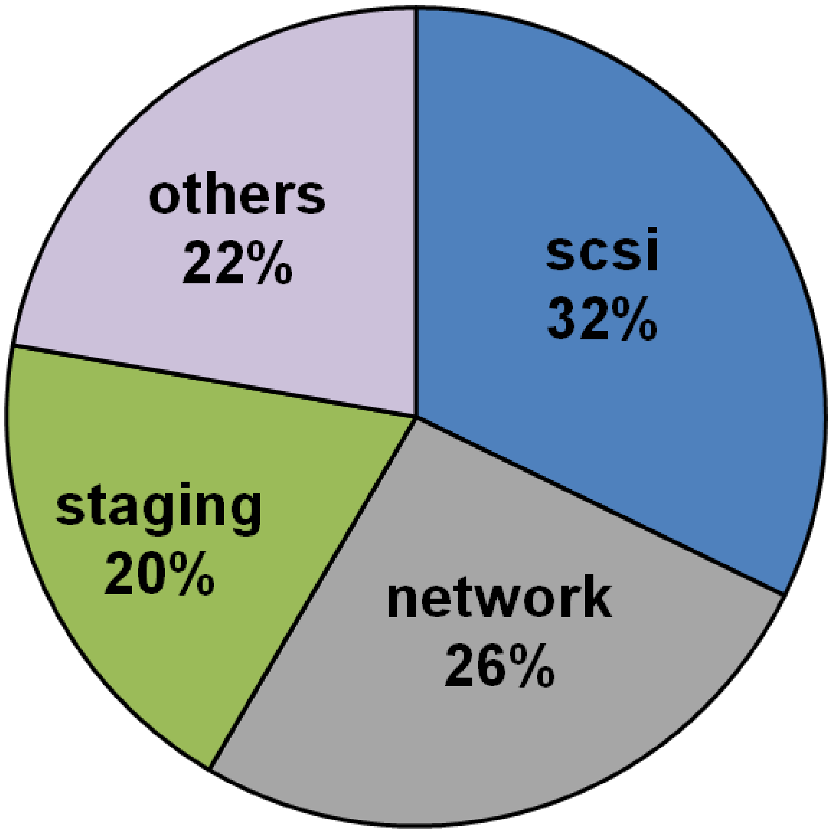
    Find 340 bugs, 320 are real

    => 209 have been confirmed
  - Recommend 43 patches to fix 82 bugs

    => 30 patches have been applied
  - False positives: path condition is not checked

Evaluation

- Linux drivers
  - SCSI and network drivers have 58% of detected bugs



Evaluation

- Other kernel modules
  - Linux network and filesystem modules
  - FreeBSD and NetBSD kernels

Description		Linux net & fs	FreeBSD-11.0	NetBSD-7.1
Bug detection	Filtered bugs	682,081	508	2,414
	Final bugs	42	39	7
	Real bugs	39	35	7
Patch generation		5	10	3
Time usage		32m45s	49m12s	43m38s

Evaluation

- Other kernel modules
  - Find 88 bugs, and 81 are real

    => 63 have been confirmed
  - Recommend 18 patches to fix 59 bugs

    => 13 have been applied

Comparison

- Coccinelle BlockLock checker [1, 2]
  - Find 31 bugs for Linux-2.6.33 drivers that are in x86 config
  - 25 are real, and 6 are false
  - Do not rely on configuration

- DSAC
  - Find 228 bugs for Linux-2.6.33 drivers of x86 config
  - 208 are real, and 20 are false
  - 53 bugs are equivalent to 23 bugs found by BlockLock
  - Rely on configuration

1. N. Palix, etc. Faults in Linux: ten years later. In ASPLOS 2011.
2. N. Palix, etc. Faults in Linux 2.6. In TOCS, 2014.

Limitations

- Function pointer
  - Field-based analysis?
- Repeated analysis
  - Summary-based analysis?
- Path condition
  - Symbolic-execution-like analysis?

Conclusion

- DSAC approach: effective and automated
  - Hybrid flow analysis
  - Heuristics-based extraction method
  - Path-check filtering method
  - Pattern-based method
- Finds 401 new real bugs in Linux, FreeBSD and NetBSD
- Overall false positive rate is about 6%