操作系统的静态分析与缺陷检测

白家驹
清华大学操作系统实验室
https://baijiaju.github.io/

Tsinghua University

Outline

1. Introduction to operating system and static analysis
2. Work1: detecting sleep-in-atomic-context bugs
3. Work2: detecting concurrency use-after-free bugs
4. Work3: detecting unsafe DMA accesses
5. Our ongoing works and discussion

Outline

1. **Introduction to operating system and static analysis**
2. Work1: detecting sleep-in-atomic-context bugs
3. Work2: detecting concurrency use-after-free bugs
4. Work3: detecting unsafe DMA accesses
5. Our ongoing works and discussion

Operating system (OS)

- Operating system is the fundamental computer software
  - Provide services for user-level applications
  - Manage computer resources (such as memory and CPUs)
  - Control hardware devices (such as USB and network devices)

Operating system (OS)

- Key parts in an operating system
  - Filesystems: ext2, ext4, ntfs, btrfs, …
  - Network stacks: ipv4, ipv6, tcp, udp, …
  - Security modules: tomoyo, yama, smack, bpf, …
  - Device drivers: USB, Ethernet, wireless, disk, …
  - ……

Operating system (OS)

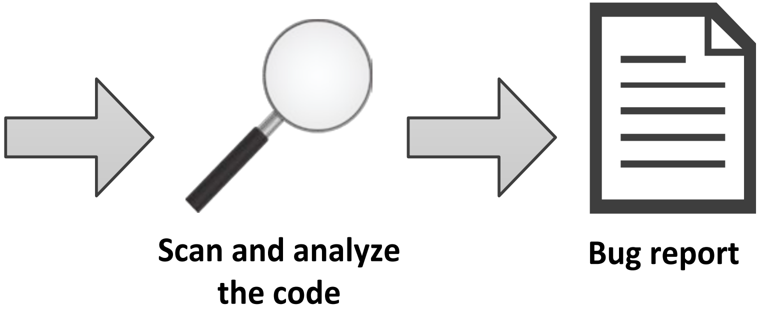
- Operating systems are not reliable and safe as expected
  - In 2016, 523 new vulnerabilities are reported in the Android OS
  - In 2017, >2000 new real bugs are reported in the Linux kernel

	Product Name	Vendor Name	Product Type	Number of Vulnerabilities
1	Android	Google	OS	523
2	Debian Linux	Debian	OS	319
3	Ubuntu Linux	Canonical	OS	278
4	Flash Player	Adobe	Application	266
5	Leap	Novell	OS	259
6	Opensuse	Novell	OS	228
7	Acrobat Reader Dc	Adobe	Application	227
8	Acrobat Dc	Adobe	Application	227
9	Acrobat	Adobe	Application	224
10	Linux Kernel	Linux	OS	216

Static analysis

- Static analysis is a common method of program analysis
  - Analyze program code without actual running
  - High code coverage
  - Easy to use and deploy

```
int func(int arg) {
    int a, b, c;
    a = 0; b = 5;
    if (arg > 0)
        b = a;
    if (arg > 5)
        c = arg / b;  // DIV ZERO
    else
        c = arg / 4;
    return c;
}
```



Static analysis

- Key techniques in static analysis
  - Inter-/intra-procedural analysis
  - Flow-sensitive/-insensitive analysis
  - Context-sensitive/-insensitive analysis
  - Field-sensitive/-based/-insensitive analysis
  - Array-sensitive/-insensitive analysis
  - Alias analysis and function-pointer analysis
  - ……

Static analysis of operating system

- Challenges
  - Inter-procedural analysis for large-scale code
  - Function-pointer analysis
  - Identification of concurrent function pairs
  - Concurrency-problem detection
  - Hardware-access checking
  - Code-path validation to reduce false positives
  - ……

Our approaches

- ***DSAC: detecting sleep-in-atomic-context bugs***
  - Inter-procedural analysis for large-scale code
  - Function-pointer analysis
- ***DCUAF: detecting concurrency use-after-free bugs***
  - Identification of concurrent function pairs
  - Concurrency-problem detection
- ***SADA: detecting unsafe DMA accesses***
  - Hardware-access checking
  - Code-path validation to reduce false positives

Outline

1. Introduction to operating system and static analysis
2. **Work1: detecting sleep-in-atomic-context bugs**
3. Work2: detecting concurrency use-after-free bugs
4. Work3: detecting unsafe DMA accesses
5. Our ongoing works and discussion

Background

- Atomic context
  - A special OS kernel state
  - A CPU core is monopolized to execute code without interruption
  - Protect resources from concurrent accesses

- Common examples of atomic context
  - Code is executed while holding a spinlock
  - Code is executed in an interrupt handler

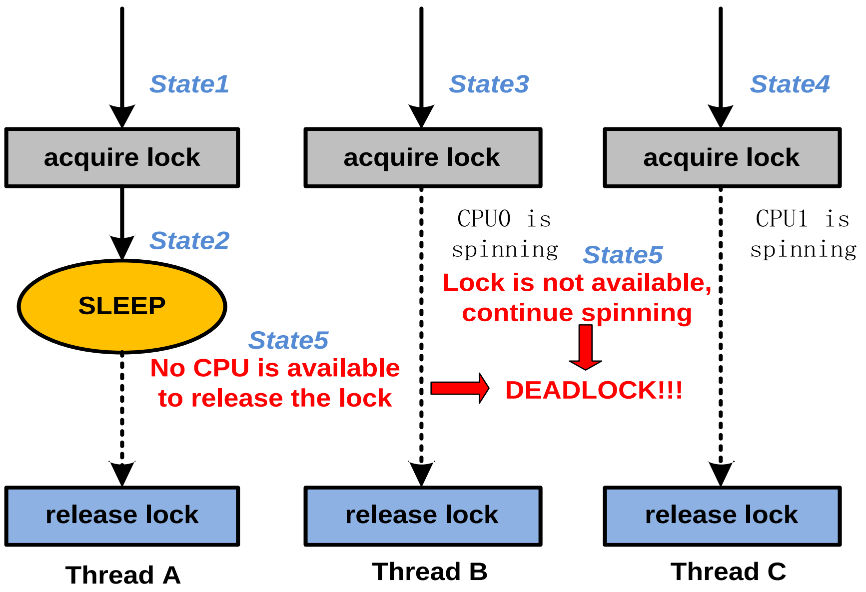
Motivation

- SAC (**S**leep in **A**tomic **C**ontext) bug
  - Sleeping in atomic context is not allowed
  - SAC bug can occasionally cause a system hang or crash when they are triggered at runtime

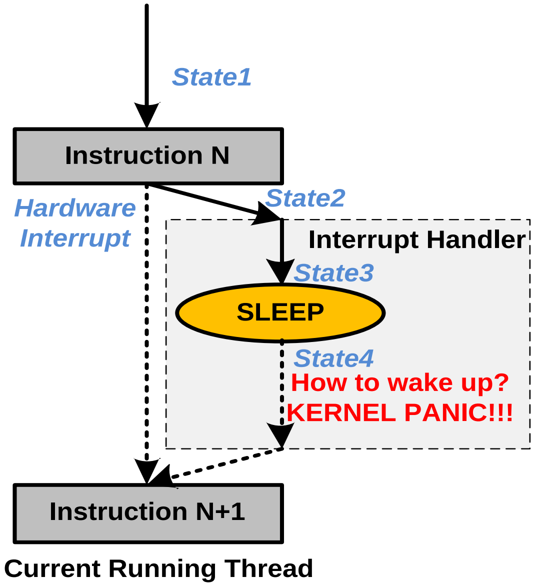
Motivation

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

Sleeping while holding a spinlock



Sleeping in an interrupt handler



Motivation

- Example SAC bug
  - First introduced in Linux 2.6.0 (Dec 2003)
  - First fixed in Linux 2.6.36 (Oct 2010)

FILE: linux-2.6.0/net/xfrm/xfrm_state.c

```
804. int xfrm_user_policy (...) {
       ......
823.   read_lock(...);  // acquire the spinlock
824.   list_for_each_entry(...) {
825.     pol = km->compile_policy(...);
         ......
829.   }
830.   read_unlock(...);  // release the spinlock
       ......
841. }
```

FILE: linux-2.6.0/net/xfrm/xfrm_user.c

```
1072. struct xfrm_policy *xfrm_compile_policy(...) {
        ......
1110.   xp = xfrm_policy_alloc(GFP_KERNEL);
        ......
1122. }
-----------------------------------------------
1168. static struct xfrm_mgr netlink_mgr = {
        ......
1172.   .compile_policy = xfrm_compile_policy,
1173.   .notify_policy = xfrm_send_policy_notify,
1174. };
```

Motivation

- Why do SAC bugs still occur in the Linux kernel?
  - Determining whether an operation can sleep requires OS-specific knowledge
  - SAC bugs occasionally cause problems in real execution and are hard to reproduce at runtime
  - Inter-procedural properties and function pointers need to be carefully considered

Most known SAC bugs are found by manual inspection or runtime failures…

Challenges

- C1: Accuracy and efficiency in code analysis
  - Linux kernel code base is large and complex
  - Flow-sensitive inter-procedural analysis is expensive
- C2: Handling function-pointer calls
  - How to identify real functions referenced by function-pointer calls?
- C3: Dropping repeated and false bugs
  - How to reduce repeated reports and false positives?

Techniques

- C1: Accuracy and efficiency in code analysis

  => Summary-based flow-sensitive analysis
- C2: Handling function-pointer calls

  => Connection-based function-pointer analysis
- C3: Dropping repeated and false bugs

  => Path-check method

T1: Summary-based analysis

- Identify code that may be executed in atomic context
  - Start from each spin-lock function call
  - Start from the entry of each interrupt handling function
  - Maintain an interrupt handling flag, held locks and code paths
- Using function summary to reduce repeated analysis
  - Function location
  - Held locks and interrupt handling flag when the function is handled
  - Sleep-able function call in the function
  - ……

Example

FILE: linux-4.17/drivers/gpu/drm/ttm/ttm_bo_manager.c

```
121. static int ttm_bo_man_takedown(...) {
        ......
126.    spin_lock(...);   // acquire the spinlock
127.    if (drm_mm_clean(...)) {
128.        drm_mm_takedown(...);
            ......
133.    }
134.    spin_unlock(...);   // release the spinlock
135.    return -EBUSY;
136. }
```

FILE: linux-4.17/drivers/gpu/drm/drm_vma_manager.c

```
105. void drm_vma_offset_manager_destroy(...) {
106.    write_lock(...);   // acquire the spinlock
107.    drm_mm_takedown(...);
108.    write_unlock(...);   // release the spinlock
109. }
```

FILE: linux-4.17/drivers/gpu/drm/amd/amdgpu/amdgpu_vram_mgr.c

```
66. static int amdgpu_vram_mgr_fini(...) {
       ......
70.    spin_lock(...);   // acquire the spinlock
71.    drm_mm_takedown(...);
72.    spin_unlock(...);   // release the spinlock
       ......
76. }
```

FILE: linux-4.17/drivers/gpu/drm/drm_mm.c

```
911. static int drm_mm_takedown(...) {
912.    if (WARN(!drm_mm_clean(...),
913.        "Memory manager not clean during takedown.\n"))
914.        show_leaks(...);
915. }
```

FILE: linux-4.17/drivers/gpu/drm/drm_mm.c

```
124. static void show_leaks(...) {
        ......
130.    buf = kmalloc(BUFSZ, GFP_KERNEL);
        ......
153. }
```

Example

Source file	Line	Caller function	Flag
.../ttm_bo_manager.c	126	ttm_bo_man_takedown	START
.../ttm_bo_manager.c	127	ttm_bo_man_takedown	BLOCK
.../ttm_bo_manager.c	128	ttm_bo_man_takedown	ENTER_FUNC
.../drm_mm.c	912	drm_mm_takedown	FUNC_ENTRY
.../drm_mm.c	914	drm_mm_takedown	ENTER_FUNC
......	......	......	......
.../drm_mm.c	130	show_leaks	SLEEP

(a) Create atomic-context code path for *ttm_bo_man_takedown*

Copy and splice the code path

Copy and splice the code path

Source file	Line	Caller function	Flag
.../drm_vma_manager.c	106	drm_vma_offset_manager_destroy	START
.../drm_vma_manager.c	127	drm_vma_offset_manager_destroy	ENTER_FUNC
.../drm_mm.c	912	drm_mm_takedown	FUNC_ENTRY
.../drm_mm.c	914	drm_mm_takedown	ENTER_FUNC
......	......	......	......
.../drm_mm.c	130	show_leaks	SLEEP

(b) Build atomic-context code path for *drm_vma_offset_manager_destroy*

Source file	Line	Caller function	Flag
.../amdgpu_vram_mgr.c	70	amdgpu_vram_mgr_fini	START
.../amdgpu_vram_mgr.c	71	amdgpu_vram_mgr_fini	ENTER_FUNC
.../drm_mm.c	912	drm_mm_takedown	FUNC_ENTRY
.../drm_mm.c	914	drm_mm_takedown	ENTER_FUNC
......	......	......	......
.../drm_mm.c	130	show_leaks	SLEEP

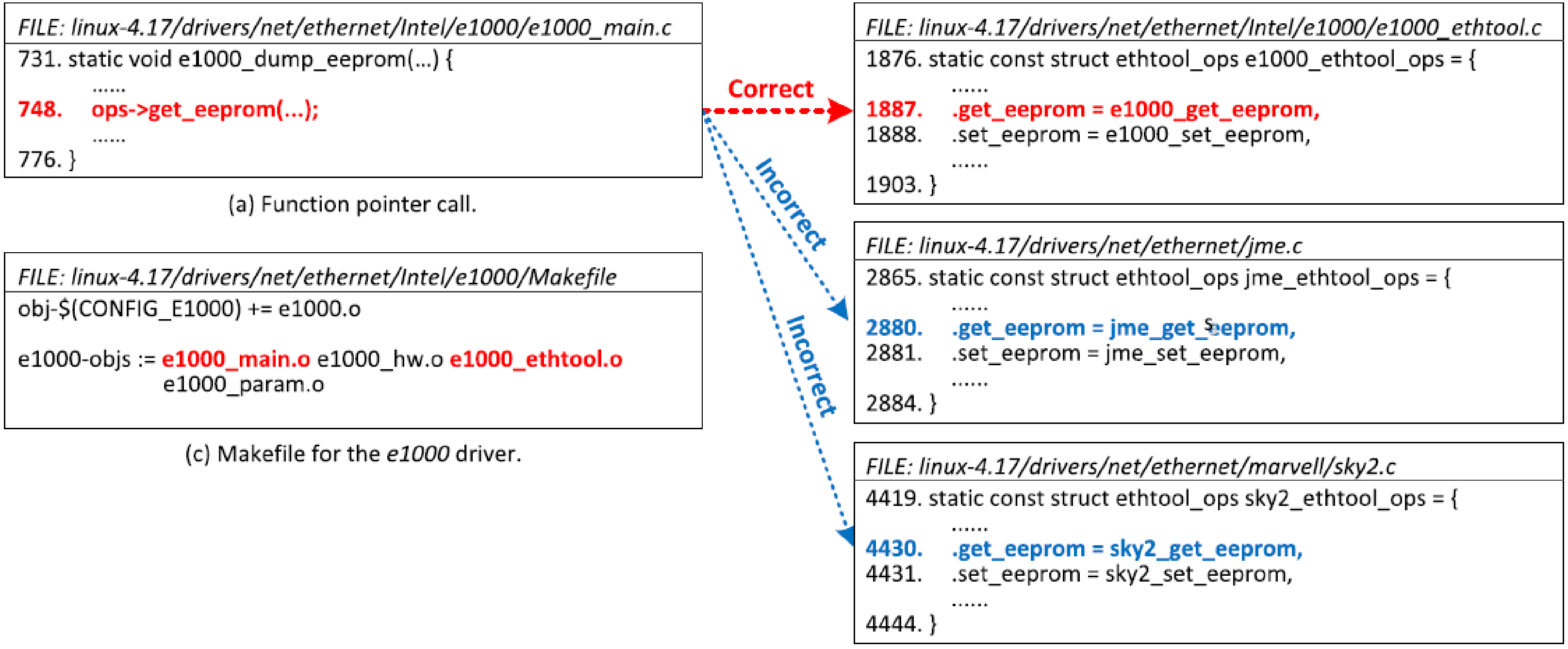
(c) Build atomic-context code path for *amdgpu_vram_mgr_fini*

T2: Connection-based analysis

- Collect candidate functions of function-pointer call
  - Handle function-pointer assignments
  - Perform field-based analysis
- Drop false candidate functions using connections
  - Link-information connection
  - Function-call connection

Link-information connection

- Handle the situations for the same kernel module



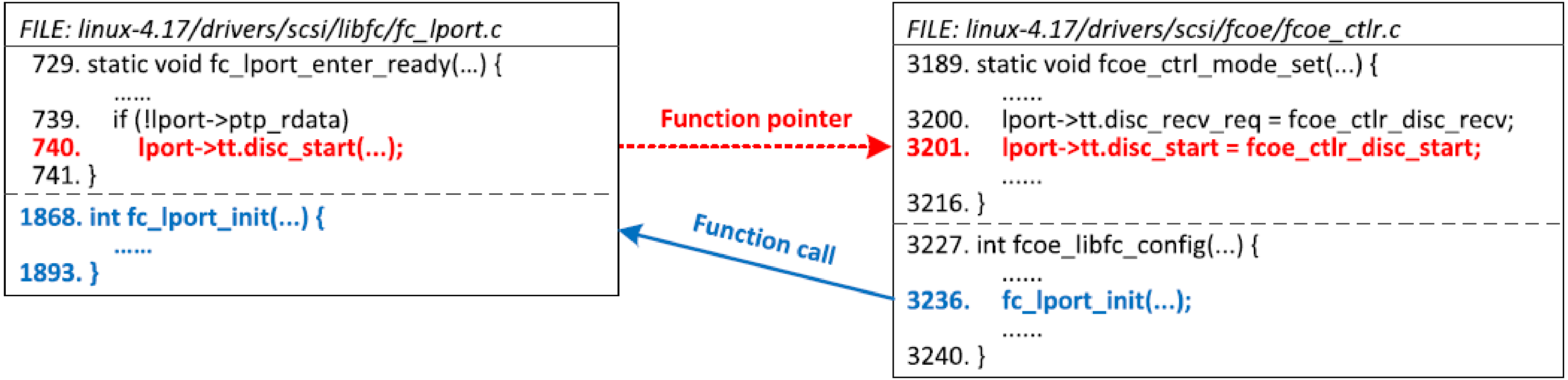
(a) Function pointer call.

(c) Makefile for the *e1000* driver.

(b) Some functions that may be referenced by the function pointer.

Function-call connection

- Handle the situations for different kernel modules



T3: Path-check method

- Drop repeated reports
  - Check the locations of analysis entry and sleep-able function call
- Drop false positives
  - Check path conditions
  - Check key function calls and macros

FILE: linux-4.17/drivers/block/DAC960.c

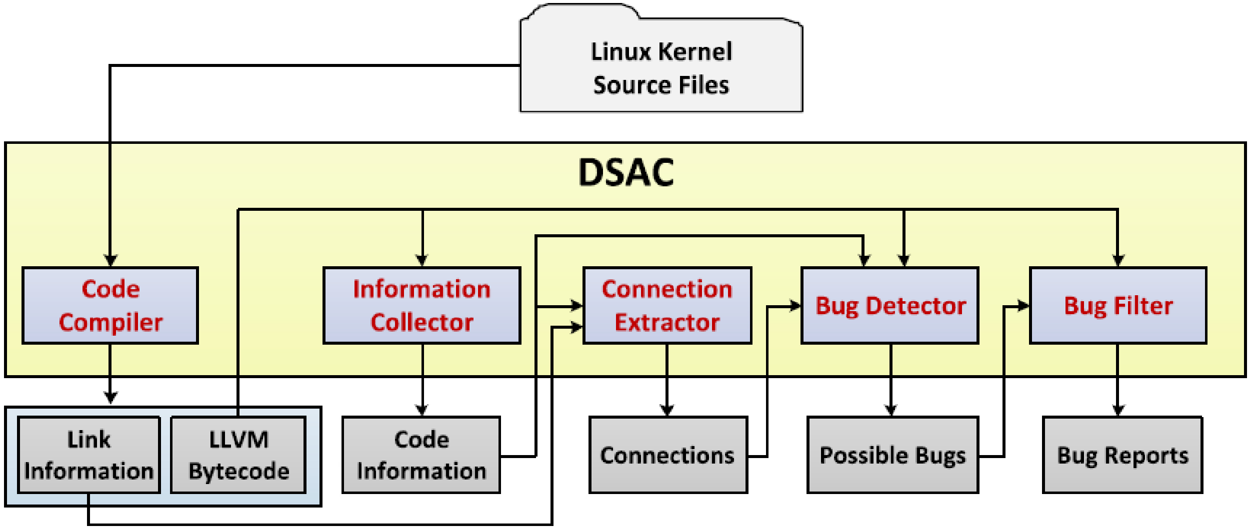
```
781. static void DAC960_ExecuteCommand(…) {
        ……
792.    if (in_interrupt())
793.        return;
794.    wait_for_completion(…);
795. }
```

FILE: linux-4.17/drivers/tty/n_r3964.c

```
837. struct void add_msg(…) {
        ……
846.    pMsg = kmalloc(sizeof(struct r3964_message),
                    error_code ? GFP_ATOMIC : GFP_KERNEL);
        ……
892. }
```

DSAC approach

- Integrate the three key techniques
- Detect SAC bugs in the Linux kernel
- Perform static analysis on LLVM bytecode



Evaluation

- Code analysis

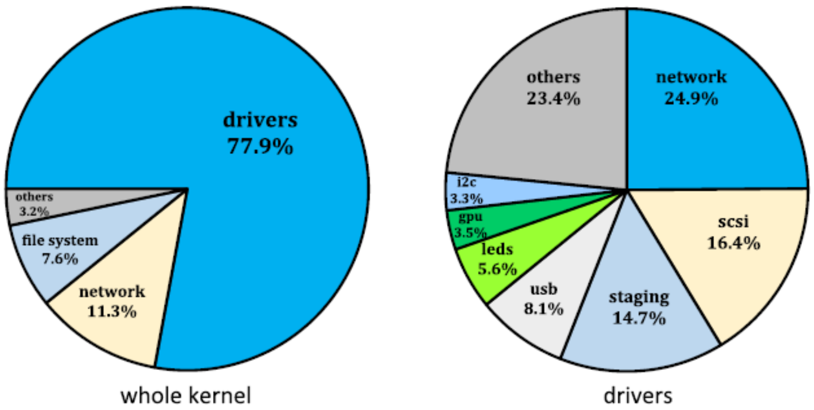
Description		Linux 3.17.2		Linux 4.17	
		DSAC	DSAC_noptr	DSAC	DSAC_noptr
Summary-based analysis	Handled functions	51K	37K	65K	47K
	Function summaries	79K	52K	103K	69K
Function-pointer analysis	Function-pointer calls	14K	-	17K	-
	Handled calls	10K		11K	
	Candidate functions	113K	-	138K	-
	Identified functions	40K	-	45K	-
Bug detection	Found bugs	891	464	1159	615
	Real bugs	805	432	1068	564
Time usage		78m	40m	97m	52m

Evaluation

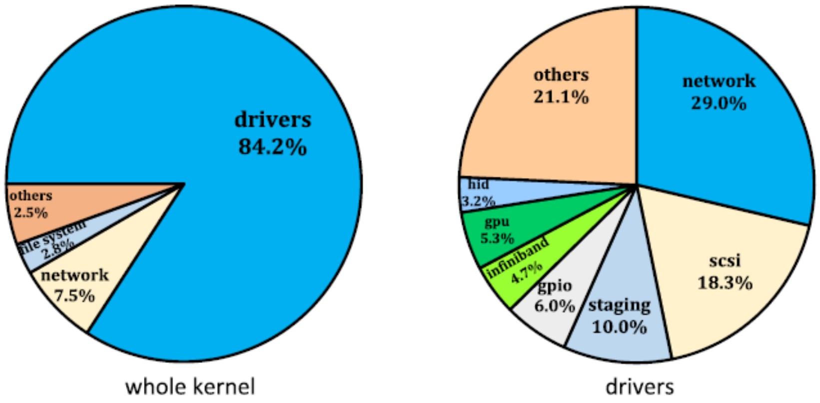
- Linux 3.17.2
  - Find 805 real bugs, with a false positive rate of 9.7%
  - 171 real bugs have been fixed in Linux 4.17
  - Find more 341 real bugs using function-pointer analysis
- Linux 4.17
  - Find 1068 real bugs, with a false positive rate of 7.9%
  - Send 300 randomly-selected bugs to kernel developers, and 220 of them have been confirmed
  - Find more 505 real bugs using function-pointer analysis

Evaluation

- Bug distribution
  - Overall 77% of all bugs occur in drivers
  - Network, SCSI and staging drivers together have >50% of the bugs in drivers



(a) Linux 3.17.2.



(b) Linux 4.17.

Comparison

- Coccinelle BlockLock checker *[ASPLOS'11+TOCS'14]*
  - Both check Linux 2.6.33
  - DSAC makes *allyesconfig* of x86, but BlockLock does not need it
  - BlockLock: 37 bugs related to x86, and 26 of them are real
  - DSAC: 772 bugs, and 719 of them are real
  - 59 real bugs found by DSAC are equivalent to 26 real bugs found by BlockLock
  - DSAC finds 660 more real bugs

Conclusion

- DSAC approach to detect SAC bugs in the Linux kernel
  - Summary-based flow-sensitive analysis
  - Connection-based function-pointer analysis
  - Path-check method
- Find 1068 new real bugs in the Linux kernel
- DSAC finds many bugs missed by existing tools
- Published in ACM TOCS'20
  - *Effective Detection of Sleep-in-Atomic-Context Bugs in the Linux Kernel. Jia-Ju Bai, et al.*

Outline

1. Introduction to operating system and static analysis
2. Work1: detecting sleep-in-atomic-context bugs
3. **Work2: detecting concurrency use-after-free bugs**
4. Work3: detecting unsafe DMA accesses
5. Our ongoing works and discussion

Background

- Use-after-free bugs in device drivers
  - Reliability: may cause system crashes
  - Security: can be exploited to attack the operating system

Background

- Sequential use-after-free bug

```
1. void DriverExit(struct device *pdev) {
2.     kfree(pdev->buf);
3.     pdev->num = 0;
4.     pdev->buf->last = NULL;
5. }
```

Thread 1

- Concurrency use-after-free bug

```
1. void DriverFunc1(struct device *pdev) {
2.     kfree(pdev->buf);
3.     pdev->buf = kmalloc(...)
4.     pdev->buf->last = NULL;
5. }
```

Thread 1

```
1. void DriverFunc2(struct device *pdev) {
2.     spin_lock(...);
3.     pdev->buf->first = NULL;
4.     spin_unlock(...);
5. }
```

Thread 2

Example

- Linux *r8a66597* USB driver

```
FILE: linux-4.19/drivers/usb/host/r8a66597-hcd.c
2304. static const struct hc_driver r8a66597_hc_driver = {
          ......
2320.     .urb_enqueue = r8a66597_urb_enqueue,
          ......
2322.     .endpoint_disable = r8a66597_endpoint_disable,
          ......
2336. }
```

```
FILE: linux-4.19/drivers/usb/host/r8a66597-hcd.c
1885. static int r8a66597_urb_enqueue(...) {
          ......
1895.     spin_lock_irqsave(&r8a66597->lock, flags);
          ......
1905.     if (!hep->hcpriv)  // READ
          ......
1951.     spin_unlock_irqrestore(&r8a66597->lock, flags);
1952.     return ret;
1953. };
```

```
FILE: linux-4.19/drivers/usb/host/r8a66597-hcd.c
1980. static void r8a66597_endpoint_disable(...) {
          ......
1995.     kfree(hep->hcpriv);  // FREE
          ......
2000.     spin_lock_irqsave(&r8a66597->lock, flags);
          ......
2010.     spin_unlock_irqrestore(&r8a66597->lock, flags);
2011. }
```

Lifetime: Jul. 2007 ~ Dec.2018

Fix Commit: c85400f886e3

Study of Linux kernel commits

- Use-after-free commits
  - Jan.2016 ~ Dec.2018 (3 years)

Time	Commits	Drivers	Concurrency	Tool use
2016 (Jan - Dec)	186	111	42 (38%)	26
2017 (Jan - Dec)	478	205	87 (42%)	49
2018 (Jan - Dec)	285	145	66 (46%)	52
Total	949	461	195 (42%)	127

42% of driver commits fixing use-after-free bugs involve concurrency

Study of Linux kernel commits

- Tool use
  - Tools mentioned in driver commits

Tool use	KASAN	Syzkaller	Coverity	Coccinelle	LDV
Type	Runtime	Runtime	Static	Static	Static
Commit	92	28	4	2	1
Concurrency	38	18	0	0	0

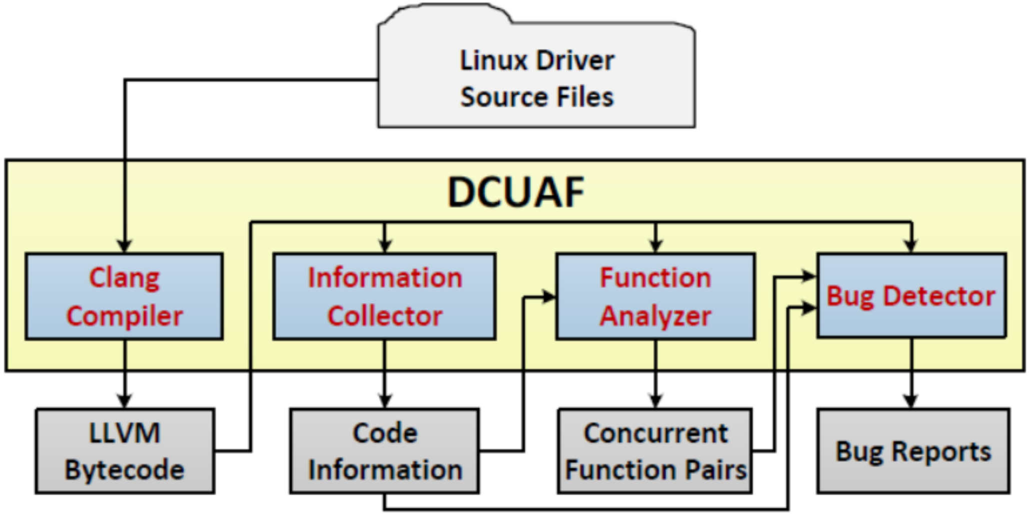
It is important to explore static analysis to detect concurrency use-after-free bugs in device drivers!

Challenges

- Identify driver functions that can be concurrently executed
  - Poor documentation about concurrency
  - Many functions defined in the driver code
- Accuracy and efficiency of code analysis
  - Large size of the Linux driver code base
  - Many function calls across different source files

Approach

- DCUAF
  - Automated and effective approach of detecting concurrency use-after-free bugs in device drivers
  - LLVM-based static analysis



Approach

- Basic idea
  - Step1: Use a ***local-global strategy*** to identify concurrent function pairs from driver source code
  - Step2: Use a ***summary-based lockset analysis*** to detect concurrency use-after-free bugs.

Local-global strategy

- Driver interfaces are the entries of a device driver
  - Kernel-driver interfaces
  - Interrupt handler interfaces
- Driver concurrency is often determined by the concurrent execution of driver interfaces

Local-global strategy

- Examples
  - Linux dl2k and ne2k-pci drivers

FILE: linux-4.19/drivers/net/ethernet/dlink/dl2k.c

```
static const struct net_device_ops netdev_ops = {
    .ndo_open = rio_open,
    .ndo_stop = rio_close,
    .ndo_start_xmit = start_xmit,
    ......
};
-------------------------------------------------
static int rio_open(...) {
    ......
    err = request_irq(irq, rio_interrupt, ...);
    ......                    interrupt_handler
}
```

FILE: linux-4.19/drivers/net/ethernet/8390/ne2k-pci.c

```
static const struct net_device_ops ne2k_netdev_ops = {
    .ndo_open = ne2k_pci_open,
    .ndo_stop = ne2k_pci_close,
    .ndo_start_xmit = ei_start_xmit,
    ......
};
-------------------------------------------------
static int ne2k_pci_open(...) {
    ......
    int ret = request_irq(dev->irq, ei_interrupt, ...);
    ......                    interrupt_handler
}
```

> ".ndo_start_xmit" can be concurrently executed with "interrupt handler"

> ".ndo_open" is never concurrently executed with ".ndo_close"

Local-global strategy

- How to extract concurrent function pairs?
  - **Local stage:** analyze the source code of each driver
  - **Global stage:** statistically analyze the local results of all drivers

Local stage

- S1: identify possible concurrent function pairs
  - Compare lock-acquiring function calls
- S2: drop possibly false concurrent function pairs
  - Collect “ancestors” of the two functions in call graph
  - Drop pairs of functions that have a common “ancestor”
- S3: extract *local concurrent interface pairs*
  - Identify and record driver interface assignments related to concurrent function pairs

Global stage

- S1: gather local concurrent interface pairs of all drivers
- S2: statistically extract *global concurrent interface pairs*
  - Ratio: concurrent pairs / all pairs

Driver Interface 1	Driver Interface 2	Pair	Concurrent	
spi_driver.probe	spi_driver.remove	227	3	✗
file_operations.open	file_operations.close	462	3	✗
hc_driver.urb_enqueue	hc_driver.endpoint_disable	16	9	✓
Interrupt handler	snd_pcm_ops.trigger	49	25	✓

- S3: identify concurrent function pairs in each driver

Summary-based lockset analysis

- Context-sensitive and flow-sensitive lockset analysis
  - Maintain locksets
- Field-based alias analysis
  - Identify the same locks
- Summary-based analysis
  - Reuse the results of already analyzed functions
- Procedure
  - S1: collect the lockset of each variable access
  - S2: check the held locksets of the variable accesses to find bugs

Evaluation

- Local-global strategy

Description		Linux 3.14	Linux 4.19
Code handling	Source files (.c)	7957	13100
	Source code lines	5.1M	7.9M
Local stage	Dropped function pairs	61.4K	99.8K
	Remaining function pairs	40.7K	67.8K
Global stage	Global concurrent interface pairs	694	1497
	Concurrent function pairs	15.6K	69.5K
Time usage		15m	18m

Evaluation

- Bug detection

Description	**Linux 3.14**	**Linux 4.19**
Detected (real / all)	526 / 559	640 / 679
Confirmed / reported	-	95 / 130
Time usage	9m	10m

Some confirmed bugs:

- https://github.com/torvalds/linux/commit/7418e6520f22
- https://github.com/torvalds/linux/commit/2ff33d663739
- https://github.com/torvalds/linux/commit/c85400f886e3

Evaluation

- False positives
  - Alias analysis may incorrectly identify the same locks
  - Flow-sensitive analysis does not validate path conditions
  - ……
- False negatives
  - Function-pointer analysis is not performed
  - Other kinds of synchronization are neglected
  - ……

Conclusion

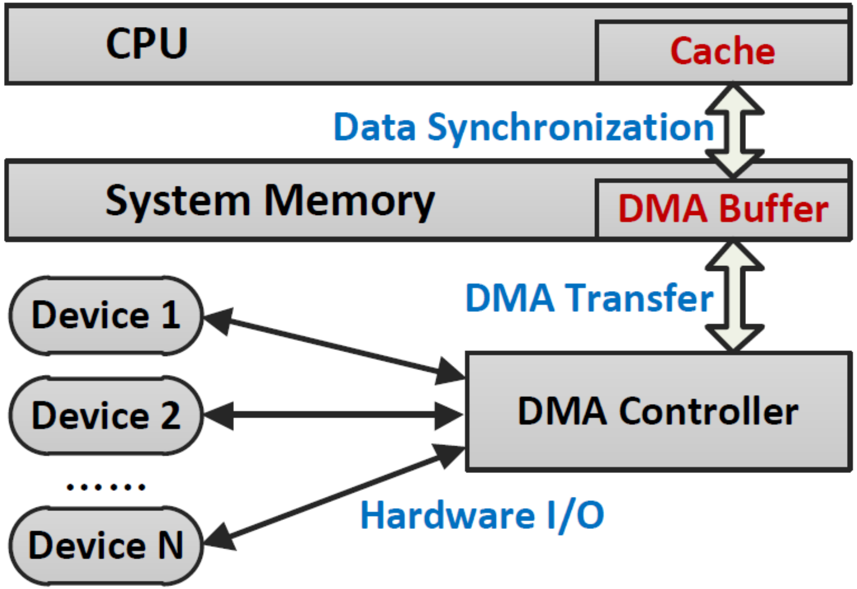
- Concurrency use-after-free bugs are often hard to detect
- DCUAF: automated and effective
  - Local-global strategy of extracting concurrent function pairs
  - Summary-based lockset analysis
- Find hundreds of new real bugs in Linux device drivers
- Published in USENIX ATC’19
  - *Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers. Jia-Ju Bai, et al.*

Outline

1. Introduction to operating system and static analysis
2. Work1: detecting sleep-in-atomic-context bugs
3. Work2: detecting concurrency use-after-free bugs
4. **Work3: detecting unsafe DMA accesses**
5. Our ongoing works and discussion

Background

- DMA is widely used in modern device drivers
  - Direct data transfer between hardware registers and system memory
  - Perform data transfer without CPU involvement



DMA access

- Basic steps
  - S1: Create a DMA buffer
  - S2: Perform a DMA access like a regular variable access

    Read a DMA buffer: `data = dma_buf->data;`

    Write a DMA buffer: `dma_buf->data = data;`
  - S3: Delete a DMA buffer

DMA type

- Streaming DMA buffer
  - It is asynchronously available to both the CPU and hardware device
  - The driver needs to explicitly synchronize the data between hardware registers and CPU cache
  - Each DMA access is relatively cheap
- Coherent DMA buffer
  - It is simultaneously available to both the CPU and hardware device
  - The driver does not need to explicitly synchronize the data between hardware registers and CPU cache
  - Each DMA access is relatively expensive

Security risks of DMA access

- Streaming DMA access
  - After a streaming DMA buffer is created, the driver should not access the content of this buffer, until this buffer is unmapped
  - The driver is allowed to access buffer content during synchronization with hardware registers and CPU cache
- Security risks of violations
  - ***Inconsistent DMA access***
  - Data inconsistency between hardware registers and CPU cache

Example

- Inconsistent DMA access in the Linux *rtl8192ce* driver
  - Introduced in Linux 4.4 (released in Jan. 2016)
  - Fixed in Oct. 2020 by us

FILE: linux-5.6/drivers/net/wireless/realtek/rtlwifi/rtl8192ce/trx.c

```
void rtl92ce_tx_fill_cmddesc(...) {
    ......
    // Streaming DMA mapping
    dma_addr_t mapping = pci_map_single(..., skb->data, ...);
    ......
    struct ieee80211_hdr *hdr = (struct ieee80211_hdr *)(skb->data);
    fc = hdr->frame_control;  // Inconsistent DMA access!
    ......
}
```

Security risks of DMA access

- Coherent DMA access
  - The hardware device can be untrusted, and thus can write bad data into coherent DMA buffers, which are used by the driver
  - The driver should perform correct validation of the data from DMA buffers before using the data
- Security risks of violations
  - ***Unchecked DMA access***
  - Security bugs, such as buffer overflow and invalid-pointer access

Example

- Unchecked DMA access in the Linux *vmxnet3* driver
  - Introduced in Linux 3.16 (released in Aug. 2014)
  - Fixed in Jun. 2020 by us

FILE: linux-5.6/drivers/net/vmxnet3/vmxnet3_ethtool.c

```
693. static int vmxnet3_get_rss(...) {
        ......
696.    struct UPT1_RSSConf *rssConf = adapter->rss_conf;
697.    unsigned int n = rssConf->indTableSize;
        ......
704.    while (n--)
705.        p[n] = rssConf->indTable[n];  // Possible buffer overflow
706.    return 0;
707. }
```

FILE: linux-5.6/drivers/net/vmxnet3/vmxnet3_drv.c

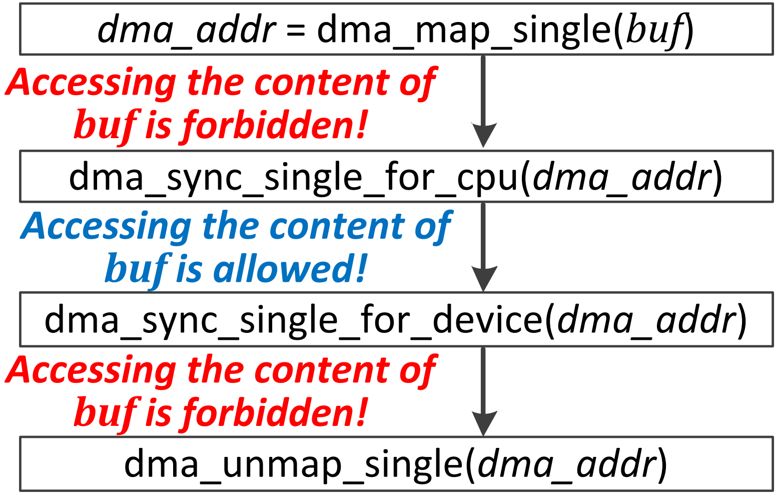
```
3240. static int vmxnet3_probe_device(...) {
          ......
          // Coherent DMA allocation
3373.     adapter->rss_conf = dma_alloc_coherent(...);
          ......
3531. }
```

FILE: linux-5.6/drivers/net/vmxnet3/upt1_defs.h

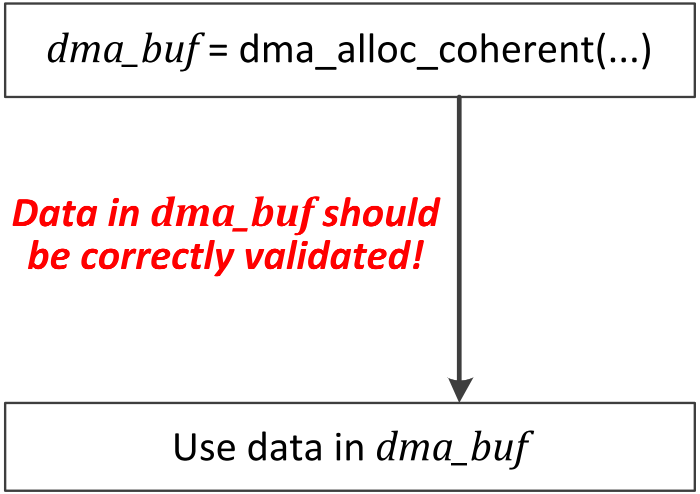
```
80. struct UPT1_RSSConf {
81.     u16 hashType;
        ......
86.     u8 indTable[UPT1_RSS_MAX_IND_TABLE_SIZE]; // Bound is 128
87. }
```

Unsafe DMA access

- Basic rules



Streaming DMA access



Coherent DMA access

Challenges of detecting unsafe DMA access

- ***C1: Identifying DMA access***
  - Each DMA access is implemented as a regular variable access, without calling specific interface functions
  - DMA creation and DMA access often have no explicit execution order from static code observation, namely in a *broken control flow*
- ***C2: Checking the safety of DMA access***
  - Accuracy and efficiency of analyzing large OS code
- ***C3: Dropping false positives***
  - Validating code-path feasibility is difficult and expensive

Key techniques

- ***C1: Identifying DMA access***
  - ***Field-based alias analysis*** to effectively identify DMA access
- ***C2: Checking the safety of DMA accesses***
  - ***Flow-sensitive and pattern-based analysis*** to accurately and efficiently check the safety of DMA access
- ***C3: Dropping false positives***
  - ***Efficient code-path validation method*** to drop false positives and reduce the overhead of using a SMT solver

DMA-access identification

- S1: Handling DMA-buffer creation
  - Identify DMA-creation function calls
  - Collect the information about their return variables, including variable names, data structure types and fields
- S2: Identifying DMA access
  - Check each variable access in the driver
  - If variable name or data structure information matches the collected information, the access is identified to be a DMA access
- Alias analysis is useful to handling variable assignments
  - Intra-procedural, flow-insensitive and Andersen-style alias analysis

DMA-access identification

- Example

FILE: linux-5.6/drivers/isdn/hardware/mISDN/hfcpci.c

```
450.  static int receive_dmsg(...) {
          ......
461.      df = &(hc->hw.fifos)->d_chan.d_rx;  // DMA access
          ......
527.  }
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
1986.  static int setup_hw(...) {
          ......
          // Coherent DMA allocation
2008.     buffer = pci_alloc_consistent(...);
          ......
2015.     hc->hw.fifos = buffer;
          ......
2043. }
```

Match the recorded data structure type and field

Alias

Record data structure type and field

DMA-access safety checking

- Checking streaming DMA access
  - Four patterns about DMA operations
  - Forward and backward flow-sensitive analysis

dma_addr = dma_map_single(*buf*) ***// Start***
↓ ***Forward flow-sensitive analysis***
Read or write the content of *buf* ***// Report!***

Pattern 1

dma_sync_single_for_device(*dma_addr*) ***// Start***
↓ ***Forward flow-sensitive analysis***
Read or write the content of *buf* ***// Report!***

Pattern 2

Read or write the content of *buf* ***// Report!***
↑ ***Backward flow-sensitive analysis***
dma_unmap_single(*dma_addr*) ***// Start***

Pattern 3

Read or write the content of *buf* ***// Report!***
↑ ***Backward flow-sensitive analysis***
dma_sync_single_for_cpu(*dma_addr*) ***// Start***

Pattern 4

DMA-access safety checking

- Checking coherent DMA access
  - Flow-sensitive taint analysis to identify DMA-affected operations
  - Three patterns about security problems

```
FILE: linux-5.6/drivers/net/wireless/intel/iwlwifi/pcie/rx.c
1693. static u32 iwl_pcie_int_cause_ict(...) {
        ......
1714.   do {
          ......
1722.     read = trans_pcie->ict_tbl[...];
          ......
1725.   } while (read);  // Possible bug
        ......
1743. }
------------------------------------------------
2054. int iwl_pcie_alloc_ict(...) {
        ......
        // Coherent DMA allocation
2058.   trans_pcie->ict_tbl = dma_alloc_coherent(...);
        ......
2071. }
```

Pattern 1: Infinite loop polling

```
FILE: linux-5.6/drivers/net/wireless/intel/ipw2x00/ipw2100.c
2661. static void __ipw2100_rx_process(...) {
        ......
        // MASK is 0x0f
2701.   frame_type = sq->drv[i].status_fields & MASK;
        ......
        // Possible bug
2710.   IPW_DEBUG_RX(..., frame_types[frame_type], ...)
        ......
2765. }
------------------------------------------------
4318. static int status_queue_allocate(...) {
        ......
        // Coherent DMA allocation
4325.   q->drv = pci_zalloc_consistent(...);
        ......
4334. }
```

Pattern 2: Buffer overflow

```
FILE: linux-5.6/drivers/net/ethernet/socionext/netsec.c
 931. static int netsec_process_rx(...) {
        ......
 948.   struct netsec_de *de = dring->vaddr + ...;
        ......
 971.   pkt_len = de->buf_len_info >> 16;
        ......
        // Possible bug, as xdp.data is a pointer
1003.   xdp.data_end = xdp.data + pkt_len;
        ......
1059. }
------------------------------------------------
1241. static int netsec_alloc_dring(...) {
        // Coherent DMA allocation
1245.   dring->vaddr = dma_alloc_coherent(...);
        ......
1259. }
```

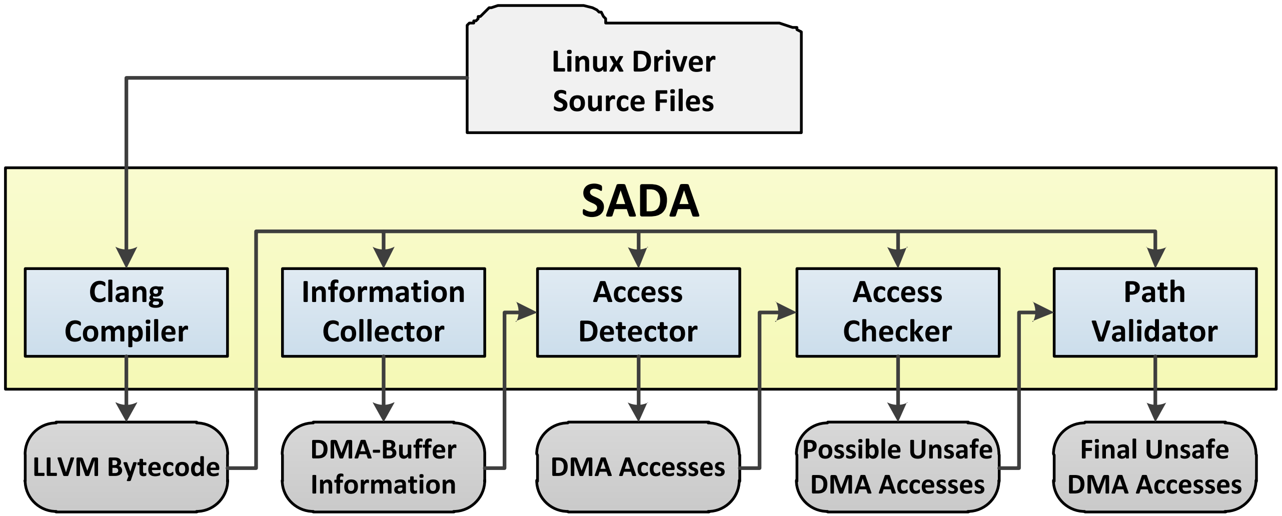
Pattern 3: Invalid pointer access

Code-Path Validation

- S1: Getting path constraints
  - Translate each instruction in the code path to an Z3 constraint
  - ***Example:*** "*a = b + c*" -> "*a == b + c*"
- S2: Adding additional constraints
  - Identify and add constraints that can trigger security bugs
  - ***Example:*** For buffer overflow, add "*frame > MAX_SIZE*" when *frame* is an index to access an array whose bound is *MAX_SIZE*
- S3: Solving all constraints
  - If the constraints cannot be satisfied, the possible unsafe DMA access is identified as a false positive and is dropped

Approach

- **SADA** (Static Analysis of DMA Access)
  - Integrate the three key techniques
  - Statically detect unsafe DMA access in device drivers
  - LLVM-based static analysis



Evaluation

- Detection of unsafe DMA accesses

Description		Linux 5.6
Code handling	Source files (.c)	14.6K
	Source code lines	8.8M
DMA-access identification	Encountered DMA-buffer creation	2,781
	DMA buffers in data structure fields	2,074
	Identified DMA accesses	28,732
DMA-access checking	Unsafe DMA accesses (real / all)	284 / 321
	Inconsistent DMA accesses (real / all)	123 / 131
	Unchecked DMA accesses (real / all)	161 / 190
Time usage	DMA-access identification	62m
	DMA-access checking	208m
	Total time	270m

Evaluation

- 123 inconsistent DMA accesses
  - Direct access after DMA creation: 108
  - Incorrect DMA synchronization: 15
- 161 unchecked DMA accesses
  - Buffer overflow: 121
  - Invalid-pointer access: 36
  - Infinite loop polling: 4
- 105 of the 284 real unsafe DMA accesses have been confirmed by driver developers

Limitations

- False positives
  - The current alias analyses is simple and not accurate enough
  - The path validation can make mistakes in complex cases
  - ……
- False negatives
  - Lack the analysis of function-pointer calls
  - Neglect other patterns of unsafe DMA accesses
  - ……

Conclusion

- DMA is popular in modern device drivers but can introduce security risks in practice
- SADA: static detection of unsafe DMA accesses
  - Field-based alias analysis
  - Flow-sensitive and pattern-based analysis
  - Efficient code-path validation method
- Find 284 real unsafe DMA accesses in Linux 5.6
- Published in USENIX Security'21
  - *Static Detection of Unsafe DMA Accesses in Device Drivers. Jia-Ju Bai, et al.*

Outline

1. Introduction to operating system and static analysis
2. Work1: detecting sleep-in-atomic-context bugs
3. Work2: detecting concurrency use-after-free bugs
4. Work3: detecting unsafe DMA accesses
5. **Our ongoing works and discussion**

Ongoing works

- Static analysis
  - Efficient alias analysis for large-scale software
  - Alias-aware bug detection in OS kernels
  - Deadlock detection in OS kernels
  - ……
- Dynamic analysis
  - Concurrency fuzzing for data-race detection
  - Semantics-aware fuzzing of DBMS
  - Fuzzing distributed systems software
  - ……

Research on systems software analysis

- Program analysis techniques
  - Static analysis
  - Dynamic analysis
- Domain-specific knowledge of specific systems software
  - OS kernels
  - Distributed systems software
  - Network protocols
  - ……
- Limitations of existing generic/specific approaches
- Characteristic techniques

谢谢聆听！
欢迎加入系统软件可靠性研究！

白家驹
清华大学操作系统实验室
https://baijiaju.github.io/

清華大學

Tsinghua University