Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers

Jia-Ju Bai, ***Tsinghua University;*** **Julia Lawall,** ***Sorbonne Université/Inria/LIP6;***
Qiu-Liang Chen and Shi-Min Hu, ***Tsinghua University***

https://www.usenix.org/conference/atc19/presentation/bai

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Effective Static Analysis of Concurrency Use-After-Free Bugs in Linux Device Drivers

Jia-Ju Bai
Tsinghua University

Julia Lawall
Sorbonne University/Inria/LIP6

Qiu-Liang Chen, Shi-Min Hu
Tsinghua University

Abstract

In Linux device drivers, use-after-free (UAF) bugs can cause system crashes and serious security problems. According to our study of Linux kernel commits, 42% of the driver commits fixing use-after-free bugs involve driver concurrency. We refer to these use-after-free bugs as *concurrency use-after-free bugs*. Due to the non-determinism of concurrent execution, concurrency use-after-free bugs are often more difficult to reproduce and detect than sequential use-after-free bugs.

In this paper, we propose a practical static analysis approach named DCUAF, to effectively detect concurrency use-after-free bugs in Linux device drivers. DCUAF combines a local analysis analyzing the source code of each driver with a global analysis statistically analyzing the local results of all drivers, forming a *local-global analysis*, to extract the pairs of driver interface functions that may be concurrently executed. Then, with these pairs, DCUAF performs a *summary-based lockset analysis* to detect concurrency use-after-free bugs. We have evaluated DCUAF on the driver code of Linux 4.19, and found 640 real concurrency use-after-free bugs. We have randomly selected 130 of the real bugs and reported them to Linux kernel developers, and 95 have been confirmed.

1 Introduction

Use-after-free (UAF) bugs in device drivers are often dangerous. They not only cause system crashes, but also can be exploited by hackers to attack the operating system [7, 39, 40]. Among use-after-free bugs, *concurrency use-after-free bugs*, which are due to concurrent execution, are more difficult to detect. Indeed, they are not always triggered at runtime due to the non-determinism of concurrent execution. According to our study of Linux kernel commits, 42% of the driver commits fixing use-after-free bugs involve driver concurrency, and nearly all of these concurrency use-after-free bugs appear to have been found by manual inspection or runtime testing.

To detect use-after-free bugs, many approaches use dynamic analysis [6, 25, 33, 36, 45] to monitor memory accesses at runtime. However, the code coverage and detection results of these approaches heavily rely on the tested workloads. Several approaches [41, 42, 44] use static analysis to detect use-after-free bugs. They can cover much code and find many possible bugs without running the tested programs. However, these approaches are designed to detect use-after-free bugs that occur within sequential execution instead of those due to concurrency. Some static approaches [12, 13, 17, 37, 38] for detecting data races in device drivers can find concurrency use-after-free bugs. However, when identifying which driver functions may be concurrently executed, they assume that all driver interface functions can be concurrently executed [13, 37, 38] or rely on manual guidance [12, 17]. These strategies can introduce many false positives or require much manual work. They often report many data races, but many of the reported races are benign or false positives, and only a few are real concurrency use-after-free bugs.

In this paper, we propose DCUAF, a static analysis approach to detect concurrency use-after-free bugs in Linux device drivers. DCUAF first uses a local-global strategy to extract *concurrent function pairs*, namely the pairs of driver interface functions that can be executed concurrently. Then, with these function pairs, DCUAF performs a summary-based lockset analysis to detect concurrency use-after-free bugs. Our local-global strategy has two stages. In the local stage, DCUAF scans the code of each driver, and identifies calls to lock-acquiring functions, such as `spin_lock`. According to these calls and the driver's function call graph, DCUAF extracts *local concurrent interface pairs*, namely the pairs of driver interfaces that may be concurrently executed for the driver. In the global stage, DCUAF gathers the local concurrent interface pairs of all drivers and performs a statistical analysis to identify the pairs of driver interfaces that are frequently considered to be concurrently executed, from which it produces *global concurrent interface pairs*. Using these interface pairs, for each driver, DCUAF identifies the driver interface functions associated with these pairs as concurrent function pairs for this driver. For each driver function in a concurrent function pair, our lockset analysis analyzes each

variable access in the driver function, and records the lockset that protects this access. Then, for each pair of accesses in the functions of a concurrent function pair, the analysis compares their variables and locksets, and reports concurrency use-after-free bugs. To improve accuracy, our lockset analysis is inter-procedural, context-sensitive and flow-sensitive, and it maintains function summaries to reduce repeated analysis.

We have implemented DCUAF using Clang 6.0 [9] for Linux drivers. DCUAF is fully automatic, given the set of driver source files in the kernel. Overall, we make four main contributions:

- We perform a study of Linux kernel commits, and find that 42% of driver commits fixing use-after-free bugs involve concurrency. Moreover, we infer that nearly all of these concurrency use-after-free bugs have been found by manual inspection or runtime testing. To find more concurrency use-after-free bugs in device drivers, we propose to explore static analysis.
- We propose DCUAF, to detect concurrency use-after-free bugs in device drivers. To our knowledge, DCUAF is the first systematic static approach that targets concurrency use-after-free bugs in device drivers.
- We propose a novel local-global strategy to extract concurrent function pairs.
- We evaluate DCUAF on device drivers in Linux 3.14 and 4.19, and find 559 and 679 concurrency use-after-free bugs, respectively. We manually check these bugs, and find that 526 and 640 bugs are real, respectively. 35 of the real bugs found in Linux 3.14 have been fixed in Linux 4.19. We have randomly selected 130 of the real bugs in Linux 4.19, and reported them to Linux kernel developers. 95 of these bugs have been confirmed.

The remainder of this paper is organized as follows. Section 2 introduces the background. Section 3 shows the challenges of detecting concurrency use-after-free bugs in Linux device drivers and our key techniques to address these challenges. Section 4 introduces DCUAF. Section 5 presents the evaluation. Section 6 discusses how to apply our approach to other kinds of driver problems. Section 7 gives related work, and Section 8 concludes.

2 Background

We first introduce the Linux driver interface model, and then motivate our work by a concurrency use-after-free bug in a Linux driver and by our study of Linux kernel commits.

2.1 Linux Driver Interface Model

A Linux device driver needs to implement some specified *driver interfaces*, including *kernel-driver interfaces* and *interrupt handler interfaces*. A kernel-driver interface is invoked by non-driver threads through function calls when the driver communicates with related management code in the kernel, and an interrupt handling interface is called when a hardware interrupt occurs. We call the driver functions implemented for these driver interfaces *driver interface functions*. The driver interface functions are assigned to driver interfaces through specific data structure fields or specific kernel interfaces. The driver interface functions form the entry points of the driver, so all other functions defined in the driver are called by them. From the kernel's point of view, different drivers of the same device class should have the same functionalities, so drivers in the same device class share the same driver interfaces.

Figure 1 shows two typical Ethernet controller drivers (*dl2k* and *ne2k-pci*) in Linux 4.19. These drivers both define a net_device_ops data structure, containing some function pointer fields. Each network controller driver uses this data structure to communicate with network management code in the kernel, and each function pointer field represents a kernel-driver interface that performs a specific functionality of the driver. For example, in the net_device_ops data structure, the field `ndo_open` is used to open a network device, the field `ndo_stop` is used to close a network device, and the field `ndo_start_xmit` is used to transmit data packets. These drivers also both call a kernel interface `request_irq`, to register their interrupt handler functions through a function pointer argument. According to the functionalities of these driver interfaces, for a given network device instance, the interface net_device_ops.`ndo_start_xmit` can be concurrently executed with the interrupt handler function, but the interface net_device_ops.`ndo_open` is never concurrently executed with the interface net_device_ops.`ndo_stop`. However, whether two driver interfaces can be concurrently executed is often poorly documented in the Linux kernel.

Based on this driver interface model, *driver concurrency is often determined by the concurrent execution of driver interfaces.* Thus, to detect concurrency problems in device drivers, we need to know which driver interfaces can be concurrently executed. We refer to driver interfaces that can be concurrently executed as *concurrent interface pairs*.

2.2 Concurrency Use-After-Free Bug

Use-after-free (UAF) bugs are known to be hard to debug. If the freed memory is not reallocated, and thus not reinitialized, before the use, there is no visible problem, so the bug can linger. If the freed memory is reallocated and reinitialized before the use, then a read use can return an unexpected value and a write use can destroy data relied on by another part of the program. These problems are compounded in the case of kernel code, as the memory can be reinitialized by a different process, allowing information leaks. Some recent works [39, 40] have discussed how to exploit use-after-free bugs to attack an operating system.

```
FILE: linux-4.19/drivers/net/ethernet/dlink/dl2k.c
 98. static const struct net_device_ops netdev_ops = {
 99.    .ndo_open = rio_open,
100.    .ndo_stop = rio_close,
101.    .ndo_start_xmit = start_xmit,
        ......
108. };
-------------------------------------------------
628. static int rio_open(...) {
        ......
640.    err = request_irq(irq, rio_interrupt, ...);
        ......
655. }
```

```
FILE: linux-4.19/drivers/net/ethernet/8390/ne2k-pci.c
203. static const struct net_device_ops ne2k_netdev_ops = {
204.    .ndo_open = ne2k_pci_open,
205.    .ndo_stop = ne2k_pci_close,
206.    .ndo_start_xmit = ei_start_xmit,
        ......
215. };
-------------------------------------------------
432. static int ne2k_pci_open(...) {
        ......
434.    int ret = request_irq(dev->irq, ei_interrupt, ...);
        ......
443. }
```

Figure 1: Examples of driver interfaces.

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/main.c
208. static const struct ieee80211_ops cw1200_ops = {
        ......
215.    .hw_scan = cw1200_hw_scan,
        ......
223.    .bss_info_changed = cw1200_bss_info_changed,
        ......
238. };
```

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/scan.c
 54. int cw1200_hw_scan(...) {
        ......
 91.    mutex_lock(&priv->conf_mutex);
        ......
123.    mutex_unlock(&priv->conf_mutex);
125.    if (frame.skb)
126.       dev_kfree_skb(frame.skb); // FREE
        ......
129. }
```

```
FILE: linux-4.19/drivers/net/wireless/st/cw1200/sta.c
1799. void cw1200_bss_info_changed(...) {
        ......
1807.    mutex_lock(&priv->conf_mutex);
        ......
1849.    cw1200_upload_beacon(...);
        ......
2075.    mutex_unlock(&priv->conf_mutex);
        ......
2081. }
-------------------------------------------------
2189. static int cw1200_upload_beacon(...) {
        ......
2221.    mgmt = (void *)frame.skb->data; // READ
        ......
2238. }
```

Figure 2: A reported bug in the *cw1200* driver in Linux 4.19.

We motivate our work by a real concurrency use-after-free bug in the Linux *cw1200* wireless controller driver. The *cw1200* driver manages the ST-Ericsson CW1200 wireless controller that is used in many embedded systems. The bug was introduced in Linux 3.11 (Sep. 2013) and was fixed 5 years later (Dec. 2018) by us, based on a report generated by DCUAF. Figure 2 shows the driver code related to this bug in Linux 4.19. In the ieee80211_ops data structure, as the driver interfaces represented by the fields `hw_scan` and `bss_info_changed` can be executed concurrently, the driver interface functions `cw1200_hw_scan` and `cw1200_bss_info_changed` can be executed concurrently. In *scan.c*, the function `cw1200_hw_scan` calls `dev_kfree_skb` to free `frame.skb` on line 126, without holding the lock `priv->conf_mutex`. In *sta.c*, the function `cw1200_bss_info_changed` calls `cw1200_upload_beacon`, which reads `frame.skb` on line 2221, while holding the lock `priv->conf_mutex`. Because the free operation is performed without holding a lock but the read operation is performed while holding a lock, a concurrency use-after-free bug may occur. To fix this bug, our commit [2] moved the call to `mutex_unlock` in `cw1200_hw_scan` behind the call to `dev_kfree_skb`.

This example illustrates some reasons why concurrency use-after-free bugs occur in device drivers: (1) Determining which driver interfaces can be executed concurrently requires substantial driver knowledge. In the example, without knowing wireless controller drivers in the Linux kernel well, it may be hard to know that the driver interfaces represented by the fields `hw_scan` and `bss_info_changed` can be executed concurrently. (2) Concurrency use-after-free bugs are not always triggered in real execution and are hard to reproduce. In the example, `cw1200_hw_scan` and `cw1200_bss_info_changed` are not always concurrently executed at runtime. (3) Multiple functions needs to be considered, including the concurrently executed driver interface functions and the functions they call. In the example, three driver functions are involved.

2.3 Our Study of Linux Kernel Commits

To understand the state of the art in detecting use-after-free bugs in the Linux kernel, we study the Linux kernel commits to the mainline kernel [27]. We select the non-merge commits from Jan. 2016 to Dec. 2018 that fix use-after-free bugs, by searching (`git --grep`) for "use after free"and "use-after-free" in the log message, resulting in 949 commits. From them, we identify the driver commits, i.e., those affecting the *drivers* or *sound* directories. For the driver commits, we then study the log messages and code changes to determine: (1) whether the reported bugs are concurrency use-after-free bugs; (2) whether the reported bugs were detected by tools. Table 1 shows the results.

As shown in the table, 49% of the use-after-free related commits are for device drivers. Moreover, 42% of the use-after-free driver commits involve concurrency. Around 65% of the driver use-after-free commits, whether or not they involve concurrency, mention the use of tools, including KASAN [19], Syzkaller [34], Coverity [11], Coccinelle [30] and LDV [24].

Time	Commits	Drivers	Concurrency	Tool
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

Table 1: Linux kernel commits fixing use-after-free bugs.

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

Table 2: Use-after-free bugs found by different analyses and testing tools.

For the remaining driver use-after-free commits, we infer that the reported bugs in these commits are found by manual inspection of the source code and execution failures.

Table 2 breaks down the tool results by the specific tools. KASAN and Syzkaller are runtime testing tools. 120 of the commits fix the bugs found by these tools, including 56 that fix concurrency use-after-free bugs. Coverity, Coccinelle and LDV are static analysis tools. Only 7 of the commits fix bugs found by these tools, with no bug involving concurrency.

From the results, we can infer that nearly all of reported use-after-free bugs in released kernels have been found by manual inspection or runtime testing. However, runtime testing heavily relies on workloads to cover code, and thus it may miss many real bugs in practice. For this reason, it is important to explore static analysis as an alternative to detect concurrency use-after-free bugs in device drivers, but no systematic static tool has yet been proposed. Thus, we aim to design an effective static approach to solve this problem.

3 Challenges and Key Techniques

Our basic idea is to first extract *concurrent function pairs*, namely the pairs of driver interface functions that can be executed concurrently, and then perform a lockset analysis on these pairs of functions to detect concurrency use-after-free bugs. Implementing this idea requires addressing two main challenges:

C1: Extracting concurrent function pairs. Determining which driver functions may be concurrently executed requires substantial driver knowledge. Moreover, the Linux kernel documentation often lacks explicit descriptions about the concurrency of driver interfaces, and thus driver developers may err when implementing the code.

C2: Accuracy and efficiency of code analysis. The Linux driver code base is very large, amounting to 12.6M code lines in our tested version Linux 4.19. Thus, the lockset analysis can be quite time-consuming.

To solve the above challenges, we propose two key techniques. For *C1*, we propose a local-global strategy to extract concurrent function pairs from driver source files in the kernel. For *C2*, we propose a summary-based lockset analysis to detect concurrency use-after-free bugs.

3.1 Local-Global Strategy

Reviewing the example in Figure 2 suggests the following strategy: *concurrent interface pairs could be inferred according to the lock-acquiring function calls in driver interface functions.* In Figure 2, `cw1200_bss_info_changed` and `cw1200_hw_scan` both call `mutex_lock` with a lock variable `priv->conf_mutex`. This information suggests that these two driver functions may be concurrently executed. From this information, we could infer that the related driver interfaces `hw_scan` and `bss_info_changed` in the data structure ieee80211_ops may be a concurrent interface pair. But this kind of inference can be wrong in two common cases:

Case 1. It is possible that two functions that acquire the same lock are actually never concurrently executed. Figure 3 shows an example in the `e100` Ethernet controller driver. The driver functions `e100_enable_irq` and `e100_disable_irq` both call the lock-acquiring function `spin_lock_irqsave` with the same lock variable `nic->cmd_lock`, but they are also both called by the driver function `e100_netpoll`. This suggests that the spinlock acquired in `e100_enable_irq` and `e100_disable_irq` is used by `e100_netpoll` to synchronize with other driver functions, not to synchronize the calls to these two functions with each other.

```
FILE: linux-4.19/drivers/net/ethernet/intel/e100.c
616. static void e100_enable_irq(...) {
        ......
620.    spin_lock_irqsave(&nic->cmd_lock, flags);
        ......
623.    spin_unlock_irqsave(&nic->cmd_lock, flags);
624. };

626. static void e100_disable_irq(...) {
        ......
630.    spin_lock_irqsave(&nic->cmd_lock, flags);
        ......
633.    spin_unlock_irqsave(&nic->cmd_lock, flags);
634. };
-------------------------------------------------
2238. static void e100_netpoll(...) {
        ......
2242.   e100_disable_irq();
        ......
2245.   e100_enable_irq();
2246. }
```

Figure 3: Part of the *e100* driver in Linux 4.19.

Case 2. For two given driver interfaces, only a few of the drivers having the both driver interfaces acquire the same lock in these two driver interfaces, but most drivers do not. Table 3 shows some examples. The first and second columns show the names of involved driver interfaces; the third column shows the number of driver source files that have both

Driver Interface 1	Driver Interface 2	Both	Concurrent
spi_driver.probe	spi_driver.remove	227	3
file_operations.open	file_operations.llseek	462	3
watchdog_ops.start	watchdog_ops.stop	75	1
net_device_ops.ndo_open	ethtool_ops.get_link	124	2

Table 3: Example drivers having the same driver interfaces.

the involved driver interfaces; the fourth column shows the number of driver source files where the involved driver interfaces both acquire the same lock. For example, 227 driver source files have the driver interfaces spi_driver.probe and spi_driver.remove, but only 3 acquire the same lock in both driver interfaces. Indeed, spi_driver.probe is used to initialize an SPI device while spi_driver.remove is used to remove a running SPI device, and a device cannot be initialized and removed at the same time. Thus, the two driver interfaces should not be concurrently executed.

To handle the above two cases, we collect information about the lock usage of each driver as *local* information, and then combine the information about all drivers to perform a *global* statistical analysis. Based on this idea, we propose a *local-global strategy* to extract concurrent function pairs from driver code. The local and global stages handle *Case 1* and *Case 2*, respectively.

Local stage. In this stage, our strategy analyzes each driver source file, and extracts *local concurrent interface pairs*, namely the pairs of driver interfaces that may be concurrently executed for each driver. Figure 4 defines this stage, which has three steps:

Step 1. This step identifies the pairs of possible concurrently executed functions in each driver source file. Firstly, this step clears the result set *pos_func_pair_set*, and collects the set of lock-acquiring function calls as the set *lock_call_set* (lines 1-2). Secondly, this step performs an alias analysis and checks each call in the set *lock_call_set* (lines 4-15). We identify whether the locks are the same by checking whether the related lock variables are aliased. If two different calls in the set have an aliased lock variable, their callers are considered as a pair of possible concurrently executed functions for the source file. In this case, the pair of callers is added to *pos_func_pair_set*. Finally, this step returns the final value of *pos_func_pair_set* (line 17).

Our alias analysis is field-based [16] and focuses on the lock variables stored in data structure fields. We take this strategy for two reasons. Firstly, drivers often use data structure fields to share data (such as locks) between different functions, as illustrated in Figures 2 and 3. Secondly, a variable stored in a data structure field can be explicitly distinguished from other variables using the data structure type and field name. However, for some lock frameworks, their lock-acquiring functions do not have any argument, such as rcu_read_lock. Thus, our analysis does not support these lock frameworks at present.

Step 2. This step filters out the pairs of possible concurrently executed functions that may actually not be executed concurrently. For each pair of possible concurrently executed functions, this step collects and checks the sets of their ancestors in the call graph (lines 2-5). Note that "ancestor" here include callers, callers of the callers, etc. Common ancestors are only collected up to the point of encountering a function that has no caller in the driver but instead is only assigned to a function pointer. As described in Section 2.1, a driver interface that forms an entry point of the driver is often presented as a function pointer stored in a data structure field. If the two driver functions have a common ancestor, the pair of the two functions is deleted from the set *pos_func_pair_set* (line 6), to avoid the possible false positives exemplified by *Case 1*. Finally, this step returns *pos_func_pair_set* (line 9).

Step 1: ***Get the pairs of possible concurrently executed functions***

```
1: pos_func_pair_set := ø;
2: lock_call_set := GetLockCall();
3: for i := 0 to SizeOf(lock_call_set) - 1 do
4:     lock_call1 := lock_call_set[i];
5:     lock_var1 := GetLockVar(lock_call1);
6:     caller_func1 := GetCallerFunc(lock_call1);
7:     for j := i + 1 to SizeOf(lock_call_set) - 1 do
8:         lock_call2 := lock_call_set[j];
9:         lock_var2 := GetLockVar(lock_call2);
10:        caller_func2 := GetCallerFunc(lock_call2);
11:        if lock_var1 is aliased to lock_var2 then
12:            func_pair := <caller_func1, caller_func2>
13:            Add func_pair to pos_func_pair_set;
14:        end if
15:    end for
16: end for
17: return pos_func_pair_set;
```

Step 2: ***Filter out may-false pairs of concurrently executed functions***

```
1: foreach func_pair in pos_func_pair_set do
2:     <caller_func1, caller_func2> := GetFuncPair(func_pair);
3:     func_set1 := GetAncestorFunc(caller_func1);
4:     func_set2 := GetAncestorFunc(caller_func2);
5:     if func_set1 ∩ func_set2 ≠ ø then
6:         Delete func_pair from pos_func_pair_set;
7:     end if
8: end foreach
9: return pos_func_pair_set;
```

Step 3: ***Get local concurrent interface pairs***

```
1: local_interface_pair_set := ø;
2: foreach func_pair in pos_func_pair_set do
3:     <caller_func1, caller_func2> := GetFuncPair(func_pair);
4:     interface_set1 := GetDriverInterface(caller_func1);
5:     interface_set2 := GetDriverInterface(caller_func2);
6:     foreach interface1 in interface_set1 do
7:         foreach interface2 in interface_set2 do
8:             if interface1 == interface2 then
9:                 continue;
10:            end if
11:            interface_pair := <interface1, interface 2>
12:            Add interface_pair to local_interface_pair_set;
13:        end foreach
14:    end foreach
15: end foreach
16: return local_interface_pair_set;
```

Figure 4: The local stage.

Step 3. From the remaining pairs of possible concurrently executed functions, this step extracts the local concurrent interface pairs for the driver. For each function in a pair of possible concurrently executed functions, this step gets the set of driver interfaces that call this function (lines 3-5). Then, this step computes the Cartesian product of the two sets of driver interfaces, omitting the pairs where both driver interfaces are the same (lines 6-14) to avoid the case that different driver functions are assigned to the same driver interface.

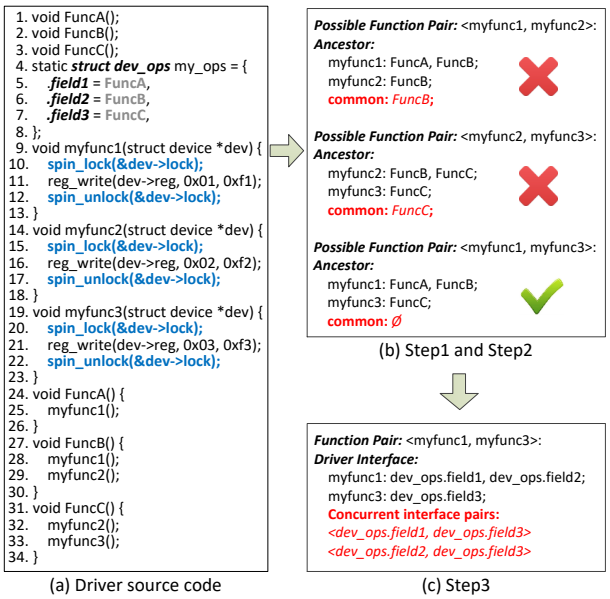

Figure 5: Example of performing the local stage.

Example. To illustrate the local stage, we use some driver-like code shown in Figure 5. In Figure 5(a), the data structure dev_ops has three fields `field1`, `field2` and `field3`, and each of them stores a driver function, namely `FuncA`, `FuncB` and `FuncC`. *Step 1* identifies three lock-acquiring function calls on lines 10, 15 and 20, with the same lock `dev->lock`. Thus, this step gets three pairs of possible concurrently executed functions, namely `<myfunc1, myfunc2>`, `<myfunc2, myfunc3>` and `<myfunc1, myfunc3>`. Then, as shown in Figure 5(b), for each pair, *Step 2* checks the ancestors of the involved functions. `myfunc1` and `myfunc2` have a common caller `FuncB`, and `myfunc2` and `myfunc3` have a common caller `FuncB`, and thus the two pairs are filtered out, leaving only `<myfunc1, myfunc3>`. Finally, as shown in Figure 5(c), *Step 3* uses this function pair to get two lock concurrent interface pairs.

Note that the local stage assumes that a driver function or interface cannot be concurrently executed with itself. The main reason is that only analyzing the lock usage in a function is insufficient to infer whether this function can be concurrently executed with itself. However, this case indeed exists for some drivers, and may make our strategy miss some real local concurrent interface pairs.

Global stage. In this stage, with the local concurrent interface pairs of each driver, we perform a statistical analysis to extract *global concurrent interface pairs* for all drivers.

As shown in Figure 6, this stage first clears the result set *global_interface_pair_set*, and gathers the local concurrent interface pairs of all drivers (lines 1-2). Secondly, this stage handles each concurrent interface pair *interface_pair* in the gathered set (lines 4-9). It calculates the percentage of source files containing the two driver interfaces that have the two driver interfaces as an extracted local concurrent interface pair. This percentage is represent as *ratio* in Figure 6. If $ratio \geqslant R$ (a given threshold), *interface_pair* is considered as a global concurrent interface pair, and is added to the set *global_interface_pair_set*. Finally, this stage returns the final value of *global_interface_pair_set* (line 11).

GlobalStage: ***Get global concurrent interface pairs***

```
1: global_interface_pair_set := ø;
2: local_interface_pair_info_set := GatherLocalInterfacePairSet();
3: foreach interface_pair in local_interface_pair_info_set do
4:     conc_num := GetFileNumOfConcInterfacePair(interface_pair);
5:     file_num := GetFileNumOfInterfacePair(interface_pair);
6:     ratio := conc_num / file_num;
7:     if ratio ⩾ R then
8:         Add interface_pair to global_interface_pair_set;
9:     end if
10: end foreach
11: return global_interface_pair_set;
```

Figure 6: The global stage.

In this stage, the value of the threshold R is important, because the number of extracted global concurrent interface pairs decreases as R becomes larger. Increasing R may cause more false global concurrent interface pairs to be dropped, but more real interface pairs may be missed. We study the impact of the value of R in Section 5.3.

With the extracted global concurrent interface pairs, we identify concurrent function pairs for each driver. Specifically, given two driver interfaces in a driver, if they are in a global concurrent interface pair, the two driver functions associated with these driver interfaces are identified as a concurrent function pair for this driver.

3.2 Summary-Based Lockset Analysis

To improve accuracy and efficiency, our summary-based lockset analysis has the following properties: (1) The analysis is context-sensitive and inter-procedural, in order to maintain locksets and detect bugs across functions calls. (2) The analysis is flow-sensitive to improve accuracy. (3) The analysis uses function summaries to reduce repeated analysis and improve efficiency. (4) The analysis is field-based, and it focuses on the variables stored in data structure fields.

Given driver source code and a concurrent function pair, our lockset analysis has two steps:

Step 1. For each driver function in the concurrent function pair, this step collects the lockset of each variable access (read or write). During the collection, this step uses function summaries to handle called driver functions. Each function summary has the function name, source file name and a set that stores the information about all variable accesses in the function, including the accessed variable, the lockset of the access, the code path that reaches the access from the start of the function and the location of the access.

Figure 7 shows the treatment of a called function *func* by the caller *caller*, with *caller*'s function summary *caller_sum*, the collected lockset *lockset_caller* and the code path *path_info_caller* through *caller* when reaching the call to *func*. Firstly, this step checks whether there is already a call to *func* in the current path by searching *path_info_caller* (lines 1-3). If so, this step returns to avoid infinite looping on recursive calls. Secondly, this step searches the stored function summaries to check whether *func* has been handled (line 4). If so, this step directly uses its function summary *func_sum*. Otherwise, this step performs flow-sensitive analysis to collect information about variable accesses in *func* and then stores *func*'s function summary *func_sum* (lines 5-8). Thirdly, whether an existing function summary is found or a new one is created, this step gets the set *access_info_set* that stores the information about all variable accesses in *func* (line 9). Fourthly, for each variable access in *access_info_set*, this step concatenates its lockset and code path to the end of the *caller*'s lockset and code path, and then stores the information about this variable access in the function summary *caller_sum* (lines 10-16). Using function summaries, repeated flow-sensitive analyses of function definitions are reduced, which can improve the efficiency of the lockset analysis.

```
HandleFunc(func, caller_sum, lockset_caller, path_info_caller)
 1: if func exists in path_info_caller then
 2:     return;
 3: end if
 4: func_sum := FindFuncSummary(func);
 5: if func_sum == ø then
 6:     func_sum := AnalyzeFuncSummary(func);
 7:     StoreFuncSummary(func_sum);
 8: end if
 9: access_info_set := GetAccessInfoSet(func_sum);
10: foreach access_info in access_info_set do
11:     lockset := lockset_caller + GetLockSet(access_info);
12:     path_info := path_info_caller + GetPathInfo(access_info);
13:     SetLockSet(access_info, lockset);
14:     SetPathInfo(access_info, path_info);
15:     AddAccessInfo(access_info, caller_sum);
16: end foreach
```

Figure 7: Handling a called function in our lockset analysis.

Step 2. For each pair of variable accesses in the driver functions of a concurrent function pair, this step compares the accessed variables and held locksets, and reports a concurrency use-after-free bug if: (1) the accessed variables are the same; (2) the intersection of the locksets is empty; (3) one of the accessed variable is used as an argument of a call to a memory freeing function. If both of the accessed variables are used as an argument of a call to a memory freeing function, a double-free bug is also reported.

4 Approach

Based on the two key techniques in Section 3, we propose a practical static approach named DCUAF, to detect concurrency use-after-free bugs in Linux device drivers. We implement DCUAF using Clang 6.0 [9], and perform static analysis on the LLVM bytecode of the driver code. Figure 8 shows the overall architecture of DCUAF.

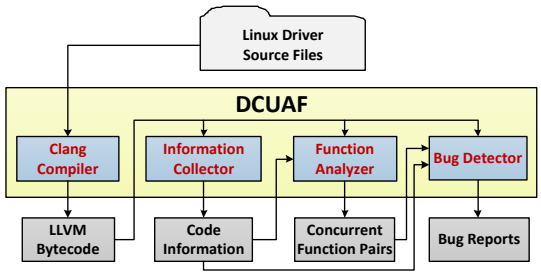

Figure 8: Overall architecture of DCUAF.

Based on this architecture, DCUAF has four phases:

P1: Source code compilation. In this phase, the *Clang compiler* compiles the driver source files and generates their LLVM bytecode files. Because a driver can be implemented across multiple source files, a driver function can call another driver function that is defined in another source file. During linking, DCUAF thus records the set of related source files. This set is used to locate the definition of a called function that is not in the same source file as the caller function.

P2: Code information collection. In this phase, the *information collector* analyzes each LLVM bytecode file, and records code information in a database. The information includes the name and position of each function definition and interrupt handler function, driver functions assigned to function pointers that are stored in data structure fields, the callee and caller functions of each function call, etc. The collected information is used in the remaining phases.

P3: Concurrent function pair extraction. In this phase, with the collected information, the *function analyzer* uses our local-global strategy to analyze LLVM bytecode files. It produces concurrent function pairs for each driver.

P4: Bug detection. In this phase, with the collected code information and extracted concurrent function pairs, the *bug detector* performs our summary-based lockset analysis to analyze each LLVM bytecode file and detect concurrency use-after-free bugs. Some reported bugs may be repeated, when the two bugs are associated with the same driver function and end up at the same variable access but differ in their code paths. Thus, the bug detector also filters out such repeated bug reports, by checking the positions of variable accesses.

Parallelism. The phases P1, P2 and P4 can work on individual LLVM bytecode files independently, and thus they can be parallelized straightforwardly. In P3, the local stage also works on individual LLVM bytecode files, and thus it can be parallelized, too. Only the global stage must be carried out in a single thread to perform the statistical analysis of the collected information. Because this stage does not require processing LLVM bytecode files, it is fast. Thus, overall, DCUAF can greatly benefit from parallelism.

5 Evaluation

We evaluate DCUAF on the source code of Linux device drivers. To cover different kernel versions, we select an old version 3.14 (released in March 2014) and a recent version 4.19 (released in October 2018). Table 4 shows information about the driver code in these kernel versions.

Description	Linux 3.14	Linux 4.19
Release time	March 2014	October 2018
Driver source files (.c)	11.2K	16.6K
Driver source code lines	6.7M	9.5M

Table 4: Properties of the evaluated driver code.

We run the experiments on a Lenovo x86-64 PC with four Intel i5-3470@3.20G processors and 8GB memory. We use the kernel configuration *allyesconfig* to enable all device drivers that can be compiled for the x86 architecture. We compile the driver code using the Clang 6.0 compiler [9]. Because DCUAF can work in parallel, we configure DCUAF to run on 4 threads.

5.1 Extracting Concurrent Function Pairs

DCUAF first uses our local-global strategy to extract concurrent function pairs. In the global strategy, we set $R = 0.2$. Table 5 shows the results for Linux 3.14 and 4.19.

The results show that DCUAF can scale to large code bases. It handles 5.1M and 7.9M source code lines in 7.9K and 13.1K source files in Linux 3.14 and 4.19, respectively, within 20 minutes. The remaining 1.6M and 1.8M source code lines in 3.3K and 3.5K source files in Linux 3.14 and 4.19 are not handled, because they are not enabled by *allyesconfig* for the x86 architecture.

Description		3.14	4.19
Code handling	Handled source files (.c)	7957	13100
	Handle code lines	5.1M	7.9M
Local stage	Dropped function pairs	61.4K	99.8K
	Remaining function pairs	40.7K	67.8K
Global stage	Candidate concurrent interface pairs	7354	11793
	Global concurrent interface pairs	694	1497
	Extracted concurrent function pairs	15.6K	69.5K
Time usage	Code information collection	10m16s	12m20s
	Local stage	4m36s	5m23s
	Global stage	10s	15s
	Total	14m52s	17m58s

Table 5: Results of extracting concurrent function pairs.

Driver Interface 1	Driver Interface 2	Both	Concurrent
tty_operations.write	tty_operations.put_char	14	12
hc_driver.urb_enqueue	hc_driver.endpoint_disable	16	9
ieee80211_ops.bss_info_changed	ieee80211_ops.hw_scan	12	7
uart_ops.set_termios	console.write	21	14
Interrupt handler	snd_pcm_ops.trigger	49	25

Table 6: Examples of global concurrent interface pairs.

Description		3.14	4.19
Bug detection	Filter repeated	348	390
	Final detected (real / all)	526 / 559	640 / 679
	Double free (real / all)	82 / 89	117 / 132
	Interrupt handler (real / all)	25 / 25	23 / 23
Time usage		8m43s	10m15s

Table 7: Results of detecting bugs.

The results also show that our local-global strategy is effective in extracting concurrent function pairs. For example, for Linux 4.19, our strategy extracts 1497 global concurrent interface pairs from the 11,793 candidate concurrent interface pairs identified in the local stage. The remaining interface pairs are not extracted, because they are not identified as being able to execute concurrently by our strategy. For example, DCUAF deletes nearly all interface pairs related to driver initialization and removal (like `probe` and `remove`), which cannot run concurrently in real execution.

Table 6 shows a few of the extracted global concurrent interface pairs in Linux 4.19. The first and second columns show the names of the involved driver interfaces; the third column shows the number of driver source files that have both the involved driver interfaces; the fourth column shows the number of driver source files where the local stage identifies the involved driver interfaces as a concurrent interface pair.

5.2 Detecting Bugs

With the extracted concurrent interface pairs, DCUAF runs our summary-based lockset analysis to detect concurrency use-after-free bugs. To validate whether DCUAF can find known bugs, we use it to check Linux 3.14 drivers. To validate whether DCUAF can find new bugs, we use it to check Linux 4.19 drivers. We also manually check all found bugs to validate accuracy. The results are shown in Table 7.

The results show that DCUAF finds 559 concurrency use-after-free bugs in Linux 3.14. We identify 526 of them as real bugs that are in 108 source files. Among these bugs, 35 have been fixed in Linux 4.19. Thus, DCUAF can find known bugs.

The results show that DCUAF finds 679 concurrency use-after-free bugs in Linux 4.19. We identify 640 of them as real bugs that are in 132 source files. Among these bugs, 372 are also found in Linux 3.14, and thus they have been present for at least 4.5 years. We have randomly selected 130 of these real bugs, and reported them to Linux kernel developers. 95 of them have been confirmed, and 12 of our patches that fix 42 real bugs have been applied (such as commits [1] and [3]) in the kernel code. Thus, DCUAF can find new bugs.

Among the bugs found by DCUAF, many are also double-free bugs. Specifically, DCUAF finds 89 and 132 double-free bugs in Linux 3.14 and 4.19, respectively, and we identify 82 and 89 of them as real bugs. Furthermore, DCUAF finds 25 and 23 bugs that involve interrupt handling in Linux 3.14 and 4.19, respectively, and we identify all of them as real bugs.

Over 60% of the real bugs found by DCUAF are in network, TTY, character and ISDN drivers. Specifically, 359 and 455 of the found real bugs are in these drivers in Linux 3.14 and 4.19, respectively, amounting to 68% and 71% of all found real bugs. Indeed, compared to other drivers, these drivers have more driver functions that can be concurrently executed.

```
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. };
-------------------------------------------------------
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. }
-------------------------------------------------------
2304. static const struct hc_driver r8a66597_hc_driver = {
        ......
2320.   .urb_enqueue = r8a66597_urb_enqueue,
        ......
2322.   .endpoint_disable = r8a66597_endpoint_disable,
        ......
2336. }

========== BUG REPORT ==========
# [READ] r8a66597_urb_enqueue (drivers/.../r8a66597-hcd.c, LINE 1905)
  [LOCK] r8a66597_urb_enqueue (drivers/.../r8a66597-hcd.c, LINE 1895)
# [FREE] r8a66597_endpoint_disable (drivers/.../r8a66597-hcd.c, LINE 1995)
```

Figure 9: A confirmed bug in the *r8a66597* driver.

Figure 9 shows a new confirmed bug found by DCUAF in the Linux 4.19 *r8a66597* driver. The *r8a66597* driver manages the Renesas R8A66597 USB host controller that is used in many embedded systems with USB ports. In the data structure hc_driver, the driver interfaces represented by the fields `urb_enqueue` and `endpoint_disable` are extracted as a global concurrent interface pair (the second row in Table 6) by our local-global strategy. Accordingly, DCUAF considers that the driver functions `r8a66597_urb_enqueue` and `r8a66597_endpoint_disable` may be concurrently executed. In `r8a66597_endpoint_disable`, the variable `hep->hcpriv` is freed on line 1995 without holding the spinlock `r8a66597->lock`. But in `r8a66597_urb_enqueue`, this variable is read on line 1905 while holding the spinlock `r8a66597->lock`. Thus, a concurrency use-after-free bug may occur. The bug report generated by DCUAF shows the related call paths, including the positions of the free, read and lock-acquiring operations. To fix this bug, we move the call to `spin_lock_irqsave` (line 2000) before the call to `kfree` (line 1995). Our patch[1] making this change has been applied by the Linux kernel maintainers.

The found bug in Figure 9 was introduced in Linux 2.6.22 (Jul. 2007), and had existed for over 11 years. Indeed, the USB related documentations in the Linux kernel [35] does not mention that the driver interfaces `urb_enqueue` and `endpoint_disable` can be concurrently executed.

5.3 Result Variation

As described in Section 3.1, the value of R is important. The above results are obtained with $R = 0.2$. To see the variation caused by R, we test $R = 0.1, 0.2, 0.3, 0.4$ and 0.5 on the Linux 4.19 drivers. Figure 10 shows the results.

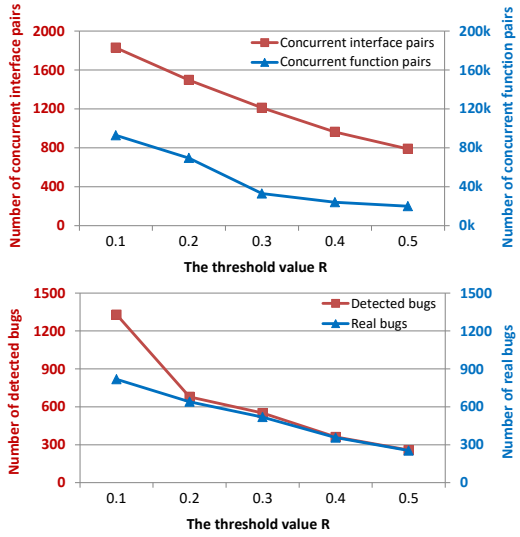

Figure 10: Variation of results by changing the value of R.

The numbers of extracted global concurrent interface pairs and extracted concurrent function pairs both decrease when R becomes larger. In this case, more false concurrent interface pairs and concurrent function pairs are dropped, but more of the dropped pairs are actually real concurrent interface pairs and concurrent function pairs, and thus the bugs involving these pairs are missed. Thus, if the value of R is too small, many false positives will be introduced, and if the value of R is too big, many false negatives will be introduced.

[1] https://lore.kernel.org/patchwork/patch/1025934/

5.4 False Positive and Negative Analysis

5.4.1 False Positives

DCUAF reports 33 and 39 false bugs in Linux 3.14 and 4.19, resulting in false positive rates of 5.9% and 5.7% respectively. These false bugs are introduced for the following reasons:

Firstly, the alias analysis in our lockset analysis is field-based. It cannot distinguish between different variables stored in the same data structure field, and thus it may identify different variables (for locks and data uses) as the same. This reason causes DCUAF to report 12 and 19 false bugs in Linux 3.14 and 4.19.

Secondly, our lockset analysis is flow-sensitive but does not validate path conditions. Thus, it may search infeasible code paths when detecting bugs. This reason causes DCUAF to report 8 and 5 false bugs in Linux 3.14 and 4.19.

Finally, our lockset analysis only handles lock-related function calls, but does not consider other kinds of synchronization. For example, the kernel interface `synchronize_irq` is used to wait until the end of an interrupt handler. Thus, code after the call to `synchronize_irq` should never be concurrently executed with an interrupt handler. But our lockset analysis does not consider this case. This reason causes DCUAF to report 13 and 15 false bugs in Linux 3.14 and 4.19.

Besides the reasons for the false positives observed in our evaluation, there are some other potential reasons for false positives that we have not yet observed in practice. For example, the value of R in our local-global strategy can largely influence the accuracy of extracting concurrent function pairs. If R is not properly set, some extracted global concurrent interface pairs and concurrent function pairs may be false. Moreover, unnecessary locks acquired in the driver code can also influence the accuracy of extracting concurrent function pairs. This case can occur when two driver functions should not be concurrently executed, but they acquire the same lock, because the driver developer is too conservative. Indeed, it has been observed that the Linux kernel does not provide systematic documentation about where locks should be used [28]. Unnecessary locks may cause DCUAF to identify the two driver functions as a concurrent function pair.

5.4.2 False Negatives

To analyze the false negatives of DCUAF, we compare its bug reports with the driver commits fixing concurrency use-after-free bugs identified in Section 2.3. Specifically, we focus on the commits in the few months after the release of Linux 4.19 in October 2018, *i.e.*, between October and December in 2018, resulting in 22 commits. DCUAF finds the bugs in 6 of these commits (including the commit in Figure 2), but misses the bugs in the remaining 16 commits. These bugs are missed for the following reasons:

Firstly, DCUAF lacks function pointer analysis in the local-global strategy and lockset analysis, and thus cannot build complete call graphs of the driver code. As a result, it cannot find real bugs involving code reached through function pointers. This reason causes DCUAF to miss the bugs in 2 commits.

Secondly, our alias analysis is field-based, and may err in complex cases, such as the cases involving function arguments and pointer assignments. It may identify two identical variables (for locks and data uses) as different variables. This reason causes DCUAF to miss the bugs in 4 commits.

Thirdly, our local-global strategy neglects some real cases of driver concurrency. For example, the strategy does not consider that a driver function can be concurrently executed with itself, or that driver functions can create new kernel threads. This reason causes DCUAF to miss the bugs in 2 commits.

Finally, DCUAF does not handle some other cases in driver code, which causes it to miss the bugs in 8 commits. For example, the RCU lock-acquiring functions do not have any argument, so DCUAF cannot use a lock argument to perform static analysis and find related bugs. Moreover, DCUAF does not consider the multi-queue framework that is used in some network and storage drivers. Besides, DCUAF does not consider reference count puts as possible freeing operations.

5.5 Sensitivity Analysis

DCUAF uses two key techniques: a local-global strategy to extract concurrent function pairs in driver code, and a summary-based lockset analysis to reduce repeated analysis. To better understand the value of these two techniques, we modify DCUAF to remove each of them, and evaluate each resulting tool on Linux 4.19 drivers.

Dropping the local-global strategy. We implement two tools by respectively following two assumptions used by previous approaches for detecting data races [37, 38]: (1) all driver interfaces can be concurrently executed; (2) driver interfaces whose field names containing some common keyword pairs for device initialization and deinitialization, including <probe, remove>, <start, stop>, <open, close>, <init, fini> and <resume, suspend>, cannot be concurrently executed . These amount to 257 pairs of driver interfaces. The first tool runs for 350 minutes and reports around 50K bugs. The second tool runs for 302 minutes and reports around 42K bugs. We found that most of the reported bugs found by these tools are false, because many involved driver interfaces that are never concurrently executed. Thus, our local-global strategy indeed reduces false positives in bug detection.

Dropping the summary-based lockset analysis. We implement this tool by dropping function summaries, keeping only the names of functions previously analyzed in the current execution path to avoid infinite loops due to recursion. The resulting tool runs for 850 minutes and then aborts due to insufficient memory. Thus, our summary-based analysis indeed improves the efficiency of the analysis.

6 Discussion

In this section, we discuss how our approach may apply to other kinds of driver problems.

Other concurrency bugs. DCUAF can be used to detect other kinds of concurrency bugs in drivers, by modifying the lockset analysis. For example, it can detect a data race in two driver functions that can be concurrently executed. We have implemented such a prototype approach based on DCUAF. It reports around 149K data races in Linux 4.19 drivers. However, many of the reported data races are benign. Thus, we have focused on a specific kind of serious concurrency bug, namely concurrency use-after-free bugs.

Violations of other properties of driver interfaces. In fact, which driver interfaces can be concurrently executed is an important property of driver interfaces. To identify this property, in DCUAF, we first collect specific code information in each driver and then perform a statistical analysis of the collected information. This idea can be used to identify other important properties of driver interfaces and detect related violations. An example property is whether a driver interface can sleep. If a driver interface is called in atomic context [10], this driver interface cannot call any function that can sleep. Otherwise, a sleep-in-atomic-context (SAC) bug will occur, which can cause a system hang or crash [4]. Following our local-global strategy in DCUAF, for a given driver interface, we can first collect the information about whether the related driver function calls sleep-able functions in each driver; then we can perform a statistical analysis of all the collected information to infer whether this driver interface is in atomic context; and finally using the inference results, we can detect SAC bugs.

7 Related Work

7.1 Detecting Use-After-Free Bugs

Many approaches [6,25,33,36,45] for detecting use-after-free bugs are based on dynamic analysis. They monitor memory accesses at runtime and report bugs according to exact runtime information. They can detect both sequential and concurrency use-after-free bugs. For example, DangSan [36] is an effective use-after-free detection system that can efficiently scale to large numbers of pointer writes and to many concurrent threads. To reduce the runtime overhead of monitoring pointer tracking, DangSan uses a lock-free design inspired by log-structured file systems. This design refrains from using complicated shared data structures and simply opts for append-only per-thread logs for each object in the common case. However, these approaches require workloads that can achieve good code coverage and bug-detection results, and they often introduce runtime overhead.

Several approaches [41, 42, 44] use static analysis to detect use-after-free bugs in user-mode applications. For example, UAFChecker [44] combines taint analysis and symbolic execution to find use-after-free bugs inter-procedurally. CRED [42] is an efficient pointer-analysis-based static analysis to detect use-after-free bugs in large code bases. It uses a spatio-temporal context reduction technique to reduce the exponential number of considered contexts in code analysis. It also uses a multi-stage analysis to efficiently filter out false alarms, and uses a path-sensitive demand-driven method to find the required points-to information.

These static approaches target use-after-free bugs that occur within sequential execution. To do this, they start dataflow analysis from a given free operation, and check whether there is a subsequent use operation. However, they do not consider bugs caused by concurrent execution. Different from these approaches, DCUAF targets concurrency use-after-free bugs. To do this, DCUAF starts alias analysis from two driver functions that may be concurrently executed. Besides, these approaches target user-mode applications, while DCUAF targets device drivers by considering the driver interface model.

7.2 Detecting Concurrency Problems

To detect concurrency problems in device drivers, many existing approaches are based on dynamic analysis or static analysis:

Dynamic analysis. Related dynamic analysis approaches are sampling-based [15, 18] or lockset-based [8, 20, 32]. DataCollider [15] is an effective sampling-based approach to detect data races in the Windows kernel. It randomly samples memory accesses at runtime. To increase the possibility of capturing concurrent accesses to identical memory addresses, it delays the current running thread for a short time, and uses hardware breakpoints to trap any second access during delay. If a second access happens and at least one is a write, a real data race is detected. Eraser [32] was the first lockset-based approach for detecting data races. It instruments binary code to perform runtime monitoring of shared-variable accesses for each running thread, and detects data races by maintaining and checking locksets of shared variables during execution.

Dynamic approaches require associated hardware devices to actually run the tested drivers, which may be hard to obtain in practice. Besides, due to the non-determinism of concurrent execution, they may miss many real concurrency bugs.

Static analysis. Most related static analysis approaches [12, 13, 17, 29, 37, 38] are based on static lockset analysis. RacerX [13] is a well-known static lockset-based approach for detecting data races and deadlocks in OS kernel code. It uses an inter-procedural, flow-sensitive and context-sensitive analysis to maintain and check locksets in code paths, and detects data races and deadlocks. It also ranks the reported bugs. WHOOP [12] is an efficient static lockset-based approach for detecting data races in device drivers. It uses a symbolic pairwise lockset analysis to attempt to prove a driver race-free. It also uses a sound partial-order reduction to accelerate CORRAL [21], an existing concurrency-bug detector.

These static approaches target general concurrency bugs such as data races and atomicity violations, and they often have many false positives (for example, the work on RacerX [13] reports a false positive rate of nearly 50%). They do not focus on concurrency use-after-free bugs. Besides, they assume that all driver interface functions can be concurrently executed [13, 37, 38] or rely on manual guidance [12, 17], which can introduce many false positives or require much manual work. Different from these approaches, DCUAF targets concurrency use-after-free bugs, and uses a local-global strategy to accurately and automatically extract concurrent function pairs from driver source code.

7.3 Mining Code Rules in Systems Software

Some approaches mine implicit code rules in systems software, and then use the mined rules to detect related bugs. They mine rules by statistically analyzing source code [14, 23, 26, 31, 46] or execution traces [5, 22, 43]. PR-Miner [26] uses data mining techniques to extract implicit programming rules from the source code of large code bases. It extracts frequent function-call patterns that occur within a single function. Using the extracted rules, it detects related violations in the source code. PairCheck [5] uses software fault injection to generate test cases that cover error handling code in device drivers, and then runs these test cases to mine resource-acquire and -release rules from execution traces. Using the mined rules, it detects resource leaks in error handling code.

These approaches focus on code rules that occur within sequential execution, such as resource-acquire and -release pairs [5, 31] and function-call sequences [26, 43], but do not consider code rules involving concurrency. Inspired by these approaches, DCUAF uses a statistical analysis of driver code information when extracting concurrent function pairs.

8 Conclusion

In this paper, we have proposed a practical static analysis approach named DCUAF, to effectively and automatically detect concurrency use-after-free bugs in Linux device drivers. DCUAF uses two key techniques: (1) a local-global strategy to extract the pairs of driver interface functions that may be concurrently executed as concurrent function pairs; (2) a summary-based lockset analysis to detect concurrency use-after-free bugs, given two driver functions that may be concurrently executed. We have evaluated DCUAF on the driver code of Linux 4.19, and found 640 real concurrency use-after-free bugs. We have randomly selected 130 of these real bugs and reported them to Linux kernel developers, and 95 have been confirmed.

DCUAF can be improved in some aspects. Firstly, the code analysis in DCUAF can be improved to reduce false positives. For example, DCUAF does not consider path conditions and non-lock-related synchronization primitives in its lockset analysis. Secondly, DCUAF still misses concurrency use-after-free bugs involving complex patterns, such as using reference counters to free objects. Runtime testing tools such as KASAN [19] have found some such bugs in Linux drivers. We will consider these complex patterns in our lockset analysis to find more concurrency use-after-free bugs. Thirdly, besides concurrency use-after-free bugs, DCUAF can be applied to other driver problems, including other concurrency bugs such as data races and violations of other properties of driver interfaces such as sleep-in-atomic-context bugs. Finally, DCUAF only checks Linux drivers at present. We will port DCUAF in other operating systems (such as FreeBSD and NetBSD) to check their driver code.

Acknowledgment

We thank our shepherd Nadav Amit and the anonymous reviewers for their helpful advice on the paper. We also thank the Linux kernel developers who gave useful feedback to us.

References

[1] BAI, J.-J. Linux kernel commit 2ff33d663739: fix some concurrency double-free bugs in the isdn_tty driver. https://github.com/torvalds/linux/commit/2ff33d663739.

[2] BAI, J.-J. Linux kernel commit 4f68ef64cd7f: fix some concurrency use-after-free bugs in the cw1200 driver. https://github.com/torvalds/linux/commit/4f68ef64cd7f.

[3] BAI, J.-J. Linux kernel commit 7418e6520f22: fix a concurrency use-after-free bug in the hfc_pci driver. https://github.com/torvalds/linux/commit/7418e6520f22.

[4] BAI, J.-J., WANG, Y.-P., LAWALL, J., AND HU, S.-M. DSAC: effective static analysis of sleep-in-atomic-context bugs in kernel modules. In *Proceedings of the 2018 USENIX Annual Technical Conference* (2018), pp. 587–600.

[5] BAI, J.-J., WANG, Y.-P., LIU, H.-Q., AND HU, S.-M. Mining and checking paired functions in device drivers using characteristic fault injection. *Information and Software Technology 73* (2016), 122–133.

[6] CABALLERO, J., GRIECO, G., MARRON, M., AND NAPPA, A. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)* (2012), pp. 133–143.

[7] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel

vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)* (2011), pp. 1–5.

[8] CHEN, Q.-L., BAI, J.-J., JIANG, Z.-M., LAWALL, J., AND HU, S.-M. Detecting data races caused by inconsistent lock protection in device drivers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), pp. 366–376.

[9] Clang compiler. http://clang.llvm.org/.

[10] CORBET, J. Atomic context and kernel api design, 2008. https://lwn.net/Articles/274695/.

[11] Coverity. https://scan.coverity.com.

[12] DELIGIANNIS, P., DONALDSON, A. F., AND RAKAMARIC, Z. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)* (2015), pp. 166–177.

[13] ENGLER, D., AND ASHCRAFT, K. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)* (2003), pp. 237–252.

[14] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th International Symposium on Operating Systems Principles (SOSP)* (2001), pp. 57–72.

[15] ERICKSON, J., MUSUVATHI, M., BURCKHARDT, S., AND OLYNYK, K. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)* (2010), pp. 151–162.

[16] HEINTZE, N., AND TARDIEU, O. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the 2001 International Conference on Programming Language Design and Implementation (PLDI)* (2001), pp. 254–263.

[17] HONG, S., AND KIM, M. Effective pattern-driven concurrency bug detection for operating systems. *Journal of Systems and Software (JSS) 86*, 2 (2013), 377–388.

[18] JIANG, Y., YANG, Y., XIAO, T., SHENG, T., AND CHEN, W. DRDDR: a lightweight method to detect data races in Linux kernel. *The Journal of Supercomputing 72*, 4 (2016), 1645–1659.

[19] The Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[20] KernelStrider: Detecting data races in Linux kernel modules by collecting runtime information. https://github.com/euspectre/kernel-strider.

[21] LAL, A., QADEER, S., AND LAHIRI, S. K. A solver for reachability modulo theories. In *Proceedings of the 2012 International Conference on Computer Aided Verification (CAV)* (2012), pp. 427–443.

[22] LAROSA, C., XIONG, L., AND MANDELBERG, K. Frequent pattern mining for kernel trace data. In *Proceedings of the 2008 ACM symposium on Applied computing* (2008), pp. 880–885.

[23] LAWALL, J. L., BRUNEL, J., PALIX, N., HANSEN, R. R., STUART, H., AND MULLER, G. WYSIWIB: a declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 39th International Conference on Dependable Systems and Networks (DSN)* (2009), pp. 43–52.

[24] Linux Driver Verification. http://linuxtesting.org/ldv.

[25] LEE, B., SONG, C., JANG, Y., WANG, T., KIM, T., LU, L., AND LEE, W. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)* (2015).

[26] LI, Z., AND ZHOU, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering (FSE)* (2005), pp. 306–315.

[27] Linux kernel source tree. https://github.com/torvalds/linux.

[28] LOCHMANN, A., SCHIRMEIER, H., BORGHORST, H., AND SPINCZYK, O. LockDoc: trace-based analysis of locking in the Linux kernel. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)* (2019), pp. 11:1–11:15.

[29] LU, S., PARK, S., HU, C., MA, X., JIANG, W., LI, Z., POPA, R. A., AND ZHOU, Y. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of 21st International Symposium on Operating Systems Principles (SOSP)* (2007), pp. 103–116.

[30] PADIOLEAU, Y., LAWALL, J., HANSEN, R. R., AND MULLER, G. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)* (2008), pp. 247–260.

[31] SAHA, S., LOZI, J.-P., THOMAS, G., LAWALL, J. L., AND MULLER, G. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN)* (2013), pp. 1–12.

[32] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS) 15*, 4 (1997), 391–411.

[33] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), pp. 309–318.

[34] Syzkaller: an unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[35] The USB related documentations in the Linux kernel. https://www.kernel.org/doc/Documentation/usb/.

[36] VAN DER KOUWE, E., NIGADE, V., AND GIUFFRIDA, C. DangSan: scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)* (2017), pp. 405–419.

[37] VOJDANI, V., APINIS, K., RÕTOV, V., SEIDL, H., VENE, V., AND VOGLER, R. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)* (2016), pp. 391–402.

[38] VOUNG, J. W., JHALA, R., AND LERNER, S. RELAY: static race detection on millions of lines of code. In *Proceedings of the 2007 International Symposium on Foundations of Software Engineering (FSE)* (2007), pp. 205–214.

[39] WEICHBRODT, N., KURMUS, A., PIETZUCH, P., AND KAPITZA, R. AsyncShock: exploiting synchronisation bugs in Intel SGX enclaves. In *Proceedings of the 2016 European Symposium on Research in Computer Security (ESORICS)* (2016), pp. 440–457.

[40] XU, W., LI, J., SHU, J., YANG, W., XIE, T., ZHANG, Y., AND GU, D. From collision to exploitation: unleashing use-after-free vulnerabilities in Linux kernel. In *Proceedings of the 22nd International Conference on Computer and Communications Security (CCS)* (2015), pp. 414–425.

[41] YAN, H., SUI, Y., CHEN, S., AND XUE, J. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)* (2017), pp. 42–54.

[42] YAN, H., SUI, Y., CHEN, S., AND XUE, J. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)* (2018), pp. 327–337.

[43] YANG, J., EVANS, D., BHARDWAJ, D., BHAT, T., AND DAS, M. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of 28th International Conference on Software Engineering (ICSE)* (2006), pp. 282–291.

[44] YE, J., ZHANG, C., AND HAN, X. UAFChecker: scalable static detection of use-after-free vulnerabilities. In *Proceedings of the 21st International Conference on Computer and Communications Security (CCS)* (2014), pp. 1529–1531.

[45] YOUNAN, Y. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)* (2015).

[46] YUN, I., MIN, C., SI, X., JANG, Y., KIM, T., AND NAIK, M. APISan: sanitizing API usages through semantic cross-checking. In *Proceedings of the 2016 USENIX Security Symposium* (2016), pp. 363–378.