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

Jia-Ju Bai[1], Yu-Ping Wang[1], Julia Lawall[2], Shi-Min Hu[1]
[1]Tsinghua University, [2]Sorbonne Université/Inria/LIP6

Abstract

In a modern OS, kernel modules often use spinlocks and interrupt handlers to monopolize a CPU core for executing concurrent code in atomic context. In this situation, if the kernel module performs an operation that can sleep at runtime, a system hang may occur in execution. We refer to this kind of concurrency bug as a sleep-in-atomic-context (SAC) bug. In practice, SAC bugs have received insufficient attention and are hard to find, as they do not always cause problems in real executions.

In this paper, we propose a practical static approach named DSAC, to effectively detect SAC bugs and automatically recommend patches to help fix them. DSAC uses four key techniques: (1) a hybrid of flow-sensitive and -insensitive analysis to perform accurate and efficient code analysis; (2) a heuristics-based method to accurately extract sleep-able kernel interfaces that can sleep at runtime; (3) a path-check method to effectively filter out repeated reports and false bugs; (4) a pattern-based method to automatically generate recommended patches to help fix the bugs.

We evaluate DSAC on kernel modules (drivers, file systems, and network modules) of the Linux kernel, and on the FreeBSD and NetBSD kernels, and in total find 401 new real bugs. 272 of these bugs have been confirmed by the relevant kernel maintainers, and 43 patches generated by DSAC have been applied by kernel maintainers.

1. Introduction

Concurrency bugs are known to be difficult to debug. Many tools have been proposed to detect common concurrency bugs such as atomicity violations and data races. However, as a kind of concurrency bug, sleep-in-atomic-context (SAC) bugs have received less attention. SAC bugs occur at the kernel level when a sleeping operation is performed in atomic context [10], such as when holding a spinlock or executing an interrupt handler. Code executing in atomic context monopolizes a CPU core, and the progress of other threads that need to concurrently access the same resources is delayed. Thus the code execution in atomic context should complete as quickly as possible. Sleeping in atomic context is in particular forbidden, as it can block CPU cores for long periods and may lead to a system hang.

Even though sleeping in atomic context is forbidden, many SAC bugs still exist, especially in kernel modules, such as device drivers and file systems. The main reasons why SAC bugs continue to occur include: (1) Determining whether an operation can sleep often requires system-specific experience; (2) Testing kernel modules can be difficult, for example, running a device driver requires its associated device; (3) SAC bugs do not always cause problems in real execution, and they are often hard to reproduce at runtime. Recent studies [12, 48] have shown that SAC bugs have caused serious system hangs at runtime. Thus, it is necessary to detect and fix SAC bugs in kernel modules.

Many existing approaches [7, 19, 28, 31, 42] can detect concurrency bugs, but most of them are designed for user-level applications. Some approaches [13, 17, 18, 41, 44] can detect some common kinds of kernel-level concurrency bugs, such as atomicity violations and data races, but they have not addressed SAC bugs. Several approaches [2, 9, 16, 34, 53] can detect common kinds of OS kernel faults, including SAC bugs. But they are not specific to SAC bugs, and most of them [9, 16, 34] are designed to collect statistics rather than reporting specific bugs to the user, making issues such as detection time and false positive rate less important.

In this paper, we propose a static approach named DSAC that targets accurately and efficiently detecting SAC (sleep-in-atomic-context) bugs in kernel modules, and can automatically recommend patches to help fix the detected bugs. DSAC consists of four phases. Firstly, DSAC uses a *hybrid of flow-sensitive and -insensitive analysis* (subsequently referred to as a *hybrid flow analysis*) to analyze the source code, in order to collect the set of functions that are possibly called in atomic context. Secondly, from the collected functions, DSAC exploits a *heuristics-based* method, which uses some heuristics based on the analysis of the call graphs and comments of the kernel code, to extract kernel interfaces that can sleep at runtime. Thirdly, with the extracted sleep-able kernel interfaces, DSAC first reuses the hybrid flow analysis to detect possible bugs, and then uses a *path-check* method to filter out repeated reports and false bugs by validating the code path of each detected bug. Finally, DSAC exploits a *pattern-based* method to automatically generate patches to help fix the bugs. This method analyzes the bug reports generated in the previous phase, and uses common fixing patterns to correct the buggy code.

We have implemented DSAC using LLVM [51]. To validate its effectiveness, we first evaluate DSAC on Linux drivers, which are typical of modules in the Linux kernel. To validate the generality and portability, we then use DSAC to check file systems and network modules in the Linux kernel, and finally use DSAC in FreeBSD and NetBSD to check their kernel source code. The results show that DSAC can indeed accurately and efficiently find real SAC bugs and recommend a number of correct patches to help fix the bugs.

DSAC has four main advantages in practical use:

1) Efficient and accurate code analysis. DSAC uses an efficient inter-procedural and context-sensitive analysis to maintain a lock stack across function calls, which can accurately identify the code in atomic context. All source files of the kernel module are analyzed at once to perform accurate analysis of function definitions.

2) Precise and detailed bug reports. To achieve precise bug detection, DSAC uses a heuristics-based method to accurately extract sleep-able kernel interfaces, and uses a path-check method to filter out repeated reports and false bugs. It also produces detailed reports of the found bugs, including code paths and source file names, for the user to locate and check.

3) Recommended patch generation. With the generated bug reports, DSAC uses a pattern-based method to automatically generate patches to help fix the detected bugs, which can reduce the manual work of bug fixing.

4) High automation, generality and portability. After the user provides the names of spin-lock and -unlock functions, interrupt-handler-register functions and basic sleep-able kernel interfaces, the remaining phases of DSAC are fully automated. DSAC can effectively check common kernel modules, including drivers, file systems and network modules. And it can also be easily ported in another OS to check the kernel code.

In this paper, we make three main contributions:

- We first analyze the challenges in detecting SAC bugs in kernel modules, and then propose four key techniques to address these challenges: (1) a hybrid flow analysis to perform accurate and efficient code analysis; (2) a heuristics-based method to accurately extract sleep-able kernel interfaces in the analyzed kernel modules; (3) a path-check method to effectively filter out repeated reports and false bugs; (4) a pattern-based method to automatically generate recommended patches to help fix the bugs.
- Based on the four techniques, we propose a practical approach named DSAC, to accurately and efficiently detect SAC bugs in kernel modules and automatically recommend patches to help fix the bugs.
- We evaluate DSAC on drivers in Linux 3.17.2 and 4.11.1. We select these kernel versions as they are near the beginning of stable series, and thus the simplest bugs have been fixed in them. We find 200 and 320 real bugs respectively in these versions. 50 real bugs in 3.17.2 have been fixed in 4.11.1, and 209 real bugs in 4.11.1 have been confirmed by kernel maintainers. To validate the generality and portability, we use DSAC to check file systems and network modules in the Linux kernel, and then run it in FreeBSD 11.0 and NetBSD 7.1 to check their kernel code, and find 81 new real bugs. 43 generated patches for the three OS kernels have been applied by kernel maintainers.

The remainder of the paper is organized as follows. Section 2 introduces the motivation. Section 3 presents the challenges and our techniques. Section 4 introduces DSAC in detail. Section 5 presents the evaluation. Section 6 compares DSAC to previous approaches. Section 7 presents limitations and future work. Section 8 gives the related work. Section 9 concludes this paper.

2. Motivation

In this section, we first introduce atomic context, and then motivate our work by an example of a real SAC bug in a Linux driver.

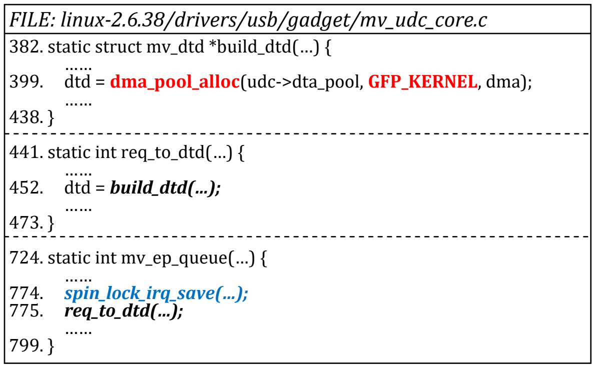
2.1 Atomic Context

Atomic context is an OS kernel state to protect resources from concurrent access, in which the code execution should complete as quickly as possible without able to be rescheduled. In this context, a CPU core is monopolized to execute the code, and the progress of other threads that need to concurrently access the same resources is delayed. Due to this special situation, sleeping in atomic context is forbidden, as it can block CPU cores for long periods and may lead to a system hang.

There are two common examples of atomic context in the kernel, namely *holding a spinlock* and *executing an interrupt handler*. If a thread sleeps when holding a spinlock, another thread that requests the same spinlock will spin on a CPU core to wait until the former thread releases the spinlock. If threads spin on all CPU cores like this, no CPU core will be available for the former thread to release the spinlock, causing a deadlock [11]. If an interrupt handler sleeps, the kernel scheduler cannot reschedule it and a system hang may occur, as the interrupt handler is not backed by a process [29].

原子上下文包括：
（1）spinlock保护的临界区
（2）interrupu handler

Note that atomic context only occurs at the kernel level, as user-level applications are regularly interrupted by the OS scheduler when their time slices end. Though



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

Figure 1: Part of the *usb_gadget* driver code in Linux 2.6.38.

kernel developers often know that sleeping is not allowed in atomic context, many SAC bugs still exist [16, 34], especially in kernel modules.

2.2 Motivating Example

We motivate our work by a real bug in the *usb_gadget* that persisted over 8 releases (1.5 years) from Linux 2.6.38 to Linux 3.7. Figure 1 presents part of the source code for the driver. The function *mv_ep_queue* calls *spin_lock_irqsave* to take a spinlock (line 774) and then calls *req_to_dtd* (line 775). The function *req_to_dtd* calls *build_dtd* (line 452), which calls *dma_pool_alloc* with *GFP_KERNEL* to request a DMA memory pool (line 399). According to the kernel documentation [50], *dma_pool_alloc* called with *GFP_KERNEL* can sleep, thus a SAC bug exists. This bug was first fixed in Linux 3.7, by replacing *GFP_KERNEL* with *GFP_ATOMIC*, which indicates to *dma_pool_alloc* that it cannot sleep.

This example illustrates three main reasons why SAC bugs occur in kernel modules. (1) Determining whether an operation can sleep requires OS-specific knowledge. In this example, without experience in Linux kernel development, it may be hard to know that the function *dma_pool_alloc* called with *GFP_KERNEL* can sleep at runtime. (2) SAC bugs do not always cause problems in real execution and are hard to reproduce at runtime. In this example, the function *dma_pool_alloc* called with *GFP_KERNEL* only sleeps when memory is insufficient. Even in a low-memory situation, this SAC bug is not always triggered at runtime in a multi-core system, because of the non-determinism of concurrent execution. (3) Multiple layers of function calls need to be considered when finding SAC bugs. In this example, the function *dma_pool_alloc* is called across two function levels after *spin_lock_irqsave* is called.

The bug in Figure 1 has been fixed, but many SAC bugs still remain in current kernel modules. Some recent studies [12, 48] have shown that SAC bugs have caused serious system hangs, and these bugs were often hard to locate and reproduce. Thus, to improve the reliability of the operating system, it is necessary to design an approach to detect SAC bugs in kernel modules.

Several previous approaches [2, 9, 16, 34, 53] can detect common kinds of OS kernel faults, including SAC bugs. But they do not target detecting SAC bugs. Indeed, most previous approaches [9, 16, 34] have focused on collecting statistics about different kinds of faults, rather than reporting specific bugs to the user, making accuracy and efficiency less important. Different from them, our goal is to design an effective and accurate approach to detect SAC bugs and to recommend patches to help fix them.

3. Challenges and Techniques

In this section, we first discuss the main challenges in detecting SAC bugs and then propose our techniques to address these challenges.

3.1 Challenges and Overview of Our Solutions

There are four main challenges in detecting SAC bugs in kernel modules:

C1: *Code analysis coverage, accuracy and time.* A key goal in bug detection is to efficiently cover more code and generate accurate results. Running kernel modules is can be difficult (for example, running a driver needs the associated device), and thus we use static analysis to achieve high code coverage without the need to execute the code. Static analysis can be either flow-sensitive or flow-insensitive. Flow-sensitive analysis searches each code path of a branch and can cover all code paths. For this reason, it can produce accurate results, but it often requires much time and memory especially in inter-procedural analysis. Flow-insensitive analysis handles each code line instead of each path. Thus, it is more efficient, but its results may be less accurate. We propose a *hybrid flow analysis* to obtain the advantages of both flow-sensitive and -insensitive analysis. It uses flow-sensitive analysis when its accuracy is expected to be beneficial and falls back to flow-insensitive analysis when full accuracy is not necessary. We will introduce the hybrid flow analysis in Section 3.2.1.

C2: *Sleep-able function extraction.* Determining whether a function can sleep often requires good understanding of the kernel code. Specifically, for a function defined in the kernel module code (referred to as a *module function* subsequently), whether it can sleep depends on whether the called kernel interfaces can sleep. Using this idea, we design a *heuristics-based* method that first collects all kernel interfaces possibly called in atomic context of the kernel module, and then analyzes kernel source code and comments to identify sleep-able ones. We will introduce this method in Section 3.2.2.

C3: *Filtering out repeated and false bugs.* Some detected bugs may be repeated, because they take the spinlock at the same place and call the same sleep-able function, but only differ in their code paths. Moreover, some detected bugs may be false positives, as the analysis does not consider variable value information, and thus may search some invalid code paths. We design a *path-check* method that checks the code path of each detected bug to filter out repeated reports and false bugs. We will introduce it in Section 3.2.3.

C4: *Bug fixing recommendation.* After finding real bugs, the user may manually write patches to fix them. If the patches are incorrect, new bugs may be introduced [21, 46]. To reduce the manual work of bug fixing, we summarize common patterns for fixing SAC bugs, and propose a *pattern-based* method to automatically generate recommended patches to help fix the bugs. We will introduce this method in Section 3.2.4.

3.2 Key Techniques

3.2.1 Hybrid Flow Analysis

（1）上下文敏感
（2）过程间分析

目的：维护spinlock的状态，检测跨函数的原子上下文

Our hybrid flow analysis is used to identify the code in atomic context. It is based on two points: (1) The analysis is context-sensitive and inter-procedural, in order to maintain the spinlock status and detect atomic context across functions calls. (2) The choice of flow-sensitive or -insensitive analysis is made as follows: *if a module function calls a spin-lock or spin-unlock function (this module function is referred to as a target function) or it is called by an interrupt handler, flow-sensitive analysis is used to analyze each code path from the entry basic block; otherwise, flow-insensitive analysis is used to handle each function call made by the function.* In the first case, flow-sensitive analysis is used to accurately maintain the spinlock status and collect code paths for subsequent bug filtering. In the second case, flow-insensitive analysis is used to reduce analysis cost, because in this case, the spinlock status is expected not to change and code paths are unnecessary for subsequent bug filtering.

target function：

文中把调用了spin-lock或者spin-unlock的函数称为traget function。

Our hybrid flow analysis has two steps. The first step is to identify target functions and interrupt handler functions, as flow-sensitive analysis is performed in these functions. For target functions, we analyze the definition of each module function and check whether it calls a spin-lock or spin-unlock function. For interrupt handler functions, we identify the calls to interrupt-handler-register kernel interfaces (like *request_irq* in the Linux kernel), and extract interrupt handler functions from the related arguments. ？？？

对于interrupt handler，会找到中断注册的内核interface（比如request_irq之类的）

The second step is to perform the main hybrid flow analysis. Figure 2 presents the procedure *FlowAnalysis*. It maintains two stacks, namely a path stack (*path_stack*) to store the executed code path, a lock stack (*lock_stack*) to store the spinlock status. A flag (*g_intr_flag*) is used to indicate whether the code is in an interrupt handler. If *lock_stack* is not empty or *g_intr_flag* is *TRUE*, the code is in atomic context. *FlowAnalysis* uses *HanCall* to handle a function call and *HanBlock* to handle a basic block. We introduce them as follows:

lock_stack不为空，或者g_intr_flag为TRUE，表示当前代码处于原子上下文。

```
HanCall(mycall, path_stack, lock_stack)
 1: if lock_stack == ø and g_intr_flag == FALSE then
 2:     return;
 3: end if
 4: if PathHaveExisted(mycall, path_stack) == TRUE then
 5:     return;
 6: end if
 7: AddPathStack(mycall, path_stack);
 8: myfunc := GetCalledFunction(mycall);
 9: HowToFunc(myfunc, path_stack, lock_stack, g_intr_flag);
10: if IsModuleFunc(myfunc) == FALSE then
11:     return;
12: end if
13: if IsTargetFunc(myfunc) == TRUE or g_intr_flag == TRUE then
14:     entry_block := GetEntryBlock(myfunc);
15:     HanBlock(entry_block, path_stack, lock_stack);
16: else
17:     foreach call in FunctionCallList(myfunc) do
18:         HanCall(call, path_stack, lock_stack);
19:     end foreach
20: end if
```

第4～7行代码是前面说的path-check，用于过滤重复检查的路径

```
HanBlock(myblock, path_stack, lock_stack)
 1: if PathHaveExisted(myblock, path_stack) == TRUE then
 2:     return;
 3: end if
 4: AddPathStack(myblock, path_stack);
 5: foreach func_call in FunctionCallList(myblock) do
 6:     if func_call is a call to a spin-lock function then
 7          ChangeLockStack(func_call, lock_stack, LOCK_FUNC);
 8:     else if func_call is a call to a spin-unlock function then
 9:         ChangeLockStack(func_call, lock_stack, UNLOCK_FUNC);
10:     else
11:         HanCall(func_call, path_stack, lock_stack);
12:     end if
13: end foreach
14: if lock_stack == ø and g_intr_flag == FALSE then
15:     return;
16: end if
17: foreach block in SuccessorBlocks(myblock) do
18:     HanBlock(block, path_stack, lock_stack);
19: end foreach
```

```
FlowAnalysis: Main hybrid flow analysis
 1: foreach func in target_func_set do
 2:     lock_block_set := GetLockBlockSet(func);
 3      foreach block in lock_block_set do
 4:         path_stack := ø; lock_stack := ø; g_intr_flag := FALSE;
 5:         HanBlock(block, path_stack, lock_stack);
 6:     end foreach
 7: end foreach
 8: foreach func in intr_handler_func_set do
 9:     path_stack := ø; lock_stack := ø; g_intr_flag := TRUE;
10:     entry_block := GetEntryBlock(func);
11:     HanBlock(entry_block, path_stack, lock_stack);
12: end foreach
```

Figure 2: Hybrid flow analysis.

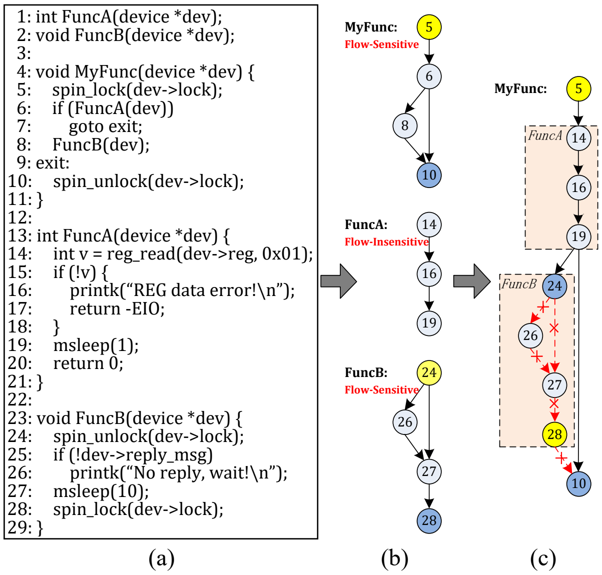
HanCall. It handles the function call *mycall* with the arguments *path_stack* and *lock_stack*, to check if the definition of the function called by *mycall* needs to be handled, and if so to determine if the flow-sensitive or -insensitive analysis should be used. Firstly, *HanCall* checks if *lock_stack* is empty and *g_intr_flag* is *FALSE* (lines 1-3). If so, no spinlock is held and the code is not in an interrupt handler, thus *HanCall* returns. Secondly, *HanCall* uses *path_stack* to check if *mycall* has been analyzed (lines 4-6). If so, it returns to avoid repeated analysis. Thirdly, it adds *mycall* into *path_stack* (line 7), and gets the called function *myfunc* (line 8). Fourthly, *HowToFunc* (line 9) performs specific analyses (shown in Sections 3.2.2 and 4.1.3). Fifthly, *HanCall* checks if *myfunc* is a module function (lines 10-12). If not, it returns. Finally, it handles the definition of *myfunc* (lines 13-20). If *myfunc* is a target function or *g_intr_flag* is *TRUE*, flow-sensitive analysis is used to handle its entry basic block using *HanBlock* (lines 13-15); otherwise, flow-insensitive analysis is used to handle each function call made by *myfunc* using *HanCall* (lines 16-20).

HanBlock. It handles the basic block *myblock* with the arguments *path_stack* and *lock_stack*, to perform flow-sensitive analysis as well as maintain the spinlock status. Firstly, *HanBlock* uses *path_stack* to check if *myblock* has been analyzed (lines 1-3). If so, *HanBlock* returns. Secondly, *HanBlock* adds *myblock* into *path_stack* (line 4). Thirdly, *HanBlock* handles each function call in *myblock* (lines 5-13). If the function call is a call to a spin-lock or spin-unlock function, the top item is added or deleted in *lock_stack*; otherwise, this function call is handled by *HanCall*. Fourthly, it checks if *lock_stack* is empty and *g_intr_flag* is *FALSE* (lines 14-16). If so, it directly returns. Otherwise, each successive basic block of *myblock* is handled using *HanBlock* (linse 17-19).

FlowAnalysis. It performs the main analysis, in two steps. Firstly, each target function is analyzed (line 1-7). For a target function, each basic block that contains a spin-lock function call is an analysis entry. In this case, *path_stack* and *lock_stack* are first set to empty, and *g_intr_flag* is set to *FALSE*. Then, the analysis is started by using *HanBlock* to handle this basic block. Secondly, each interrupt handler function is analyzed (line 8-12). In this case, *path_stack* and *lock_stack* are set to empty, but *g_intr_flag* is set to *TRUE*. Then, the analysis is started by using *HanBlock* to handle the entry basic block of the interrupt handler function.

Our hybrid flow analysis has three main advantages: (1) The functions that are possibly called in atomic context can be accurately detected; (2) Detailed code paths and complete spinlock status are maintained, to help accurately detect atomic context; (3) Many unnecessary paths are not considered to reduce the analysis time. However, a main limitation of our analysis is that variable value information is not considered, which may cause false positives in bug detection.

We illustrate our hybrid flow analysis using some simplified driver-like code shown in Figure 3. As shown in Figure 3(a), the module consists of *MyFunc*, *FuncA* and *FuncB*, where *MyFunc* calls *FuncA* and *FuncB*. Because *MyFunc* and *FuncB* both call *spin_lock* and *spin_unlock*, they are target functions and handled by the flow-sensitive analysis; because *FuncA* does not call spin-lock or spin-unlock functions, it is handled by the flow-insensitive analysis. Figure 3(b) presents the call path of each function, with the code line numbers from Figure 3(a) shown in the vertices. Figure 3(c) shows the call paths used in performing inter-procedural analysis for *MyFunc*. During the analysis, no spinlock is held after the first line of *FuncB* (line 24), thus the next call paths in *FuncB* are not analyzed. In total, only two useful call paths marked in solid edges in Figure 3(c) are handled when analyzing *MyFunc*, and they are the only necessary call paths for atomic context analysis in this case. Other call paths are unnecessary and not analyzed.



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

Figure 3: Example of hybrid flow analysis.

3.2.2 Heuristics-Based Sleep-able Function Extraction

We use some heuristics to accurately extract sleep-able kernel interfaces in the kernel modules. Firstly, we perform our hybrid flow analysis on the analyzed kernel module(s), to collect all kernel interfaces that are possibly called in atomic context, through *HowToFunc* in Figure 2. The collected information is stored into a database as *intermediate results*, including the function name, constant arguments, file name and so on. Secondly, we use some heuristics to inter-procedurally analyze the call graph of each collected kernel interface, and determine whether it can sleep. If a kernel interface satisfies one of the five criteria, we identify it sleep-able:

- It calls a basic sleep-able function, like *schedule* in the Linux kernel and *sleep* in the NetBSD kernel.
- It is called with a specific constant argument indicating it can sleep, like *GFP_KERNEL* in the Linux kernel and *M_WAITOK* in the FreeBSD kernel.
- It calls a specific macro that indicates the operation can sleep, like *might_sleep* in the Linux kernel.
- The comments in or before it contain keywords like *“can sleep”* and *“may block”*.
- It calls an already identified sleep-able kernel interface in the call graph.

To avoid repeated checking, we maintain two cache lists. If a function is marked as sleep-able, it is added to a *sleep-able* list; otherwise it is added to a *non-sleep* list. When analyzing a function, we first check whether the function is in either of these lists.

启发式算法：
搜索内核模块中可能会睡眠的内核接口

第一步：
用前面介绍的hybrid flow analysis搜索所有内核模块，收集所有可能在原子上下文中被调用的内核接口。

第二步：
用启发式的过程间分析去分析上一步中收集的每一个内核接口的call graph，如果一个内核接口满足下面五个条件之一，我们就认为它是可能会睡眠：
(1)它会直接调用会引起睡眠的基本的函数，比如schedule、sleep等。
(2)它在调用过程中使用了某些固定的参数，可能引起睡眠，比如Linux中的GFP_KERNEL等。
(3)它调用了一些指示这里可能会睡眠的宏，比如Linux中的might_sleep等。
(4)函数代码前面的注释里包含诸如“can sleep”、“may block”之类的关键词
(5)在它的call graph中，它调用了一个已经被判定为可睡眠的内核接口。

一个优化，用来防止重复检测的：
维护两个list，一个用来存放已经被标记为sleep–able的内核接口，一个用来存放被标记为non–sleep的内核接口。
批注：这里可能的问题就是sleep–able list是比较好确认的，但是non–sleep的不能一次确认，因为可能会调用一些还未判定的内核接口，而这些还未判定的内核接口有可能sleep–able，这是一个算法问题。

```
FILE: linux-4.11.1/drivers/scsi/ufs/ufshcd.c
504. static int ufshcd_wait_for_register(..., bool can_sleep) {
        ......
515.    if (can_sleep)
516.       usleep_range(...);
517.    else
518.       udelay(...);
        ......
527. }
```

(a) Checking a variable

```
FILE: linux-4.11.1/block/DAC960.c
783. static void DAC960_ExecuteCommand(...) {
        ......
794.    if (in_interrupt())
795.       return;
796.    wait_for_completion(...);
797. }
```

(b) Checking the return value of a kernel interface

Figure 4: Examples of path checks in drivers.

After the extraction, we get the sleep-able kernel interfaces that are possibly called in atomic context of the analyzed kernel modules(s). These kernel interfaces can be used to detect SAC bugs in the kernel module(s).

3.2.3 Path-Check Bug Filtering

We use the detailed code paths recorded in our hybrid flow analysis to filter out repeated and false SAC bugs.

Repeated bug检测：
(1)这个bug的起始block和结束block，都与已经找到的某个bug相同；
(2)这个bug调用的sleep-able的内核接口和(1)中那个已经找到的bug也相同。
满足上面两个条件的，我们认为是重复的bug，会过滤掉。

Firstly, we filter out repeated bugs. For each new possible bug, we check whether its start and end basic block nodes are the same as those of an already detected bug, and whether they call the same sleep-able kernel interface. If both conditions are satisfied, this possible bug is marked as a repeated bug and is filtered out.

DSAC的hybrid flow analysis过程忽略了变量值这个信息。

Secondly, we filter out false bugs, which are mainly introduced by the fact that our hybrid flow analysis neglects variable value information. The best strategy is to validate path conditions [6]. But it is often hard to ensure the accuracy and efficiency when control flow is complex, especially across function calls.

如果一个条件语句，满足如下两个条件之一，那这个条件语句通常是用于判断这里是否允许睡眠：
(1)它检查了函数中的某个参数，比如can_sleep等；
(2)它检查了某些特定内核接口的返回值，比如in_interrupt()等。

By studying the Linux kernel source code, we find a useful and common semantic information for variables: *a conditional that checks a parameter of the containing function or the return value of a specific kernel interface is often used to decide whether sleeping is allowed.* Figure 4 presents two examples in Linux driver code. In Figure 4(a), a conditional checks the function parameter *can_sleep* to decide whether the sleep-able kernel interface *usleep_range* can be called. In Figure 4(b), a conditional checks the return value of the kernel interface *in_interrupt* to check whether the code is executed in an interrupt handler to decide the sleep-able kernel interface *wait_for_completion* can be called. Using this semantic information, we design a straightforward strategy to cover common cases. If the code path of a possible bug satisfies one of the two criteria, we mark this bug as a false bug and filter it out:

如果一条代码路径满足下列两个条件之一，我们就将它标记为一个假的bug，并过滤掉：
(1)代码路径中包含一个条件语句，检查函数的某个参数，参数的名字包含诸如"can_sleep", "atomic", "can_block"等关键字。
(2)代码路径中包含一个条件语句，检查某些特定内核接口的返回值，比如"in_interrupt"等。

- The path contains a conditional that checks a parameter of the containing function, and the name of this parameter contains a keyword like *"can_sleep"*, *"atomic"* and *"can_block"*.
- The path contains a conditional that checks the return value of a kernel interface used to check atomic context, like *in_interrupt* in the Linux kernel.

We propose a path-check method that uses the above steps, to automatically and effectively filter out repeated reports and false bugs.

3.2.4 Pattern-based Patch Generation

By studying Linux kernel patches, we have found four common patterns of fixing SAC bugs:

P1: Replace the sleep-able kernel interface with a non-sleep kernel interface having the same functionality, like *usleep_range* ⇒ *udelay* in Figure 4(a).

P2: Replace the specific sleep-able constant flag with a non-sleep flag, like *GFP_KERNEL* ⇒ *GFP_ATOMIC* in Figure 1.

P3: Move the sleep-able kernel interface to some place where a spinlock is not held.

P4: Replace the spinlock with a lock that allows sleeping, like *spin_lock* ⇒ *mutex_lock* and *spin_unlock* ⇒ *mutex_unlock* in the Linux kernel.

These patterns have different usage scenarios and raise different challenges. Firstly, P1 and P2 can be used for all atomic contexts, while P3 and P4 are only used when holding a spinlock. Secondly, P1 and P2 need simple modifications, while P3 and P4 need much more difficult modifications and are error-prone. Using P3 requires carefully determining where the sleep-able function should be moved to. Using P4 requires modifying all locking and unlocking operations. Thus, using P3 and P4 to automatically generate patches is hard.

We only use P1 and P2 to automatically generate recommended patches, because these patterns are simple and effective. Supporting P3 and P4 is left as future work. The method has three steps. Firstly, the bug is located using its code path, and the relevant fixing pattern (P1 or P2) is selected according to the code. If no relevant pattern is available, no patch is generated. Secondly, the buggy code is corrected by using the selected pattern. Finally, a patch is generated by comparing the corrected code to original code.

This pattern-based method has two advantages. Firstly, it can reduce the manual work of bug fixing. Secondly, by using common fixing patterns, it can ensure the correctness of the generated patches.

4. Approach

Based on the four techniques in Section 3.2, we propose a practical static approach named DSAC, to accurately and efficiently detect SAC bugs in kernel modules, and to recommend patches to help fix the detected bugs. We have implemented DSAC with the Clang compiler [49], and perform static analysis on the LLVM bytecode of the kernel module. Figure 5 presents the architecture of DSAC, which has five parts:

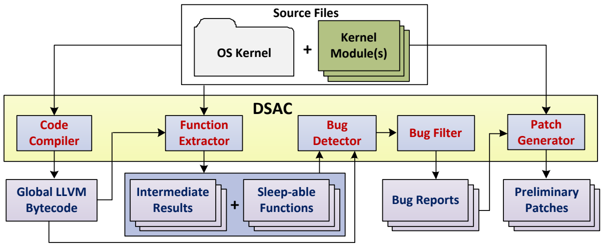


Figure 5: Overall architecture of DSAC.

- ***Code compiler.*** For a given kernel module, this part compiles all the source files of the kernel module into a single LLVM bytecode file.
- ***Function extractor.*** With the LLVM bytecode and kernel source code, this part uses our hybrid flow analysis and heuristics-based method to generate intermediate results and extract sleep-able kernel interfaces called in the kernel module(s).
- ***Bug detector.*** With the extracted sleep-able kernel interfaces and intermediate results, this part performs our hybrid flow analysis to automatically detect possible SAC bugs from the LLVM bytecode.
- ***Bug filter.*** This part uses our path-check method to filter out repeated and false bugs from the possible bugs, and generates the reports of the final detected SAC bugs.
- ***Patch generator.*** With the bug code paths and kernel module source code, this part exploits our pattern-based method to automatically generate recommended patches to help fix the bugs.

Based on the architecture, DSAC consists of four phases which are introduced as follows.

4.1 Function Information Collection

In this phase, DSAC performs two steps:

Firstly, the code compiler compiles each source file of the kernel module into a LLVM bytecode file, and then links all bytecode files into a single bytecode file. This single bytecode file includes all module function definitions, thus all analyses of the kernel module can be directly performed only on this single bytecode file.

Secondly, the function extractor performs the hybrid flow analysis to collect the information about functions that are possibly called when holding a spinlock or in an interrupt handler. The information is stored in a MySQL [52] database as intermediate results, including function name, constant arguments, code file name, etc. The intermediate results will be later used in sleep-able kernel interface extraction and bug detection.

4.2 Sleep-able Kernel Interface Extraction

In this phase, the function extractor first extracts function call graphs of kernel interfaces and comments of these kernel interfaces, and then uses the heuristics-based method to extract sleep-able kernel interfaces. After that, the user can check and modify the extracted sleep-able kernel interfaces as needed.

4.3 Bug Detection

In this phase, DSAC first detects possible SAC bugs, and then filters out repeated reports and false bugs.

Firstly, the bug detector uses the hybrid flow analysis to check whether each extracted sleep-able kernel interface is called in atomic context, which is implemented in *HowToFunc* in Figure 2. If so, a possible bug and its detailed code path to the sleep-able kernel interface call are recorded. To speed up analysis, we use the intermediate results to only check the buggy kernel modules.

Secondly, the bug filter filters out repeated reports and false bugs. Finally, DSAC produces detailed reports for the found bugs (including code paths and source file names), for the user can to locate and check the bugs.

4.4 Recommended Patch Generation

In this phase, the patch generator automatically generates recommended patches to help fix the bugs. Then, the user can use the detailed code paths found in the bug reports to write log messages, and finally submit these patches to kernel maintainers.

5. Evaluation

5.1 Experimental Setup

To validate the effectiveness of DSAC, we first evaluate it on Linux drivers, which are typical kernel modules. To cover different kernel versions, we select an old version 3.17.2 (released in October 2014), and a new version 4.11.1 (released in May 2017). Then, to validate the generality of DSAC, we use it to check file systems and network modules in the Linux kernel. Finally, to validate the portability of DSAC, we run it in FreeBSD and NetBSD to check their kernel code.

We run the experiments on a Lenovo x86-64 PC with four Intel i5-3470@3.20G processors and 4GB memory. We compile the code using Clang 3.2. We use the kernel configuration *allyesconfig* to enable all drivers, file systems and network modules for the *x86* architecture.

To run DSAC, the user performs three steps. Firstly, the user configures DSAC for the checked OS kernel, by providing the names of spin-lock and -unlock functions (like *spin_lock_irq* and *spin_unlock_irq* for the Linux kernel), interrupt-handler-register functions (like *request_irq* for the Linux kernel), and basic sleep-able kernel interfaces (like *schedule* for the Linux kernel). Secondly, the user compiles the source code of the kernel modules and OS kernel using the kernel's underlying *Makefile* and DSAC's compiling script. As a result, DSAC produces sleep-able functions and intermediate results. Finally, the user executes DSAC's bug-detecting script to detect bugs and generate recommended patches. The second and third steps are fully automated.

Table 1: Results of extracting sleep-able kernel interfaces.

Description		3.17.2	4.11.1
Code handling	Handled bytecode files	3377	4396
	Source files (.c)	8321	11153
	Source code lines	7392K	9464K
Hybrid flow analysis	Entry basic blocks	32167	37770
	Handled INTR functions	578	673
Heuristic extraction	Recorded functions	3104	3613
	Sleep-able kernel interfaces	70 (51)	94 (63)
Time usage	Original compilation	47m21s	55m34s
	DSAC total	108m43s	129m58s
	DSAC pure	61m22s	74m22s

5.2 Extracting Sleep-able Kernel Interfaces

We first extract the sleep-able kernel interfaces that are called in atomic context of the drivers. Table 1 presents the results for Linux 3.17.2 and 4.11.1. We make the following observations:

1) DSAC can scale to large code bases. It handles 7M and 9M source code lines from 8K and 11K source files. And the analysis is started from many entry basic blocks and many interrupt handler (INTR) functions.

2) Our heuristics-based method can efficiently extract real sleep-able kernel interfaces that are called in atomic context of the analyzed drivers. In Linux 3.17.2 and 4.11.1, 70 and 94 sleep-able kernel interfaces are respectively identified from among 3104 and 3613 different kernel interfaces (candidate functions) that are possibly called in atomic context. We manually check the kernel interfaces identified as sleep-able, and find that all of them can sleep at runtime. Over 97% of the candidate functions are automatically filtered out, thus the manual work of checking these functions is saved.

3) Our code analysis is efficient. DSAC respectively spends around 108 and 129 minutes on handling 8K and 11K driver source files, including the compilation time of these source files using the Clang compiler. Excluding compilation time, DSAC spends 61 and 74 minutes respectively, amounting to less than 0.44 seconds per source file.

4) Many of the extracted sleep-able kernel interfaces are related to resource handling (such as allocation and release). The data in parentheses present the number of these kernel interfaces, which amount to more than 60% of all the sleep-able kernel interfaces.

5.3 Detecting Bugs and Generating Patches

Based on the above extracted sleep-able kernel interfaces, we use DSAC to perform bug detection and recommend patches. Firstly, to validate whether DSAC can find known fixed bugs, we use DSAC to check the drivers in the older Linux kernel version 3.17.2. We do not generate patches in this case, because this kernel version is very old. Secondly, we use DSAC to check the drivers in Linux kernel version 4.11.1, in order to validate whether DSAC can find new bugs and recommend patches to help fix them. Finally, to validate the accuracy of DSAC, we manually check all detected bugs to identify whether they are real bugs. Table 2 shows the results. We have the following findings:

Table 2: Detected bugs and generated patches in drivers.

Description		3.17.2	4.11.1
Detected bugs	Repeated filtered	479630	629924
	False filtered	282	430
	Final detected	215	340
	Interrupt handling	7	17
	Real	200	320
Patch generation	P1	-	28
	P2	-	15
	Total	-	43
Time usage	Bug detection	6m31s	8m46s
	Patch generation	-	1m02s
	Total	6m31s	9m48s

1) Our path-check filtering method in DSAC is effective, automatically filtering out many repeated reports and false bugs.

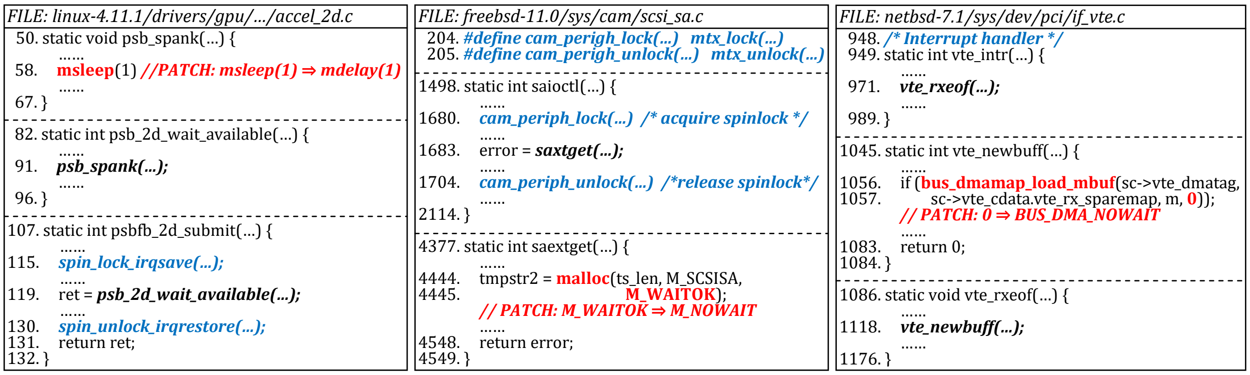
2) Of the 215 bugs reported by DSAC in the drivers of Linux 3.17.2, we have identified 200 as real bugs, 50 of which have been fixed in Linux 4.11.1. By reading the messages in relevant Linux driver mailing lists, we find that kernel maintainers confirmed that these fixed bugs could cause serious problems, like system hangs. The results indicate DSAC can indeed find known bugs.

3) Of the 340 bugs reported by DSAC in the drivers of Linux 4.11.1, we have identified 320 as real bugs. 150 bugs are inherited from the legacy code in 3.17.2, and 170 bugs are introduced by new functionalities and new drivers. We have reported all the bugs that we identified as real to kernel maintainers. As of January 2018, 209 bugs have been confirmed, and replies for the other bugs have not been received. The results indicate DSAC can indeed find new real bugs.

4) DSAC can accurately find real bugs in our evaluated driver code. The false positive rates are respectively only 7.0% and 5.9% in the drivers of Linux 3.17.2 and 4.11.1, based on our identification of real bugs. Reviewing the driver source code, we find these false positives are mainly introduced by the fact that some invalid code paths are searched by our hybrid flow analysis and our path-check method does not filter them out.

5) Few of the detected bugs are in interrupt handlers (7 bugs in 3.17.2, and 17 bugs in 4.11.1). Indeed, driver developers often write clear comments to mark the driver functions that are called from an interrupt handler, to prevent calling sleep-able functions in these functions.

6) DSAC automatically and successfully generates 43 patches that it recommends to help fix 82 real bugs in Linux 4.11.1. Table 2 classifies the patches by the pattern in Section 3.2.4 that is used. We manually review these patches, add appropriate descriptions, and then submit them to the relevant kernel maintainers. As of January 2018, 30 patches have been applied. 2 patches



```
FILE: linux-4.11.1/drivers/gpu/.../accel_2d.c
 50. static void psb_spank(...) {
         ......
 58.     msleep(1) //PATCH: msleep(1) ⇒ mdelay(1)
         ......
 67. }
------------------------------------------------
 82. static int psb_2d_wait_available(...) {
         ......
 91.     psb_spank(...);
         ......
 96. }
------------------------------------------------
107. static int psbfb_2d_submit(...) {
         ......
115.     spin_lock_irqsave(...);
         ......
119.     ret = psb_2d_wait_available(...);
         ......
130.     spin_unlock_irqrestore(...);
131.     return ret;
132. }
```

(a) Linux *gma500* driver

```
FILE: freebsd-11.0/sys/cam/scsi_sa.c
 204. #define cam_perigh_lock(...)   mtx_lock(...)
 205. #define cam_perigh_unlock(...)   mtx_unlock(...)
------------------------------------------------
1498. static int saioctl(...) {
          ......
1680.     cam_periph_lock(...)  /* acquire spinlock */
          ......
1683.     error = saxtget(...);
          ......
1704.     cam_periph_unlock(...)  /*release spinlock*/
          ......
2114. }
------------------------------------------------
4377. static int saextget(...) {
          ......
4444.     tmpstr2 = malloc(ts_len, M_SCSISA,
4445.                          M_WAITOK);
          // PATCH: M_WAITOK ⇒ M_NOWAIT
          ......
4548.     return error;
4549. }
```

(b) FreeBSD *scsi_sa* driver

```
FILE: netbsd-7.1/sys/dev/pci/if_vte.c
 948. /* Interrupt handler */
 949. static int vte_intr(...) {
          ......
 971.     vte_rxeof(...);
          ......
 989. }
------------------------------------------------
1045. static int vte_newbuff(...) {
          ......
1056.     if (bus_dmamap_load_mbuf(sc->vte_dmatag,
1057.             sc->vte_cdata.vte_rx_sparemap, m, 0));
          // PATCH: 0 ⇒ BUS_DMA_NOWAIT
          ......
1083.     return 0;
1084. }
------------------------------------------------
1086. static void vte_rxeof(...) {
          ......
1118.     vte_newbuff(...);
          ......
1176. }
```

(c) NetBSD *if_vte* driver

Figure 6: Examples of the real bugs detected by DSAC.

were not directly applied as the maintainers wanted to fix the bugs in other ways (such as P3 and P4). There has been no reply yet for 11 patches. There are still 238 real bugs for which DSAC cannot recommend patches, as DSAC identifies that these bugs cannot be fixed using P1 or P2. We find that most of these bugs can be fixed using P3 or P4. But those patterns require more difficult changes, and DSAC is not currently able to automatically apply them. In general, the results indicate that DSAC can generate a number of correct patches to reduce the manual work of bug fixing.

7) Bug detection and patch generation are efficient, requiring less than 10 minutes. The reasons include that intermediate results are used to reduce repeated analysis and our hybrid flow analysis is efficient.

Reviewing the results, we find two interesting things. Firstly, most of the detected bugs involve multiple functions. Indeed, driver developers may easily forget that the code is in atomic context across multiple function calls. Secondly, many of the detected bugs are related to resource allocation and release, because many extracted sleep-able functions relate to this issue.

Figure 6(a) shows a real bug detected by DSAC in the *gma500* driver of Linux 4.11.1, which has been confirmed by the developer. The function *psbfb_2d_submit* first calls *spin_lock_irqsave* to acquire a spinlock (line 115), and then it calls *psb_2d_wait_available* (line 119) that calls *psb_spank* in definition (line 91). The function *psb_spank* calls *msleep* (line 58) that can sleep at runtime, but a spinlock is held at that time, thus a SAC bug exists in the code. To help fix the bug, our pattern-based method recommends a patch that replaces *msleep* with *mdelay* (P1), and this patch has been applied by the kernel maintainer. Part of the report generated by DSAC for this bug is listed above Table 3.

5.4 Generality and Portability

We use DSAC to check file systems and network modules in Linux 4.11.1. Then we run DSAC in FreeBSD 11.0 and NetBSD 7.1 to check their kernel code. Table 3 shows the results. We have the following findings:

```
********** BUG **********
Sleep-able function: msleep
[FUNC] psb_spank (drivers/gpu/.../accel_2d.c: LINE 58)
[FUNC] psb_2d_wait_available (drivers/gpu/.../accel_2d.c: LINE 91)
......
[FUNC] psbfb_2d_submit (drivers/gpu/.../accel_2d.c: LINE 119)
......
[FUNC] psbfb_2d_submit (drivers/gpu/.../accel_2d.c: LINE 115)
```

Table 3: Results of Linux *fs* and *net*, FreeBSD and NetBSD.

Description		fs & net	FreeBSD	NetBSD
Code handling	Handled bytecode files	925	632	710
	Source files	2506	1615	1977
	Source code lines	2013K	1759K	1896K
Function extraction	Recorded functions	1927	582	304
	Sleep-able kernel interfaces	34	12	10
Bugs & patches	Filtered bugs	682081	508	2414
	Final detected bugs	42	39	7
	Real bugs	39	35 (26)	7 (7)
	Generated patches	5	10	3
	Pure time usages	32m45s	49m12s	43m38s

1) DSAC works normally when checking Linux file systems and network modules and other OS kernels. DSAC can handle their source code in a modest amount of time. And it can extract real sleep-able kernel interfaces from the code and filter out many repeated reports and false bugs.

2) DSAC in total finds 81 real bugs out of the 88 detected bugs. The false positive rate is thus 8.0%. As of January 2018, 63 of these bugs have been confirmed by kernel developers. Figure 6(b) and (c) present two real SAC bugs found by DSAC in the FreeBSD *scsi_sa* and NetBSD *if_vte* drivers. These bugs involve respectively a spinlock and an interrupt handler.

3) DSAC in total generates 18 recommended patches to help fix 59 real bugs. We manually add appropriate descriptions and submit them to kernel maintainers. As of January 2018, 13 of the patches have been applied.

Reviewing the results, we find two interesting things. Firstly, compared to the Linux kernel, fewer SAC bugs are detected in FreeBSD and NetBSD. The main reason is that in FreeBSD and NetBSD, many kernel interfaces that can sleep are carefully designed to avoid SAC bugs. For example, the FreeBSD *msleep* function takes the held spinlock as an argument and unlocks the spinlock before actually sleeping and then locks it again. Secondly, in FreeBSD and NetBSD, most of the detected bugs are in drivers, as shown in the parentheses on "Real bugs" line of Table 3. It shows that drivers remain a significant cause of system failures [39].

5.5 Sensitivity Analysis

DSAC performs flow-insensitive analysis to reduce time usage in specific cases when doing so is expected to not affect accuracy, and also maintains a lock stack to accurately identify the code in atomic context. To show the value of these two techniques, we modify DSAC to remove each of them, and evaluate each modified tool on a typical SCSI driver *fnic (drivers/scsi/fnic/)* of Linux 4.11.1. Original DSAC checks the driver in three seconds, and finds two real confirmed SAC bugs.

Flow-insensitive analysis. We use a full flow-sensitive analysis rather than the hybrid flow analysis. It finds the two SAC bugs too, but it spends two minutes, which is much longer than original DSAC.

Lock stack. We only keep a single bit indicating whether a lock is held rather than the lock stack during analysis. It also spends three seconds, but does not find any bugs. Indeed, the two bugs exist when two spinlocks are held and just one spinlock has been released, thus keeping a single bit cannot identify this atomic context.

5.6 Summary of Results

Our experiments show three significant results of using DSAC on the Linux, FreeBSD and NetBSD kernels:

- 401 new real bugs are found, of which 272 have been confirmed by kernel developers.
- Only 27 reports are false positives. Thus the overall false positive rate of bug detection is only 6.3%.
- 61 recommended patches are generated, of which 43 have been applied by kernel maintainers.

6. Comparison to Previous Approaches

Several previous approaches [2, 9, 16, 34] have considered SAC bugs. Among them, we select the *BlockLock* checker [34] to make a detailed comparison. We select this approach because: (1) It is a state-of-the-art tool to detect SAC bugs in the Linux kernel. (2) It is open-source and its bug reports are available [54]. In design, DSAC has some key improvements over *BlockLock*:

Code analysis. *BlockLock* only uses one bit of context information to check if a lock is held, so it may not accurately identify the code in atomic context when multiple locks are taken but only some of them are released. DSAC maintains a complete lock stack and performs context-sensitive analysis, thus it can accurately detect all code in atomic context. *BlockLock* is also not sensitive to the module *Makefile*, and thus may choose the wrong definition when unfolding a function call if the called function has multiple definitions. DSAC uses the module *Makefile* to accurately identify the definition of each function. And DSAC can detect SAC bugs in interrupt handlers and involving sleeping operations other than a call to an allocation function with *GFP_KERNEL*, which are not considered by *BlockLock*.

Sleep-able function extraction. *BlockLock* regards all functions called in the kernel as candidate functions to extract sleep-able functions. This strategy entails checking each function in the kernel inter-procedurally, so it may require much time. DSAC only treats the kernel interfaces possibly called in atomic context of the analyzed kernel module(s) as candidate functions, and skips the other functions not called in atomic context. DSAC also considers comments and specific macros that are neglected by *BlockLock*.

False bug filtering. *BlockLock* does not consider variable value information to validate path conditions, which may cause a number of false positives. DSAC checks the detailed code path of each possible bug, and filters out false bugs using useful and common semantic information for variables in atomic context.

Patch generation. *BlockLock* only reports bugs, but it does not help fix the bugs. DSAC uses common fixing patterns to generate recommended patches to help fix the bugs. The produced code paths of the bugs are also useful to help the user write log messages in the patches.

We also compare the results of *BlockLock* and DSAC, with two steps. Firstly, we download the bug reports of *BlockLock* on Linux 2.6.33 drivers, and get 49 reported bugs. We select the bugs related to the *x86* architecture based on driver *Kconfig* files. We get 31 reported SAC bugs (25 real bugs and 6 false bugs). Secondly, we use DSAC to check the Linux 2.6.33 driver source code. We use the kernel configuration *allyesconfig* to enable all drivers for the *x86* architecture. DSAC reports 42 sleep-able kernel interfaces and 228 reported SAC bugs. We manually check the bugs and find that 208 are real.

By manually comparing the bug reports shows: (1) 53 real bugs reported by DSAC are equivalent to 23 real bugs reported by *BlockLock*. DSAC reports more bugs because it detects sleep-able kernel interfaces, while *BlockLock* detects sleep-able functions. Thus, if a function defined in the kernel module calls several sleep-able kernel interfaces in atomic context, DSAC reports all these kernel interfaces, while *BlockLock* only reports this function. The two remaining real bugs reported by *BlockLock* are missed by DSAC, as Clang-3.2 cannot successfully compile the related driver source code. (2) DSAC filters out all false bugs reported by *BlockLock*. (3) DSAC reports 155 real bugs missed by *BlockLock*. Most of these bugs involve multiple source files, and *BlockLock* cannot handle them very precisely. And 18 bugs are related to interrupt handling, which is not considered by *BlockLock*. (4) The false positive rate of DSAC is 8.8%, which is lower than that of *BlockLock*.

However, compared to *BlockLock*, an important limitation of DSAC is that its results are specific to a single kernel configuration. *BlockLock* is based on Coccinelle [33], which does not compile the kernel module source code. Thus it can conveniently check all source files in a directory, without any kernel configuration. DSAC is based on LLVM, which compiles the kernel module source code with a selected kernel configuration. Thus, the 18 bugs found by *BlockLock* for *non-x86* architectures are all missed by DSAC, because the drivers containing these bugs are not for the *x86* architecture.

7. Limitations and Future Work

DSAC still has some limitations. Firstly, DSAC analyzes LLVM bytecode in which macros are expanded, thus the user needs to configure DSAC in terms of expanded versions of the functions and constants that are defined by macros. We plan to introduce source code information to address this issue. Secondly, DSAC cannot handle function pointers. We plan to use alias analysis [22, 47] to analyze them. Thirdly, as is typical for static analysis, the path-check method cannot filter out all invalid code paths produced by the hybrid flow analysis, which can introduce false positives. We plan to improve our path-check method by checking path conditions more accurately. Finally, the bug-fixing patterns P3 and P4 need to be supported, e.g., we plan to add the analysis for other kinds of locks to support P4.

8. Related Work

8.1 Detecting Concurrency Bugs

Many approaches [7, 14, 19, 28, 31, 32, 38, 42, 43] have been proposed to detect concurrency bugs in user-mode applications. Some of them [7, 19, 31, 42] use dynamic analysis to collect and analyze runtime information to detect concurrency bugs. But the code coverage of dynamic analysis is limited by the available test cases. Others [14, 32, 38, 43] use static analysis to cover more code without running the tested programs. But static analysis often introduces false positives, as it lacks the precise context information of real execution for tested programs. Some approaches [8, 26, 28] combine static and dynamic analysis to achieve higher code coverage with fewer false positives. Even though DSAC uses static analysis, it also exploits complementary information such as semantic information for variables to check code paths to filter out false positives.

To improve OS reliability, some approaches [13, 15, 17, 18, 40, 41, 44] detect some kinds of concurrency bugs like data races, but they do not detect SAC bugs. Several approaches [2, 9, 16, 34, 53] can detect common kinds of OS kernel bugs, including SAC bugs. But they do not specifically target SAC bugs, thus they may miss many real bugs or report many of false positives. For example, *BlockLock* [34] has an overall false positive rate of 20%, while DSAC has a lower one of 6.3%, and it also misses some real bugs found by DSAC.

8.2 Checking API Rules

Checking API rules is a promising way of finding deep and semantic bugs in the OS kernel. Some approaches [3, 5, 30] exploit specified and known API rules to statically or dynamically detect API misuses. For example, using known paired reference count management functions, RID [30] uses a summary-based inter-procedural analysis to detect reference counting bugs. To find implicit API rules, some approaches [4, 23, 24, 27, 37, 45] additionally perform specification mining by analyzing source code [24, 27, 37] or execution traces [4, 23, 45], and then use the mined API rules to detect violations.

Most of these approaches focus on the temporal rules of common API usages, such as resource acquiring and releasing pairs [37, 45] and error handling patterns [4, 24], but these approaches have not targeted SAC bugs.

8.3 Improving Kernel Module Architecture

To prevent concurrency bugs, several improved kernel module architectures have been proposed, typically for device drivers. The active driver architecture [1, 36] runs each driver in a separate thread, and thus the driver code cannot be concurrently invoked by multiple external threads through function calls. In this way, the driver works serially and does not need to use locks, thus many common concurrency bugs will never occur. The user-mode device-driver architecture [20, 25, 35] runs each driver in a separate user-mode process. This architecture protects the OS kernel against crashes or hangs caused by driver code.

These approaches still have two main limitations in practical use. Firstly, the driver code needs to be manually rewritten. Secondly, serial execution often decreases the driver performance.

9. Conclusion

In this paper, we have proposed a practical static approach named DSAC,[1] to accurately and efficiently detect SAC bugs and automatically recommend patches to help fix them. DSAC uses four key techniques: (1) a hybrid flow analysis to perform accurate and efficient code analysis; (2) a heuristics-based method to accurately extract sleep-able kernel interfaces (3) a path-check method to effectively filter out repeated reports and false bugs; (4) a pattern-based method to automatically generate recommended patches to help fix the bugs. We have used DSAC to check the kernel code of Linux, FreeBSD and NetBSD, and in total find 401 new real bugs (272 of them have been confirmed). 43 of the patches generated by DSAC have been applied.

[1] DSAC website: *http://oslab.cs.tsinghua.edu.cn/DSAC/index.html*

References

[1] S. Amani, P. Chubb, A. F. Donaldson, A. Legg, K. C. Ong, L. Ryzhyk, and Y. Zhu. Automatic verification of active device drivers. In *ACM SIGOPS Operating System Review*, volume 48, pages 106-118, 2014.

[2] Z. Anderson, E. Brewer, J. Condit, R. Ennals, D. Gay, M. Harren, G. C. Necula, and F. Zhou. Beyond bug-finding: sound program analysis for Linux. In *Proceedings of the 11th International Workshop on Hot Topics in Operating Systems (HotOS)*, pages 1-6, 2007.

[3] J. J. Bai, H. Q. Liu, Y. P. Wang, and S. M. Hu. Runtime checking for paired functions in device drivers. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC)*, pages 407-414, 2014.

[4] J. J. Bai, Y. P. Wang, H. Q. Liu, and S. M. Hu. Mining and checking paired functions in device drivers using characteristics fault injection. In *Information and Software Technology*, volume 73, pages 122-133, 2016.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st European Conference on Computer Systems (EuroSys)*, pages 73-85, 2006.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 209-224, 2008.

[7] Y. Cai, J. Zhang, L. Cao, and J. Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 24th International Symposium on Foundations of Software Engineering (FSE)*, pages 810-821, 2016.

[8] L. Chew, and D. Lie. Kivati: fast detection and prevention of atomic violations. In *Proceedings of 5th European Conference on Computer Systems (EuroSys)*, pages 307-320, 2010.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th International Symposium on Operating Systems Principles (SOSP)*, pages 73-88, 2001.

[10] Jonathan Corbet. Atomic context and kernel API design. In *Linux Weekly News (LWN.net)*, 2008. *https://lwn.net/Articles/274695/.*

[11] J. Corbet, A. Rubini, and G. K. Hartman. Spinlocks and atomic context. In *Linux Device Drivers*, 3rd edition, pages 118-119, 2005.

[12] D. Cotroneo, R. Natella, and S. Russo. Assessment and improvement of hang detection in the Linux operating system. In *Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS)*, pages 288-294, 2009.

[13] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 166-177, 2015.

[14] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*, pages 480-491, 2009.

[15] D. Engler, and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, pages 237-252, 2003.

[16] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 1-16, 2000.

[17] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 151-162, 2010.

[18] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 415-431, 2014.

[19] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, pages 215-228, 2011.

[20] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168-178, 2008.

[21] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 55-64, 2010.

[22] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)*, pages 249-259, 2008.

[23] C. LaRosa, L. Xiong, and K. Mandelberg. Frequent pattern mining for kernel trace data. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC)*, pages 880-885, 2008.

[24] J. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. 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)*, pages 43-53, 2009.

[25] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. T. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: achieved performance. In *Journal of Computer Science and Technology (JCST)*, volume 20, issue 5, pages 654-664, 2005.

[26] Q. Li, Y. Jiang, T. Gu, C. Xu, J. Ma, X. Ma, and J. Lu. Effectively manifesting concurrency bugs in Android apps. In *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 209-216, 2016.

[27] Z. Li, and Y. Zhou. 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)*, pages 306-315, 2005.

[28] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian. DCatch: automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 677-691, 2017.

[29] R. Love. Interrupt context. In *Linux Kernel Development*, 3rd edition, page 122, 2010.

[30] J. Mao, Y. Chen, Q. Xiao, and Y. Shi. RID: finding reference count bugs with inconsistent path pair checking. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 531-544, 2016.

[31] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, pages 134-143, 2009.

[32] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the 27th International Conference on Programming Language Design and Implementation (PLDI)*, pages 308-319, 2006.

[33] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, pages 247-260, 2008.

[34] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall. Faults in Linux 2.6. In *ACM Transactions on Computer Systems (TOCS)*, volume 32, issue 2, pages 4:1-4:40, 2014.

[35] M. J. Renzelmann, and M. M. Swift. Decaf: moving device drivers to a modern language. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX ATC)*, pages 1-14, 2009.

[36] L. Ryzhyk, Y. Zhu, and G. Heiser. The case for active device drivers. In *Proceedings of the 1st Aisa-Pacific Workshop on Systems (APSys)*, pages 25-30, 2010.

[37] S. Saha, J. P. Lozi, G. Thomas, J. Lawall, and G. Muller. 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)*, pages 1-12, 2013.

[38] A. Santhiar, and A. Kanade. Static deadlock detection for asynchronous C# programs. In *Proceedings of the 38th International Conference on Programming Language Design and Implementation (PLDI)*, pages 292-305, 2017.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, pages 207-222, 2003.

[40] L. Tan, Y. Zhou, and Y. Padioleau. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 11-20, 2011.

[41] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 391-402, 2016.

[42] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, pages 253-264, 2010.

[43] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, pages 602-629, 2005.

[44] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*, pages 501-504. 2007.

[45] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of 28th International Conference on Software Engineering (ICSE)*, pages 282-291, 2006.

[46] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th International Symposium on Foundations of Software Engineering (FSE)*, pages 26-36, 2011.

[47] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th International Symposium on Code Generation and Optimization (CGO)*, pages 218-229, 2010.

[48] Y. Zhu, Y. Li, J. Xue, T. Tan, J. Shi, Y. Shen, and C. Ma. What is system hang and how to handle it. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 141-150, 2012.

[49] Clang compiler. *http://clang.llvm.org/.*

[50] Linux kernel document for memory allocation. *https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html.*

[51] LLVM Compiler Infrastructure. *https://llvm.org/.*

[52] MYSQL database. *https://www.mysql.com/.*

[53] Syzkaller tool. *https://github.com/google/syzkaller/.*

[54] Website for “Faults in Linux: Ten years later”. *http://faultlinux.lip6.fr/.*