DSAC: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules
Jia-Ju Bai, Yu-Ping Wang, Julia Lawall, Shi-Min Hu

To cite this version:

HAL Id: hal-01853268
https://inria.hal.science/hal-01853268
Submitted on 3 Aug 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
DSAC: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules

Jia-Ju Bai and Yu-Ping Wang, Tsinghua University;
Julia Lawall, Sorbonne Université/Inria/LIP6; Shi-Min Hu, Tsinghua University

https://www.usenix.org/conference/atc18/presentation/bai
DSAC: Effective Static Analysis of Sleep-in-Atomic-Context Bugs in Kernel Modules

Jia-Ju Bai\textsuperscript{1}, Yu-Ping Wang\textsuperscript{1}, Julia Lawall\textsuperscript{2}, Shi-Min Hu\textsuperscript{1}
\textsuperscript{1}Tsinghua University, \textsuperscript{2}Sorbonne Université/Inria/LIP6

Abstract

In a modern OS, kernel modules often use spinlocks and interrupt handlers to monopolize a CPU core to execute 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. 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 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 forbidden, as it can block a CPU core for a long period 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, 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 report 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\textsuperscript{1} 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

\textsuperscript{1}DSAC website: http://oslab.cs.tsinghua.edu.cn/DSAC/index.html
sleepable 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 across function calls.

2. **Precise and detailed bug reports.** To achieve precise bug detection, DSAC uses a heuristics-based method to extract sleepable 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.** Once the user offers the names of spin-lock and -unlock functions, interrupt-handler-register functions and basic sleepable kernel interfaces, the remaining phases of DSAC are fully automated. DSAC can effectively check 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 sleepable kernel interfaces in the analyzed kernel modules; (3) a path-check method to effectively filter out repeated reports and false bugs; and (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 should 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 presents the background. 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. Background

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 that 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. This context can protect resources from concurrent access, in which the code execution should complete as quickly as possible without able to be rescheduled. 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].
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.

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 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 flow-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 a good understanding of the kernel code. Specifically, for a function defined in the kernel module (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 the 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 infeasible 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. Besides, incorrect patches can introduce new bugs [21]. 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

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 explicitly.

Our hybrid flow analysis has two steps. The first step identifies 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.

The second step performs the main analysis. Figure 2 presents the procedure FlowAnalysis. It maintains two stacks, namely a path stack (path_stack) to store the executed code path and 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:

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, and 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. Note that this prevents infinite looping on recursive calls. Thirdly, HanCall adds mycall into path_stack, and gets the called function myfunc (lines 7-8). Fourthly, HowToFunc (line 9) performs the analyses presented 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, it returns to avoid repeated analysis. 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, `HanBlock` pushes the call onto or pops an item from `lock_stack`; otherwise, the call is handled by `HanCall`.

Fourthly, `HanBlock` checks if `lock_stack` is empty and `g_intr_flag` is `FALSE`. If so, it returns (lines 14-16); otherwise, each successive basic block of `myblock` is handled using `HanBlock` (lines 17-19).

**Flow Analysis.** It performs the main analysis, in two steps. Firstly, each target function is analyzed (lines 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 inter-procedural analysis of `MyFunc`. During the analysis, no spinlock is held after the first line of `FuncB` (line 24), thus the following 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.

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.

After the extraction, we get the sleep-able kernel interfaces that are possibly called in atomic context of the analyzed kernel module(s). These kernel interfaces can be used to detect SAC bugs in the kernel module(s).
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 involve simple modifications, while P3 and P4 involve 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 static approach DSAC, to effectively detect SAC bugs in kernel modules and 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:

- **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 the 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 reuses 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 and generates reports for the final detected SAC bugs.

• **Patch generator.** With the bug code paths and kernel module source code, this part uses our pattern-based method to automatically recommend 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 on only 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 the intermediate results, including the function name, constant arguments, 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. 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 and false bugs. Finally, DSAC produces detailed reports for the found bugs (including code paths and source file names), so the user can 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 (such as `spin_lock_irq` and `spin_unlock_irq` for the Linux kernel), interrupt-handler-register functions (such as `request_irq` for the Linux kernel), and basic sleep-able kernel interfaces (such as `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.
5.2 Extracting Sleepable Kernel Interfaces

We first extract the sleepable 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 sleepable kernel interfaces that are called in atomic context of the analyzed drivers. In Linux 3.17.2 and 4.11.1, 70 and 94 sleepable 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 sleepable, 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 sleepable 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 sleepable kernel interfaces.

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

Table 1: Results of extracting sleepable kernel interfaces.

5.3 Detecting Bugs and Generating Patches

Based on the above extracted sleepable kernel interfaces, we use DSAC to perform bug detection and recommend patches. Firstly, to validate whether DSAC can find known bugs, we use DSAC to check the drivers in Linux 3.17.2. We do not generate patches in this case, because this kernel version is very old. Secondly, to validate whether DSAC can find new bugs and recommend patches to help fix them, we use DSAC to check the drivers in Linux 4.11.1. We count the bugs according to the pair of entry and terminal basic blocks. To check results’ accuracy, we manually check all detected bugs to identify whether they are real bugs. Table 2 shows the results. We have the following findings:

1) Our path-check filtering method is effective in automatically filtering out 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 sleepable 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, noted in

<table>
<thead>
<tr>
<th>Description</th>
<th>3.17.2</th>
<th>4.11.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detected bugs</td>
<td>Repeated filtered: 479630</td>
<td>629924</td>
</tr>
<tr>
<td>False filtered: 282</td>
<td>430</td>
<td></td>
</tr>
<tr>
<td>Final detected: 215</td>
<td>340</td>
<td></td>
</tr>
<tr>
<td>Interrupt handling: 7</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Real: 200</td>
<td>320</td>
<td></td>
</tr>
<tr>
<td>Patch generation</td>
<td>P1 (replace the function): -</td>
<td>28 (18)</td>
</tr>
<tr>
<td>P2 (replace the flag): -</td>
<td>15 (12)</td>
<td></td>
</tr>
<tr>
<td>Total: -</td>
<td>43 (30)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Detected bugs and generated patches in drivers.
parentheses in Table 2. 2 patches 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 they do not match P1 or P2. 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 sleepable functions relate to this issue.

We also classify the 320 real bugs found by DSAC in Linux 4.11.1 drivers, according to driver class. Table 3 shows the top results. We find that SCSI and network drivers share 58% of all bugs.

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 psbf_submit first calls spin_lock_irqsave to acquire a spinlock (line 115), and then it calls psbf_2d_wait_available (line 119) that calls psb_spank in definition (line 91). The function psb_spank calls msleep (line 58) that can sleep. 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 DSAC’s report 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 4 shows the results. We have the following findings:

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. It can extract real sleepable kernel interfaces 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%. The false positives are again due to searching invalid code paths. 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 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 

\texttt{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 4. 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 \texttt{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 \texttt{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 \texttt{BlockLock}:

**Code analysis.** \texttt{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. \texttt{BlockLock} is also not sensitive to the module \texttt{Makefile}, and thus may choose the wrong definition when unfolding a function call if the called function has multiple definitions. DSAC uses the module \texttt{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 \texttt{GFP_KERNEL}, which are not considered by \texttt{BlockLock}.

**False bug filtering.** \texttt{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.** \texttt{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 \texttt{BlockLock} and DSAC, with two steps. Firstly, we download the bug reports of \texttt{BlockLock} on Linux 2.6.33 drivers, and get 49 reported bugs. We select the bugs related to the x86 architecture based on driver \texttt{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 \texttt{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 \texttt{BlockLock}. DSAC reports more bugs because it detects sleep-able kernel interfaces, while \texttt{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 \texttt{BlockLock} only reports this function. The two remaining real bugs reported by \texttt{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 source code. Thus it can conveniently check all source files without any kernel configuration. DSAC is based on LLVM, which compiles the source code with a selected kernel configuration. Thus, the 18 bugs found by BlockLock for non-x86 architectures are missed by DSAC.

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, 46] 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, 32, 38, 42, 43] have been proposed to detect concurrency bugs in user-mode applications. Some of them [7, 19, 42] use dynamic analysis to collect and analyze runtime information to detect concurrency bugs. But the code coverage of dynamic analysis is limited by 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. 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, 31] use specified and known API rules to statically or dynamically detect API misuses. For example, with 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, 47] do specification mining by analyzing source code [24, 27, 37, 47] 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, which can serialize access to the driver and eliminate the possibility of concurrency bugs. 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 caused by driver code.

These approaches have a main limitation, namely that the driver code must be manually rewritten.

9. Conclusion
In this paper, we have proposed DSAC, a static approach, to effectively detect SAC bugs and automatically recommend patches to help fix them. DSAC uses four key techniques: (1) a hybrid flow analysis to identify the code in atomic context; (2) a heuristics-based method to extract sleep-able kernel interfaces; (3) a path-check method to 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 find 401 new real bugs. As of January 2018, 272 of them have been confirmed, and 43 of the patches generated by DSAC have been applied.
References


