DCNS: Automated Detection of Conservative Non-Sleep Defects in the Linux Kernel

Jia-Ju Bai
Tsinghua University
baijiaju1990@gmail.com

Julia Lawall
Sorbonne University/Inria/LIP6
Julia.Lawall@lip6.fr

Wende Tan, Shi-Min Hu
Tsinghua University
twd2@163.com
shimin@tsinghua.edu.cn

Abstract

For waiting, the Linux kernel offers both sleep-able and non-sleep operations. However, only non-sleep operations can be used in atomic context. Detecting the possibility of execution in atomic context requires a complete inter-procedural flow analysis, often involving function pointers. Developers may thus conservatively use non-sleep operations even outside of atomic context, which may damage system performance, as such operations unproductively monopolize the CPU. Until now, no systematic approach has been proposed to detect such conservative non-sleep (CNS) defects.

In this paper, we propose a practical static approach, named DCNS, to automatically detect conservative non-sleep defects in the Linux kernel. DCNS uses a summary-based analysis to effectively identify the code in atomic context and a novel file-connection-based alias analysis to correctly identify the set of functions referenced by a function pointer. We evaluate DCNS on Linux 4.16, and in total find 1629 defects. We manually check 943 defects whose call paths are not so difficult to follow, and find that 890 are real. We have randomly selected 300 of the real defects and sent them to kernel developers, and 251 have been confirmed.

CCS Concepts • **Software and its engineering** → **Software defect analysis**; *Operating systems*; • **Computer systems organization** → Reliability;

Keywords Linux kernel; atomic context; defect detection; function-pointer analysis

ACM Reference Format:
Jia-Ju Bai, Julia Lawall, and Wende Tan, Shi-Min Hu. 2019. DCNS: Automated Detection of Conservative Non-Sleep Defects in the Linux Kernel. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3297858.3304065

1 Introduction

The Linux kernel provides fundamental support, such as memory management and hardware access, for high-level applications. To limit the overhead on applications, kernel code should complete as quickly as possible. Nevertheless, in many cases, the Linux kernel must wait, *e.g.* until a resource is available or a response from a hardware device is received. For efficiency, the Linux kernel offers a range of waiting operations, tailored to the requirements of different execution contexts.

Waiting operations in the Linux kernel can be categorized as *non-sleep* and *sleep-able*. Non-sleep operations, such as the function `mdelay`, spin by monopolizing a CPU core until a specified amount of time has passed. These operations have quick reactivity when the waiting time expires, as the thread is already running on the CPU, but spinning can reduce the overall system throughput, because the monopolized CPU is not available to other more productive threads. In contrast, sleep-able operations, such as the function `msleep`, can make the thread yield the CPU to sleep (or block), until a timer expires or some other expected condition becomes true. These operations have slower reactivity, because a waking thread must wait further until the scheduler allows it to run on the CPU, but system throughput can be improved, because other threads can run on the CPU during the wait. A second non-sleep alternative is to abort the operation and return an error code, which is allowed by the non-sleep flag `GFP_ATOMIC` in the Linux kernel. This strategy eliminates the overhead incurred by sleeping, but increases the probability of failure of the operation. For example, resource allocation using `GFP_ATOMIC` is more likely to fail than resource allocation using the sleep-able flag `GFP_KERNEL`. It also imposes on the calling context the burden of handling the error case.

Given this range of waiting operations, the challenge for the kernel developer is to choose the best one for each execution context. A general "rule of thumb" is that spinning is fine for short durations, due to the improved reactivity, but sleeping should be used for longer waiting times, to avoid wasting CPU resources.[1] In the Linux kernel, however, in *atomic context* [9], where a spinlock is held or within an interrupt handler, sleeping is not allowed, and non-sleep operations must be used. Unlike the rule about the expected waiting time, the property of being in atomic context is not a local property, *i.e.*, it does not depend on the arguments of the call but rather on the possible code execution paths through the system that can reach the operation. Understanding the complete set of such possible execution paths can require tracing through many function calls across many different source files, which is challenging given the large size of the Linux kernel. The challenge is compounded when the execution path contains function pointers, which the Linux kernel uses often for modularity. Kernel developers who are unsure about whether their code is in atomic context may simply use non-sleep operations, to the detriment of the overall system performance. We call such unnecessary non-sleep operations *conservative non-sleep (CNS) defects*. There is a need for tool support to detect such issues.

In this paper, we propose a practical static analysis approach named DCNS, to automatically detect conservative non-sleep defects in the Linux kernel. Overall, DCNS first identifies non-sleep function calls in the Linux kernel code, then marks those occurring in atomic context, and finally reports the remaining ones as CNS defects. Specifically, DCNS consists of three phases. Firstly, DCNS analyzes the Linux kernel code to identify non-sleep function calls, and collects some useful code information, such as function definitions, function-pointer assignments, etc. Secondly, using the collected information, DCNS identifies source files that are connected by direct function calls. We call this connection a *file connection*, and it guides the function-pointer analysis in the next phase. Thirdly, DCNS performs a summary-based analysis to mark all non-sleep function calls that occur in atomic context. To cover all possible code execution paths in atomic context, this analysis is inter-procedural, and is started from each spinlock-acquire function call and the entry of each interrupt handler function. It maintains a lock stack to accurately identify atomic context, and uses function summaries to avoid repeated analysis. To handle function-pointer calls, DCNS performs a file-connection-based alias analysis, to correctly identify the set of functions referenced by the function pointer. After this phase, DCNS reports unmarked non-sleep function calls as CNS defects.

We have implemented DCNS using LLVM [24]. DCNS works automatically, given the Linux kernel source code. Moreover, DCNS can work in parallel to speed up the analysis. Overall, we make the following contributions:

[1]Indeed, spinlocks are designed according to this principle, where they spin for a short time in case the lock becomes available quickly, and then sleep if it is necessary to wait for a longer time.

- Using an experiment with a standard benchmark, we show that CNS defects can damage system performance. We also study the commits fixing CNS defects in the Linux kernel, and find that these CNS defects have all been manually detected, because no systematic approach has been proposed to detect such defects.
- We analyze the main challenges of detecting CNS defects in the Linux kernel, and then propose a practical static approach named DCNS to automatically detect CNS defects in the Linux kernel code. To our knowledge, DCNS is the first approach to automatically detect CNS defects.
- In DCNS, we propose a file-connection-based alias analysis to correctly identify the set of functions referenced by a function pointer.
- We evaluate DCNS on Linux 4.16, and in total find 1629 defects. We manually check 943 defects whose call paths are not so difficult to follow, and find that 890 are real, giving an accuracy rate of 94%. We have randomly selected 300 of the real defects and sent them to kernel developers, and 251 have been confirmed.

The remainder of this paper is organized as follows. Section 2 reviews the motivation. Section 3 shows the main challenges of detecting CNS defects. Section 4 introduces our key techniques. Section 5 introduces DCNS in detail. Section 6 presents our evaluation. Section 7 discusses the results. Section 8 gives related work, and Section 9 concludes.

2 Motivation

We first introduce sleeping and atomic context, then introduce conservative non-sleep (CNS) defects, and finally study the existing commits fixing CNS defects in the Linux kernel.

2.1 Sleeping and Atomic Context

Sleeping. To improve system performance and utilize multiple CPUs, the Linux kernel allows multiple threads to run concurrently. When there are more threads than CPUs, it is undesirable for a kernel thread to monopolize a CPU. Thus, when a kernel thread has nothing useful to do, it should yield the CPU for other threads to use. We refer to the action of yielding the CPU as *sleeping*. An operation that can sleep is a *sleep-able* operation.

Atomic context. Sleeping is not allowed in a special kernel context known as *atomic context* [9]. Typical examples of atomic context are holding a spinlock and executing an interrupt handler. Atomic context is used to protect data from concurrent access. In such a context, a thread monopolizes a CPU, and the progress of other threads that need to concurrently access the same data is delayed. Code in atomic context should thus complete as quickly as possible. Code in atomic context is furthermore not able to be rescheduled. As sleeping can block the CPU for a long time and requires the code to be rescheduled, sleeping in atomic context is forbidden. It may cause a system hang or crash at runtime.

2.2 Conservative Non-Sleep Defects

While waiting in atomic context must be implemented with non-sleep operations, the use of non-sleep operations outside of atomic context may damage system performance by wasting CPU resources and increasing the likelihood of failure of some operations. Thus, such use represents a form of defect, which we refer to as a *conservative non-sleep (CNS) defect*. Still, whether code is in atomic context often depends on inter-procedural properties, which are tricky to follow, especially in code with the size and complexity of the Linux kernel. Kernel developers may be unsure about whether their code may occur in atomic context, and may use non-sleep operations to be on the safe side. We now give an example of Linux kernel code containing CNS defects, and study the impact of these defects on performance.

CNS defect example. Figure 1 shows part of a patch (commit `4e0d8f7d97f9`) to the *e1000* Ethernet controller driver in Linux 3.2 for fixing CNS defects. This patch was applied in 2011, at which point the CNS defects had been present since Linux 2.6.10, released nearly 7 years earlier. In Figure 1, the function `e1000_polarity_reversal_workaround` can be called when the Ethernet controller checks or changes the link. This function calls `mdelay`, which is a non-sleep function, to busily wait for some time before and after writing hardware registers. But this function is never called in atomic context, so `mdelay` is unnecessary, and it will monopolize a CPU to busily wait with no benefit. The patch thus replaces the calls to `mdelay` by calls to the sleep-able function `msleep`.

CNS defect impact. To show the potential impact of CNS defects on system performance, we run the *e1000* driver in Linux 3.2 with and without the patch in Figure 1, on a Lenovo PC with four CPUs and an Intel 82543 Ethernet controller (managed by the *e1000* driver). To cause the code in Figure 1 to be executed, we can change the network link status when the driver is running. To measure the system performance, we select a standard benchmark *sysbench* [36] and run its CPU tests with N (the number of enabled CPUs in the system) running threads, and measure the execution time. We run the experiment in four situations:

- *S1*: Without applying the patch, and keeping the network link status unchanged during the test.
- *S2*: Without applying the patch, and changing the network link status once during the test.
- *S3*: Applying the patch, and keeping the network link status unchanged during the test.
- *S4*: Applying the patch, and changing the network link status once during the test.

Table 1 shows the results. By comparing the results of S1 and S2, we find that without fixing the CNS defects, when the network link status is changed, the execution time of the benchmark increases by 25%, 22%, 19% and 13%, respectively with 1-4 enabled CPUs. The impact decreases as the number of enabled CPUs increases, because more CPUs are available to run the benchmark, even when the *e1000* driver monopolizes one CPU to run the non-sleep function `mdelay`. By comparing the results of S3 and S4, we find that after fixing the defects, the performance is essentially the same, whether the line status is kept the same (S3) or changed (S4). Thus, our experiment shows that CNS defects can damage system performance, so they should be detected and fixed.

```
--- a/drivers/net/ethernet/intel/e1000/e1000_hw.c
+++ b/drivers/net/ethernet/intel/e1000/e1000_hw.c
@@ - 5716,85 +5716,85 @@
static s32 e1000_polarity_reversal_workaround(...) {
     ...
     /* Recommended delay time after link has been lost */
-    mdelay(1000);
+    msleep(1000);

     ret_val = e1000_write_phy_reg(...);
     if (ret_val)
         return ret_val;
-    mdelay(50);
+    msleep(50);
     ret_val = e1000_write_phy_reg(...);
     if (ret_val)
         return ret_val;
-    mdelay(50);
+    msleep(50);
     ret_val = e1000_write_phy_reg(...);
     if (ret_val)
         return ret_val;
-    mdelay(50);
+    msleep(50);
     ...
}
```

Figure 1. Part of a patch for fixing CNS defects in the *e1000* driver of Linux 3.2.

Table 1. Running time of the benchmark.

N (CPU)	S1	S2	S3	S4
1	16.7s	20.9s	16.7s	16.8s
2	8.3s	10.1s	8.3s	8.3s
3	5.7s	6.8s	5.7s	5.8s
4	4.5s	5.1s	4.5s	4.5s

2.3 Study on the Commits of Fixing CNS Defects

Table 2 shows the total number of commits between Linux 4.0 (released in April 2015) and Linux 4.12 (released in July 2017) that fix CNS defects by changing `mdelay` (non-sleep) to `msleep` (sleep-able) or `usleep_range` (sleep-able), and by changing `GFP_ATOMIC` (non-sleep) to `GFP_KERNEL` (sleep-able). During this period, the Linux kernel received almost 160K non-merge commits, amounting to almost 6000 per month, of which only around 1 commit per month fixes such a CNS defect. Furthermore, only one of the CNS-defect fixing commits mentions the use of a tool, *checkpatch* [5]. While checkpatch is well known to kernel developers for checking

Table 2. Commits of fixing CNS defects in Linux 4.0 to 4.12.

Time	Total	Per month	Wait	GFP Flag	Tools
2015 (Apr-Dec)	13	1.44	6	7	none
2016 (Jan-Dec)	11	0.92	4	7	none
2017 (Jan-Jun)	4	0.67	3	1	checkpatch (1 commit)

patches for coding style violations, it is not effective to find CNS defects. Indeed, checkpatch relies on regular expressions, and is thus not able to sufficiently analyze the context of a non-sleep operation to justify transforming it into a sleep-able operation. This analysis must still be performed manually.

Some fixing commits explicitly mention the side effects of the CNS defects. For example, commit `55bf851b4ad7` mentions concern about a busy wait of upwards of 100ms, which is a long time for the kernel. Commit `0c87b672098b` mentions that `GFP_ATOMIC` can cause an allocation failure that will cause an easily avoidable user-visible failure. Commit `fa17806cde76` mentions that `GFP_ATOMIC` can cause dropping packets. From these commits, we find that CNS defects can cause serious system problems besides damaging system performance.

The logs associated with these fixing commits furthermore show that CNS defects are often introduced in two ways. Firstly, due to complex control logic across multiple function calls and source files, the kernel developer may not realize that the code never occurs in atomic context (*e.g.*, commits `55bf851b4ad7` and `c01c77ce4b2`). Secondly, with the evolution of the Linux kernel, code that originally occured in atomic context may be moved out of atomic context, without changing the original non-sleep operations (*e.g.*, commits `882e1492c7ca` and `cc1674eeee60`).

Detecting CNS defects requires checking the code across multiple function calls, which is time-consuming and error-prone. To our knowledge, no systematic approach has been proposed to detect CNS defects. Thus, our goal is to propose an automated and effective approach to detect CNS defects in the Linux kernel.

3 Challenges

Our basic strategy for detecting CNS defects is to first find non-sleep operations and then identify those that may be executed in atomic context. The remaining ones are then defects. Implementing this strategy involves three main challenges:

3.1 C1: Atomic Context Analysis

There are two aspects of this challenge:

Identify the code in atomic context. Whether the code occurs in atomic context depends on whether the code can be reached by an execution path where a spinlock is held or that is part of the execution of an interrupt handler. Checking this condition is straightforward when *e.g.*, a spinlock is acquired in the same function, but it can be much more difficult when multiple function calls have to be taken into account.

Ensure completeness. To accurately detect CNS defects, *all* code that may occur in atomic context needs to be completely identified. Otherwise, non-sleep operations that are actually executed in atomic context may be not marked, and they will be reported as CNS defects, leading to false positives. If the user fixes these false CNS defects by using sleep-able operations, serious sleep-in-atomic-context bugs may be introduced, which can cause system hangs and crashes.

Several previous approaches [3, 13, 27] can detect sleep-in-atomic-context (SAC) bugs, and they perform atomic context analysis. However, they focus on how to accurately find the *existence* of a path where the code occurs in atomic context. For these approaches, missing some code that may occur in atomic context (like the code in a function referenced by a function pointer) only introduces false negatives, but for detecting CNS defects, it introduces false positives. Thus, a new atomic context analysis is needed for the latter problem.

3.2 C2: Function-Pointer Analysis

The Linux kernel is essentially an object-oriented system, in which data structures containing function pointers play the role of objects containing methods. This design improves extensibility, but greatly complicates static analysis. Namely, many possible execution paths contain calls to function pointers, and static analysis needs to correctly determine the set of functions that may be referenced by these pointers. An accurate function-pointer analysis is thus very important when detecting CNS defects. Without function-pointer analysis, non-sleep operations in the referenced functions may not be marked as being in atomic context, and thus many false positives will be introduced. On the other hand, if incorrect functions are identified for function pointers, the amount of code that is considered to be in atomic context will increase, so many false negatives will occur. Some alias analysis approaches (*e.g.* [17–19, 25]) can handle function pointers, but they are reported to produce many incorrect results for function pointers in large, complex software [17, 19].

3.3 C3: Analysis efficiency

The Linux kernel code base is very large, amounting to over 16.8M lines in our tested version Linux 4.16, and complex. Moreover, it contains many source files in multiple directories. For these reasons, performing static analysis of the Linux kernel code may be quite time-consuming.

4 Key Techniques

To solve the above challenges, we propose two key techniques. For *C1* and *C3*, we propose a summary-based analysis to effectively identify the code that may be executed in atomic context. For *C2*, we propose a file-connection-based alias analysis to correctly identify the set of functions referenced by a function pointer. We now introduce these techniques.

4.1 Summary-Based Analysis of Atomic Context

Atomic context analysis is the basis of detecting CNS defects. Our analysis extends that of DSAC, our previous approach for detecting sleep-in-atomic-context (SAC) bugs [3]. DSAC provides a hybrid flow analysis to detect code in atomic context. This analysis is inter-procedural and context-sensitive, and it maintains a lock stack and an interrupt handling flag, to respectively record spinlock and interrupt handling states across function calls. To provide accuracy, this analysis is flow-sensitive if the analyzed function is in an interrupt handler or calls spin-lock and spin-unlock functions. Otherwise, to improve efficiency, the analysis is flow-insensitive.

While the analysis provided by DSAC was sufficient to find many SAC bugs, for detecting CNS defects, it has some limitations: (1) DSAC handles a single kernel module at a time and only unfolds function calls within the module. *To detect CNS defects, we must extend this analysis to consider the whole kernel, to collect the complete calling context of each non-sleep operation.* (2) DSAC considers one execution path at a time, and only aborts an analysis if the same function or basic block and context (lock stack and an interrupt handling flag) has been seen in the current execution path. *To detect CNS defects, we introduce globally visible function summaries, to avoid repeatedly unfolding core kernel functions that may be called from many different drivers.* (3) DSAC performs flow-sensitive analysis of the functions called from interrupt handlers, to detect and abort repeated analysis within the current path. In the case of interrupt handlers, code is already in atomic context, it is unnecessary to analyze each code path. Thus, *to detect CNS defects, functions called from interrupt handlers are analyzed in a flow-insensitive way, to improve efficiency.* (4) DSAC stops at function pointers, potentially missing some execution paths and thus some SAC bugs. This is not acceptable for CNS defects, where all execution paths must be considered. *To detect CNS defects, we handle function pointers using a file-connection-based alias analysis that we will introduce in Section 4.2.*

Our analysis has two steps. The first step identifies interrupt handler functions and calls to spin-lock functions, which initiate atomic context. Starting from these entry points, the second step uses *AtomicAnalysis* shown in Figure 2, to identify the code that may be executed in atomic context and mark the non-sleep function calls within this code. Like the analysis of DSAC [3], *AtomicAnalysis* maintains a lock stack (*lock_stack*) to store the spinlock state, a path stack (*path_stack*) to store analyzed basic blocks, and a global flag (*g_intr_flag*) to indicate whether the code is executed in an interrupt handler. As shown in Figure 2, *AtomicAnalysis* uses *HandleBlock* to handle a basic block, *HandleCall* to handle a function call, and *HandleFunc* (new as compared to DSAC) to handle the definition of a called function. In the following, we introduce our summary-based analysis and how it extends the hybrid flow analysis of DSAC.

HandleBlock. It handles the instructions in the basic block *myblock with* the arguments *path_stack* and *lock_stack*. It first checks the *path_stack* to see if myblock has already been analyzed with respect to the current *lock_stack* and *g_intr_flag*, to prevent infinite loops. Then, it analyzes the definition of each called function in *myblock* except for locking functions and non-sleep functions. Finally, it handles each successive basic block. The only change as compared to DSAC is to mark calls to non-sleep functions that are detected to be in atomic context (lines 10-11).

HandleCall. It handles the function call *mycall* with the arguments *path_stack* and *lock_stack*. As compared to DSAC, the analysis is simplified to process all calls rather than checking for and stopping at module boundaries. HandleCall also uses the results of our alias analysis, described in Section 4.2, to collect all possibly referenced functions when the call involves a function pointer (lines 5-8).

HandleFunc. It handles the function *myfunc* with the arguments *path_stack* and *lock_stack*. HandleFunc implements the summary-based analysis. For a given function name, *lock_stack* and *g_intr_flag*, *HandleFunc* searches the function summaries stored in a global database to check whether *myfunc* has been already handled under the current execution context (lines 1-4). If so, it returns immediately. If not, it adds the function name and context information to the database. Finally, *HandleFunc* handles the definition of *myfunc* (lines 5-14). If *g_intr_flag* is *TRUE* or *myfunc* does not contains spin-lock or spin-unlock function calls, *HandleFunc* performs flow-insensitive analysis to handle each function call in *myfunc*, using *HandleCall*. Otherwise, *HandleFunc* performs flow-sensitive analysis from the entry basic block of *myfunc*, using *HandleBlock*, because whether the code is in atomic context is decided by the positions of the spinlock-related function calls in the path.

4.2 File-Connection-Based Function-Pointer Analysis

The main technical challenge in detecting CNS defects is to accurately identify the set of functions referenced by a given function pointer. To do this, we first collect all of the functions that may be assigned to this function pointer, and then select the relevant function(s) at a given call site from this collected set. The first step is done straightforwardly, by scanning the source code for assignments to function pointers. The second step, of how to identify the relevant function(s) at a particular call site, is more difficult.

To illustrate the issues involved, we consider the Linux kernel code shown in Figure 3. The function `ixgbe_get_ethtool_stats` calls the function `dev_get_stats` (line 1211) that is defined in another source file. The function `dev_get_stats` uses a function-pointer call through the `ndo_get_stats64` field of a `net_device_ops` data structure. This field is initialized in multiple drivers to various functions; Figure 4(b) shows three examples. Because this function

```
AtomicAnalysis: Identify the code in atomic context
InitializeFunctionSummary();
g_cur_file := GetCurrentSourceFileName();
foreach func_call in spinlock_func_set do
    myblock := GetBasicBlock(func_call);
    lock_stack := ø; path_stack := ø; g_intr_flag := FALSE;
    HandleBlock(block, path_stack, lock_stack);
end foreach
foreach func in intr_handler_func_set do
    lock_stack := ø; path_stack := ø; g_intr_flag := TRUE;
    entry_block := GetEntryBlock(func);
    HandleBlock(entry_block, path_stack, lock_stack);
end foreach
```

```
HandleBlock(myblock, path_stack, lock_stack)
if PathHasExisted(myblock, path_stack, lock_stack) == TRUE then
    return; /* prevent infinite handling on loops and recursive calls */
end if
AddPathStack(myblock, path_stack);
foreach func_call in FunctionCallList(myblock) do
    if func_call is a call to a spin-lock function then
        Push func_call onto lock_stack;
    else if func_call is a call to a spin-unlock function then
        Pop an item from lock_stack;
    else if func_call is a call to a non-sleep function then
        MarkNonSleepFunction(func_call);
    else
        HandleCall(func_call, path_stack, lock_stack);
    end if
end foreach
if lock_stack == ø and g_intr_flag == FALSE then
    return;
end if
foreach block in SuccessorBlocks(myblock) do
    HandleBlock(block, path_stack, lock_stack);
end foreach
```

```
HandleCall(mycall, path_stack, lock_stack)
if lock_stack == ø and g_intr_flag == FALSE then
    return;
end if
func_set := ø;
if mycall is a call to a function pointer then
    /* File-connection-based alias analysis to
     * get the set of referenced functions */
    func_set := FileConnAliasAnalysis(mycall, g_cur_file);
else
    myfunc := GetCalledFunction(mycall);
    Push myfunc into func_set;
end if
foreach func in func_set do
    HandleFunc(func, path_stack, lock_stack);
end foreach
```

```
HandleFunc(myfunc, path_stack, lock_stack)
if FindFuncSummary(myfunc, lock_stack, g_intr_flag) == TRUE then
    return;
end if
AddFuncSummary(myfunc, lock_stack, g_intr_flag);
if g_intr_flag == TRUE or HaveSpinFuncCall(myfunc) == FALSE then
    /* Flow-insensitive analysis */
    foreach call in FunctionCallList(myfunc) do
        HandleCall(call, path_stack, lock_stack);
    end foreach
else
    /* Flow-sensitive analysis */
    entry_block := GetEntryBlock(myfunc);
    HandleBlock(entry_block, path_stack, lock_stack);
end if
```

Figure 2. Atomic context analysis.

pointer is called from the *ixgbe* driver, only `ixgbe_get_stats64` can actually be called at this call site. The functions `rtl8139_get_stats64` and `usbnet_get_stats64` cannot be called, as their source files are not related to the *ixgbe* driver. `ixgbe_get_stats64`, on the other hand, is stored in the `ndo_get_stats64` field of a `net_device_ops` data structure in a source file of the *ixgbe* driver.

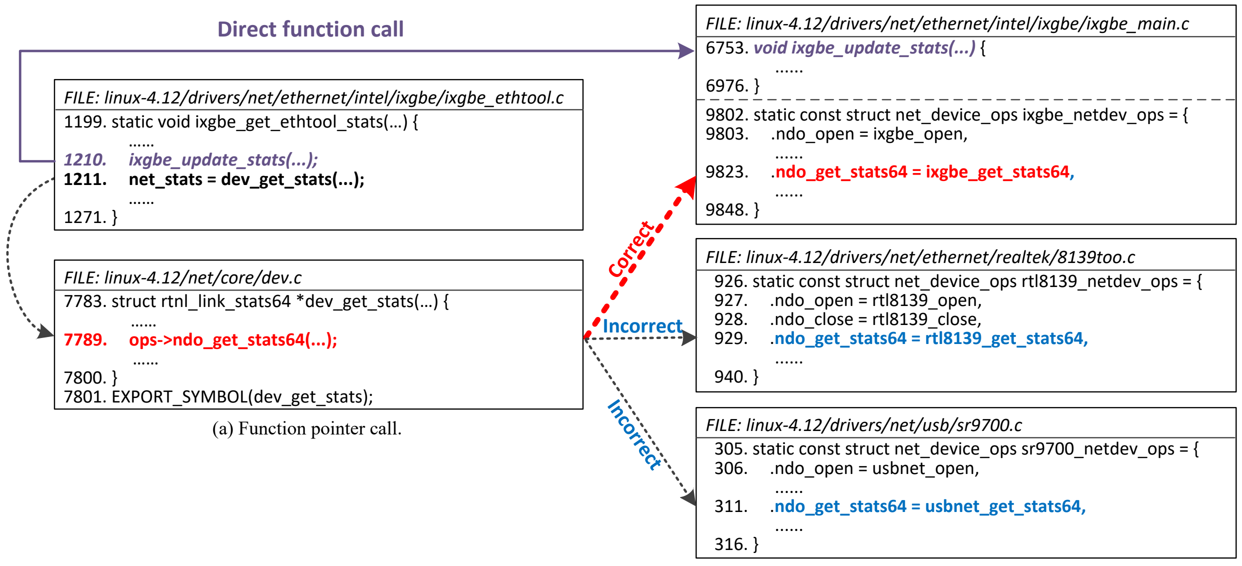
Based on this example, we observe that for a function to be invoked from a function-pointer call site, there needs to be some form of communication between the file in which the function is defined and the file containing the call site. Thus, a function can only be relevant at a function-pointer call site if such communication exists. The most basic form of communication, which does not require the prior establishment of any function pointers, is via a direct function call. If there are direct function calls between two files, we say that they have a file connection. Specifically, *if there exists a direct function call between File A and File B (File A → File B or File B → File A), then File A and File B have a direct connection.* Similarly, if there exists a direct function call from *File A* to *File B* (*File A* → *File B*) and from *File B* to *File C* (*File B* → *File C*), then *File A* and *File C* have an indirect connection. We thus filter possible functions at a function-pointer call using the transitive closure of file connections.

Consider again the example of Figure 3. There exists a direct function call `ixgbe_update_stats` from *ixgbe_main.c* to *ixgbe_ethtool.c*, and thus these two source files have a connection. But there does not exist any sequence of calls between *8139too.c* or *sr9700.c* and *ixgbe_ethtool.c*, and thus these files do not have any connection.

Based on our observation, we propose a file-connection-based alias analysis, to identify the set of functions referenced by the function pointer. For efficiency, our alias analysis is flow-insensitive. Moreover, in the Linux kernel, many function pointers are obtained from data structure fields [4], especially in device drivers, as illustrated in Figure 3. To handle such function pointers in large-scale code, our alias analysis performs a field-based analysis of data structure fields [18]. Our alias analysis has two steps:

Step 1: Collect code information. This step is performed before the atomic context analysis in Section 4.1. It traverses the Linux kernel code to identify the assignments to function pointers, and collects the possible function values in each case. Specifically, for each function pointer represented by a data structure field, our alias analysis extracts the data structure type, field offset and function-pointer type to differentiate it from other function pointers. Moreover, our alias analysis also collects direct file connections, representing direct function calls. All of the above information is stored in a database, and will be used in the next step.

Step 2: Identify the set of functions. This step, defined in Figure 4, is performed when the atomic context analysis encounters a function-pointer call, to identify the set of functions referenced by the function pointer. The inputs of this step



```
FILE: linux-4.12/drivers/net/ethernet/intel/ixgbe/ixgbe_ethtool.c
1199. static void ixgbe_get_ethtool_stats(...) {
        ......
1210.   ixgbe_update_stats(...);
1211.   net_stats = dev_get_stats(...);
        ......
1271. }
```

```
FILE: linux-4.12/net/core/dev.c
7783. struct rtnl_link_stats64 *dev_get_stats(...) {
        ......
7789.   ops->ndo_get_stats64(...);
        ......
7800. }
7801. EXPORT_SYMBOL(dev_get_stats);
```

(a) Function pointer call.

```
FILE: linux-4.12/drivers/net/ethernet/intel/ixgbe/ixgbe_main.c
6753. void ixgbe_update_stats(...) {
        ......
6976. }
9802. static const struct net_device_ops ixgbe_netdev_ops = {
9803.   .ndo_open = ixgbe_open,
        ......
9823.   .ndo_get_stats64 = ixgbe_get_stats64,
        ......
9848. }
```

```
FILE: linux-4.12/drivers/net/ethernet/realtek/8139too.c
926. static const struct net_device_ops rtl8139_netdev_ops = {
927.   .ndo_open = rtl8139_open,
928.   .ndo_close = rtl8139_close,
929.   .ndo_get_stats64 = rtl8139_get_stats64,
       ......
940. }
```

```
FILE: linux-4.12/drivers/net/usb/sr9700.c
305. static const struct net_device_ops sr9700_netdev_ops = {
306.   .ndo_open = usbnet_open,
       ......
311.   .ndo_get_stats64 = usbnet_get_stats64,
       ......
316. }
```

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

Figure 3. Example of identifying functions referenced by the function pointer.

```
Procedure: FileConnAliasAnalysis
Input: func_ptr_call - function pointer call;
       src_file - name of the source file where atomic context analysis starts
Output: func_set - set of the functions referenced by the function pointer
 1: func_set := ø;
 2: func_ptr := GetCalledValue(func_ptr_call);
 3: if func_ptr is a data structure field then
 4:     pos_func_set := FindFuncDataStructField(func_ptr);
 5: else
 6:     pos_func_set := FindFuncPossibleValue (func_ptr);
 7: end if
 8: foreach pos_func in pos_func_set do
 9:     pos_src_file := GetSourceFile(pos_func);
10:     if HaveFileConnection(pos_src_file, src_file) == TRUE then
11:         AddFuncSet(pos_func, func_set);
12:     end if
13: end foreach
14: return func_set;
```

Figure 4. Procedure of file-connection-based alias analysis.

are the function-pointer call *func_ptr_call* and the name of the source file *src_file* where the atomic context analysis started. The output is a set of functions *func_set* that contains the functions referenced by the function pointer. This step first checks the form of the function-pointer expression. If it has the form of a data structure field reference, the analysis extracts the functions according to the type of this data structure, the offset of the field and the function-pointer type (line 4). Otherwise, the analysis extracts all functions stored in any function pointer of the given type (line 6). The analysis then returns the subset of these functions for which there is a direct or indirect file connection, as identified in Step 1, between the file containing the starting point of the atomic context analysis and the file defining the given function (lines 8-14).

The main advantage of our file-connection-based alias analysis is to reduce the amount of incorrect functions identified for function pointers. This reduction can benefit the detection of CNS defects in two aspects: (1) These incorrect functions are not analyzed during the atomic context analysis, which can improve the efficiency of defect detection. (2) The non-sleep function calls in these incorrect functions are not marked in atomic context analysis, which can reduce the number of false negatives.

5 Approach

Based on the two techniques in Section 4, we propose a practical static analysis approach named DCNS, to automatically detect conservative non-sleep (CNS) defects in the Linux kernel. We implement DCNS using the Clang compiler [7], and store the code information collected during analysis in a MySQL database [26]. DCNS performs static analysis on the LLVM bytecode of the Linux kernel. Figure 5 shows the overall architecture of DCNS, which consists of four modules:

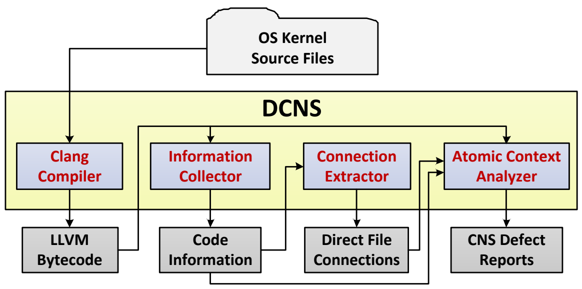


Figure 5. Overall architecture of DCNS.

- *Clang compiler.* This module compiles each source file of the Linux kernel into an LLVM bytecode file.
- *Information collector.* This module traverses the Linux kernel code by analyzing each LLVM bytecode file, to record useful code information in the database. This code information is used in the remaining analyses.
- *Connection extractor.* This module uses the collected code information to extract direct file connections, and records them in the database.
- *Atomic context analyzer.* This module performs atomic context analysis to mark non-sleep function calls that may be executed in atomic context, and then reports the remaining ones as CNS defects.

Based on this architecture, DCNS has three phases. We introduce them below.

5.1 Code Information Collection

In this phase, the code compiler first compiles each source file into an LLVM bytecode file. Then, by analyzing each LLVM bytecode file, the code information collector extracts and records the name and position of each function definition, the callee and caller functions of each direct function call, the possible functions stored in function pointers and the position of each non-sleep function call.

5.2 Direct File Connection Extraction

In this phase, the connection extractor extracts direct file connections, using the collected code information. The extractor analyzes each function call to check whether the call and the definition of the called function are in different source files. If so, the two source files are regarded as having a direct file connection, and this connection is recorded in the database.

5.3 Code Analysis of Atomic Context

In this phase, the atomic context analyzer performs summary-based atomic context analysis on the LLVM bytecode files, using the collected code information. It marks non-sleep function calls that may be executed in atomic context. To handle function-pointer calls, the analyzer uses the file connection information to identify the set of functions referenced by the function pointer, and processes each function in the resulting set. Finally, the analyzer reports unmarked non-sleep function calls as CNS defects.

The first two phases of DCNS work on individual files independently, benefiting from the collected information. Thus, these phases can be parallelized straightforwardly. The atomic context analysis traces through function calls, which may cross file boundaries, when the called function is defined in another file and no function summary is available for the current context. In this case, we can also run the analysis in parallel, allowing the global database of function summaries to prevent repeated analysis of functions in the same context across different threads.

Table 3. Detection of known defects.

Root Cause	**Defects**	**DCNS**	**Example Commit IDs**
mdelay	16	16	8c15d66e429a, 3e0f86b33709
GFP_ATOMIC	30	26	9ec6f6b89a13, 55bf851b4ad7
Total	46	42	9631739f8196 (not found by DCNS)

6 Evaluation

6.1 Experimental Setup

To validate the effectiveness of DCNS, we evaluate it on Linux kernel code. We run the evaluation on an x86-64 PC with four Intel i5-3470@3.20GHz processors and 8GB physical memory. We use the kernel configuration *allyesconfig* to enable all kernel code for the *x86* architecture. We compile the kernel code using Clang 6.0 [7]. Because DCNS can work in parallel, we configure it to run on 4 threads.

In the evaluation, we consider two common kinds of non-sleep operations in the Linux kernel, which have been involved in many previously reported CNS defects:

- Calls to `mdelay` with a waiting time $\geqslant$ 20ms. The function `mdelay` is used to busily wait for a specified number of milliseconds. This function has a counterpart `msleep`, which can sleep to wait, when the waiting time is $\geqslant$ 20ms.[2]
- Calls to a function with the argument `GFP_ATOMIC`. Such a call is often used to allocate a resource. If the resource is not available, the call directly returns an error. `GFP_ATOMIC` has a corresponding sleep-able flag `GFP_KERNEL`, that allows the called function to sleep and wait for the resource to become available.[3]

Given this configuration information and the Linux kernel source code, DCNS works fully automatically, and produces readable CNS defect reports.

6.2 Detecting Known Defects

To check whether DCNS can find known CNS defects, we select the commits fixing CNS defects submitted from January 2016 to July 2017 that change `mdelay` to `msleep` and `GFP_ATOMIC` to `GFP_KERNEL`. In total, there are 15 commits (see Table 2), which fix 46 CNS defects. For each commit, we revert the kernel code to the status just before the commit, and run DCNS. Table 3 shows the results classified by the defect's root cause. DCNS finds 42 of the defects. It misses 4 defects, because the related source code is not enabled in the kernel configuration for the *x86* architecture. These results indicate that DCNS can find known CNS defects.

6.3 Detecting New Defects

We next check whether DCNS can find new CNS defects. We evaluate DCNS on Linux kernel version 4.16 (released in

[2]The Linux kernel documentation for sleep and delay: https://www.kernel.org/doc/Documentation/timers/timers-howto.txt

[3]The Linux kernel documentation for memory allocation: https://www.kernel.org/doc/htmldocs/kernel-api/API-kmalloc.html

Table 4. Detection results on Linux 4.16.

Description		DCNS
Code handling	Source files (.c)	16.6K
	Source code lines	12.1M
Atomic context analysis	Handled interrupt handler functions	1408
	Handled functions	181K
	Function summaries	323K
Function-pointer analysis	Handled function-pointer calls	23,680
	Unhandled function-pointer calls	1996
	Identified referenced functions	245K
	Discarded referenced functions	1435K
Defect detection	Non-sleep operations	4589
	Final CNS defects	1629
Time usage	P1: Code information collection	13m
	P2: Direct file connection extraction	10m
	P3: Code analysis of atomic context	56m
	Total	79m

April 2018). Table 4 summarizes the results, and we have the following findings:

(1) DCNS can scale to large code bases. It handles 12.1M source code lines from 16.6K source files in 79 minutes (elapsed time). The remaining 4.7M source code lines of the Linux kernel are not enabled for the *x86* architecture, and thus DCNS does not handle them.

(2) DCNS is effective in handling function-pointer calls. Firstly, DCNS successfully identifies the referenced functions for 23,680 function pointer calls, accounting for more than 92% of all encountered function pointer calls. The remaining 8% fall into two cases. One case is when there is no initialization of the function pointer in the code selected by the compiler as being relevant to the *x86* architecture, which causes 55% of the unhandled calls. In these cases, the code tests whether the pointer is null before making the call. The other case is when DCNS observes relevant assignments in the analyzed code, but the assignment is somehow too complex for DCNS to be able to identify the function(s) assigned to the function pointer, which causes 45% of the unhandled calls. Figure 6 shows an example of the latter case. `hfi1_handle_cnp_tb` is a global array of function pointers, and it is initialized on lines 426-429. The structure of the bytecode generated by LLVM for this code prevents DCNS from detecting the array initialization, so the function-pointer call on line 480 is not handled.

```
426 static hfi1_handle_cnp hfi1_handle_cnp_tbl[2] = {
427   [HFI1_PKT_TYPE_9B] = &return_cnp,
428   [HFI1_PKT_TYPE_16B] = &return_cnp_16B
429 };
430
431 void hfi1_process_ecn_slowpath(...) {
      ...
479   if (do_cnp && (bth1 & IB_FECN_SMASK))
480     hfi1_handle_cnp_tbl[hdr_type](...);
      ...
491 }
```

Figure 6. Example of unhandled function-pointer call. (File: linux-4.16/drivers/infiniband/hw/hfi1/driver.c)

Secondly, by using the file-connection-based alias analysis, DCNS identifies 245K functions referenced by handled function-pointer calls, and discards 1435K possible functions. Namely, 85% of the possible functions are discarded as they are likely to be incorrect in the calling context of the given function-pointer call. By discarding these incorrect possible functions, DCNS reduces unnecessary atomic context analysis, and also reduces false negatives in defect detection.

(3) DCNS is effective in detecting CNS defects. DCNS in total finds 1629 CNS defects, from 4589 non-sleep operations. We scan all the defects, and manually check 943 defects whose call paths are not so difficult to follow, and find that 890 are real, giving an accuracy rate of 94%. We randomly selected 300 of the real defects and sent them to kernel developers, and 251 have been confirmed. We do not manually check the remaining 686 defects, because their non-sleep function calls are touched by many code paths, and thus we are not confident in our manual analysis.

Among the 251 confirmed defects, 73 involve calls to `mdelay` with a waiting time $\geqslant$ 100ms, and 6 of the 73 have a waiting time of 1000ms. These calls busily wait for a long time with no benefit and can damage the overall system performance. Besides, 110 of the confirmed defects involve GFP_ATOMIC, with many in driver initialization functions, and are thus likely cause a user-visible failure.

(4) DCNS is efficient. In total, it handles 16.6K source files in 79 minutes running on four threads.

6.4 False Positive Analysis

False results mainly occur in two cases:

(1) Some interrupt handler functions are not identified, so the non-sleep function calls in these functions may be not marked and will be reported as CNS defects. This case is mainly caused by the fact that DCNS only identifies as interrupt handler functions the functions in specific arguments of known interrupt handler register functions (such as `request_irq` and `tasklet_init` in the Linux kernel), and when the argument is an explicit function. If the argument is a function pointer that is assigned at other places, DCNS cannot identify the concrete interrupt handler function. Figure 7 shows an example of this case. The interrupt handler register function `request_irq` uses a function pointer as an argument on line 391. This function pointer is assigned as the function `aac_sa_intr` on line 344, but DCNS does not track this information.

(2) Some function-pointer calls are not handled, so the non-sleep function calls in the referenced function may be unmarked and will be reported as CNS defects. These reported CNS defects are false positives. As described in Section 6.3, this case mainly occurs when the function pointer is assigned in complex ways that DCNS cannot handle.

Besides, another possible source of false positives is in code that is reachable from functions exported to dynamically loadable kernel modules. Such functions are declared

```
int aac_sa_init(...) {
    ...
    dev->a_ops.adapter_intr = aac_sa_intr;
    ...
    if (request_irq(..., dev->a_ops.adapter_intr,
                    IRQF_SHARED, ...) < 0)
    ...
}
```

Figure 7. Example of unhandled interrupt handler function. (File: linux-4.16/drivers/scsi/aacraid/sa.c)

using `EXPORT_SYMBOL` or similar macros. The goal of the Linux kernel developer community is that loadable kernel modules should have their source code included with the Linux source code distribution,[4] but in practice, this may not always be the case. DCNS does not see the code of such external modules, and these modules may invoke the exported functions with a different lock stack or interrupt handler state than that exhibited within the kernel source tree. If a non-sleep operation is only reachable in atomic context in this way, DCNS will report a false positive. To estimate the impact of this issue, we rerun DCNS extended to consider that all exported functions enter atomic context. DCNS finds 1101 defects, all of which are also found in the original results. Among them, 524 defects have been manually checked, and 492 are real. Nevertheless, we expect that generally, exported functions will be used by external Linux kernel modules in the same way that they are used by code in the Linux kernel source tree. And all exported function should be used somewhere within the Linux kernel [8], giving DCNS information about their expected use.

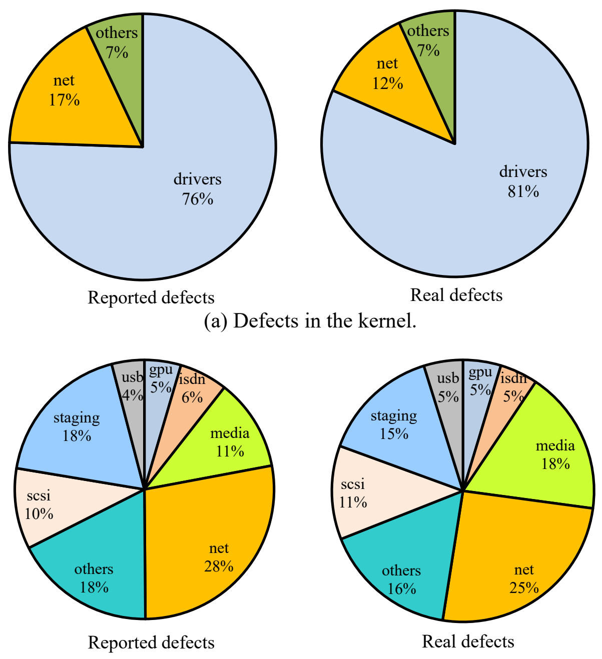
6.5 Defect Distribution

We classify the found CNS defects according to the directory of their source files. Figure 8 shows the distribution of all reported defects and the real defects identified by our manual check. Figure 8(a) shows the defect distribution in the whole kernel. We find that drivers have more than 75% of the defects. This is somewhat higher than the percentage of code represented by drivers (67% in Linux 4.16), and shows that drivers remain a significant source of system problems [35]. We then classify the driver defects by the driver class (Figure 8(b)). We find that network, staging, media and SCSI drivers are together the source of nearly 70% of the defects in drivers.

6.6 Sensitivity Analysis

DCNS uses two key techniques: a summary-based analysis to reduce repeated analysis of atomic context, and a file-connection-based alias analysis to correctly identify the set of functions referenced by a function pointer. To better understand the value of these two techniques, we modify DCNS

[4]The Linux kernel driver interface: https://www.kernel.org/doc/Documentation/process/stable-api-nonsense.rst



(b) Defects in drivers.

Figure 8. Defect distribution in the Linux kernel.

to remove each of them, and evaluate each modified tool on the Linux 4.16 kernel code.

Summary-based analysis. We implement the modified tool without using function summaries, which allows a function to be repeatedly handled in the same execution context. The PathHasExisted check is retained in lines 1-3 of HandleBlock (Figure 2), to protect against infinite loops due to recursion. The resulting tool runs for 700 minutes and finally aborts due to insufficient memory, without completing the analysis. This experiment indicates that our summary-based analysis indeed improves the efficiency of atomic context analysis.

File-connection-based analysis. We implement two modified tools. The first tool drops our file-connection-based analysis, and simply reports all functions that were collected as being possibly assigned to the function pointer. The other tool builds on our observation that only functions defined in related files are relevant. We implement the notion of a related file in a simpler way, by considering source files to be related when they are part of the same kernel module. Table 5 shows the results of the first tool (Tool 1) and the second tool (Tool 2).

For the first tool, there are three main differences between its results and those of DCNS. Firstly, the tool considers many functions as being referenced by each function pointer that are dropped by DCNS, and thus more functions pointers are considered to be handled than when using DCNS. But in fact, most of these functions are incorrect given the execution context of function-pointer calls. Secondly, the tool finds far fewer CNS defects than DCNS, and no additional ones. The modified tool marks more non-sleep function calls, which

Table 5. Sensitivity results of file-connection-based analysis.

Description		Tool 1	Tool 2
Function-pointer analysis	Handled function-pointer calls	30583	830
	Unhandled function-pointer calls	2491	8975
	Identified referenced functions	2723K	1586
	Discarded referenced functions	0	142K
Defect detection	Final CNS defects	627	2954
	Manually checked defects	244	943
	Real defects	224	890
Time usage	P1: Code information collection	13m	13m
	P2: Direct file connection extraction	0m	10m
	P3: Code analysis of atomic context	351m	31m
	Total	364m	54m

are in the functions newly identified as being referenced by functions pointers. But in fact, most of these calls are indeed real CNS defects, and thus false negatives. Referring to our manually checked results in DCNS, we find that the first tool finds 244 defects that have been manually checked, and 224 are real defects. Finally, the tool requires much more time than DCNS, as it analyzes more functions that are considered to be referenced by function pointers than DCNS does.

For the second tool, there are also three main differences between its results and those of DCNS. Firstly, the tool handles fewer function-pointer calls than DCNS. Indeed, many kernel modules are only meant to be loaded in conjunction with other kernel modules, and there may be direct interactions between them. But the tool cannot detect such a relationship from the link information, and thus drops many correct functions referenced by function pointer calls, hiding calls that initiate atomic context. Secondly, the tool reports many CNS defects that are not reported by DCNS. But in fact, most of these defects are false positives, because many function-pointer calls are not handled. Finally, the tool is faster than DCNS, because the tool analyzes fewer functions that are considered to be referenced by function pointers.

Overall, we find that DCNS has fewer false negatives than the first tool, and has fewer false positives than the second tool. Thus, our file-connection-based analysis indeed improves the accuracy of CNS defect detection.

7 Discussion

In this section, we discuss three interesting things about CNS defects and our approach:

CNS defects are "similar" but "reverse" to SAC bugs. CNS defects are similar to SAC bugs in four ways: 1) They are both related to atomic context. 2) They are both introduced when the developer does not understand what code may be executed in atomic context. 3) They both only occur in kernel-mode code, because user-mode applications never have atomic context. 4) Detecting them both requires atomic context analysis to identify the code that may be executed in atomic context.

There are also four ways in which the defect types are the reverse of each other: 1) CNS defects are caused by the non-sleep operations that are never in atomic context, while SAC bugs are caused by the sleep-able operations that are in atomic context. 2) CNS defects are introduced because developers are too conservative in using sleep-able operations, while SAC bugs are introduced because developers are too aggressive in using sleep-able functions. 3) During atomic context analysis, missing code paths that are executed in atomic context may introduce false positives when detecting CNS detects, while they may introduce false negatives when detecting SAC bugs. 4) To fix CNS defects, related non-sleep operations should be changed to sleep-able operations, while to fix SAC bugs, related sleep-able operations can be changed to non-sleep operations in some cases.

Our previous approach DSAC is related to DCNS, but it is different. DSAC [3] is our previous approach of detecting sleep-in-atomic-context (SAC) bugs. DSAC and DCNS both check the Linux kernel and identify the code in atomic context. But they are quite different due to three main aspects. Firstly, DSAC targets SAC bugs, while DCNS targets DCNS defects. Secondly, DSAC uses a hybrid of flow-sensitive and -insensitive analysis to handle one kernel module at a time, and it only analyzes code paths within the module. DCNS uses a summary-based analysis to handle the whole kernel code involving multiple kernel modules, and it analyzes code paths across different kernel modules. Finally, DSAC does not consider function pointers, but still finds real bugs. However, without handling function pointers, DCNS produces many false positives, because it has only partial knowledge of the execution context of the faulty code and cannot cover complete code paths of non-sleep operations. Thus, DCNS uses a file-connection-based alias analysis to identify the set of functions referenced by a function pointer.

Our file-connection-based alias analysis is applicable to other kernel problems. As shown in Section 6.3, 55% of unhandled function calls are caused by no initialization of a relevant function pointer. In these cases, if the code does not test whether the pointer is null before making the call, a null pointer dereference bug may occur. In fact, we did find several such bugs that have been confirmed by kernel developers, although this is not the main focus of this work and thus we do not provide further details. Moreover, our alias analysis can help to build a full call graph of the kernel code, from which it should be possible to detect hard-to-find kernel bugs, such as double free and double lock errors.

8 Related Work

8.1 Bug Finding in Systems Software

Engler et al. [13] made the first static analysis that could scale up to complete real OS kernels, by accepting the possibility of some false negatives and false positives. Chou et al. [6] followed up on this work by using the proposed analysis to conduct an extensive survey of bugs across the history of the Linux kernel; this work was reprised a decade later by Palix et al. [27] for more recent kernel versions. Although some of the considered bug types, including sleeping in atomic context, can benefit from inter-procedural analysis, they all involve finding the existence of a faulty execution path, rather than requiring checking all possible execution paths, as needed for detecting CNS defects. A paper by Hallem et al. [16] that presents the analyses behind the former works does not mention function pointers.

Inspired by the initial works, a number of bug finding approaches targeting systems code have been proposed. Engler et al. [12] and Li et al. [22] use statistics to mine and then check API usage rules. These approaches focus on common patterns occurring within a single function. Lawall et al. [20] propose patterns that can be instantiated to find various kinds of bug types, such as incorrect return values, but again do not consider function pointers. Saha et al. [30] find resource-release omission faults by comparison with other resource-release operations in the same function. This work includes inter-procedural analysis, but only across direct calls to other functions in the same file. Our previous approach DSAC [3] detects sleep-in-atomic-context bugs in kernel modules, but it does not handle function pointers.

Several previous approaches do take into account function pointers. Gunawi et al. [15] and Rubio-González et al. [29] study the propagation of errors in file system code. Like our approach, they focus on function pointers stored in data structure fields. They collect alias information based on static initializations and direct assignments, and use the collected information to transform function-pointer calls to `switch` statements considering all collected possibilities. We initially tried a similar approach, but found that it led to too many false positives. Indeed, these previous approaches analyze a single module at a time, and thus they do not need to handle function pointers across multiple modules. But finding CNS defects typically requires starting from the kernel core and tracing execution paths outward between the core and the individual kernel modules, which greatly increases the risk of confusion.

Smatch [31] is an open-source static analysis tool that mainly targets finding bugs in the Linux kernel. It collects function pointers that are stored in a given structure field or array element, passed into a particular function parameter, or returned from a particular function. The set of pointers considered at a given call site is refined by function arity. The precision is thus comparable to that of Gunawi et al. and Rubio-González et al., as discussed above. Smatch does not provide a checker for CNS defects.

8.2 Alias Analysis

Many approaches have been proposed to analyze pointers in C/C++ programs. Most of these approaches are based on either Andersen's algorithm [1] (*e.g.* [17, 34, 37, 39]) or Steensgaard's algorithm [33] (*e.g.* [10, 14, 23, 38]). Some approaches focus on handling pointers in data structure fields, and provide field-sensitive (*e.g.* [2, 11, 19, 28]), or field-based (*e.g.* [18, 21, 32]) analysis. Field-sensitive analysis uses a separate variable to model the field information for each instance, while field-based analysis uses one variable to model all instances of a particular data structure field. For example, Lattner et al. [19] propose a field-sensitive and flow-insensitive pointer analysis algorithm named DSA. DSA is Steensgaard-style, and uses some optimizations to speed up the analysis. Heintze and Tardieu [18] propose a field-based and flow-insensitive pointer analysis approach. It is Andersen-style, and uses the data structure type and field name to maintain field sensitivity.

Some of the above approaches (*e.g.* [17–19, 25]) can handle function pointers, but they are reported to produce many false results on such data in large and complex software. Indeed, they do not consider the calling context of a function-pointer call, and just produce all the functions possibly referenced by the called function pointer according to type and field information. Thus, many reported functions may be false for a given function-pointer call. In this paper, building on the field-based analysis of Heintze and Tardieu, we add file connection guidance to improve the analysis accuracy of function-pointer calls.

9 Conclusion

In this paper, we have proposed a practical static approach named DCNS, to automatically detect conservative non-sleep defects in the Linux kernel. DCNS uses two key techniques: 1) a summary-based analysis to effectively identify the code in atomic context; 2) a file-connection-based alias analysis to correctly identify the set of functions referenced by a function pointer. We evaluated DCNS on Linux 4.16, and in total found 1629 CNS defects. We manually checked 943 defects whose call paths are not so difficult to follow, and found that 890 are real. We randomly selected 300 of the real defects and sent them to kernel developers, and 251 have been confirmed.

As future work, DCNS can be improved in several ways. Firstly, the current implementation of DCNS can be improved to reduce false positives. For example, DCNS fails to handle some function-pointer calls, typically when the function pointer is initialized in complex ways (like function-pointer arrays). Secondly, we have only evaluated DCNS for the Linux kernel. In fact, sleep-in-atomic-context defects have been also found in FreeBSD and NetBSD kernel code [3], and thus these kernels may exhibit CNS defects as well. Finally, our file-connection-based alias analysis is applicable to other kernel problems.

Acknowledgments

We would like to thank the Linux kernel developers who gave helpful feedback on our defect reports and patches. Shi-Min Hu is the corresponding author.

References

[1] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Copenhagen.

[2] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. 2005. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering (ICSE).* 332–341.

[3] Jia-Ju Bai, Yu-Ping Wang, Julia Lawall, and Shi-Min Hu. 2018. DSAC: effective static analysis of sleep-in-atomic-context bugs in kernel modules. In *Proceedings of the 2018 USENIX Annual Technical Conference.* 587–600.

[4] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. 2011. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 8, 5 (2011), 670–684.

[5] Checkpatch 2018. Checkpatch.pl: a perl script to check coding style. https://github.com/ torvalds/linux/blob/master/scripts/checkpatch.pl.

[6] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. 2001. An empirical study of operating system errors. In *Proceedings of the 18th International Symposium on Operating Systems Principles (SOSP).* 73–88.

[7] Clang 2018. Clang compiler. http://clang.llvm.org/.

[8] Jonathan Corbet. 2007. Exported symbols and the internal API. https://lwn.net/Articles/249246/.

[9] Jonathan Corbet. 2008. Atomic context and kernel API design. https://lwn.net/Articles/274695/.

[10] Manuvir Das. 2000. Unification-based pointer analysis with directional assignments. In *Proceedings of the 21st International Conference on Programming Language Design and Implementation (PLDI).* 35–46.

[11] M. Emami, R. Ghiya, and L. Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the 1994 International Conference on Programming Language Design and Implementation (PLDI).* 242–256.

[12] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the 18th International Symposium on Operating Systems Principles (SOSP).* 57–72.

[13] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI.* 1–16.

[14] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. 2000. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the 21st International Conference on Programming Language Design and Implementation (PLDI).* 253–263.

[15] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. 2008. EIO: error handling is occasionally correct. In *Proceedings of the 6th International Conference on File and Storage Technologies (FAST).* 207–222.

[16] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. 2002. A system and language for building system-specific, static analyses. In *Proceedings of the 23rd International Conference on Programming Language Design and Implementation (PLDI).* 69–82.

[17] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th International Symposium on Code Generation and Optimization (CGO).* 289–298.

[18] Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the 22nd International Conference on Programming Language Design and Implementation (PLDI).* 254–263.

[19] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI).* 278–289.

[20] Julia L. Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. 2009. 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).* 43–52.

[21] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *Proceedings of the 2003 International Conference on Compiler Construction (CC).* 153–169.

[22] Zhenmin Li and Yuanyuan Zhou. 2005. 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).* 306–315.

[23] V Benjamin Livshits and Monica S Lam. 2003. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 11th International Symposium on Foundations of Software Engineering (FSE).* 317–326.

[24] LLVM 2018. LLVM compiler infrastructure. https://llvm.org/.

[25] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2004. Precise call graphs for C programs with function pointers. *Automated Software Engineering* 11 (2004), 7–26. Issue 1.

[26] MySQL 2018. MySQL database. https://www.mysql.com.

[27] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. 2014. Faults in Linux 2.6. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 4:1–4:40.

[28] David J Pearce, Paul HJ Kelly, and Chris Hankin. 2007. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 1 (2007), 4:1–4:42.

[29] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI).* 270–280.

[30] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2013. 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).* 1–12.

[31] Smatch 2018. Smatch: a static bug-finding tool for the Linux kernel. http://smatch.sourceforge.net/.

[32] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *OOPSLA.* 59–76.

[33] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd International Symposium on Principles of Programming Languages (POPL).* 32–41.

[34] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. 2014. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Software: Practice and Experience (SPE)* 44, 12 (2014), 1485–1510.

[35] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2003. Improving the reliability of commodity operating systems. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP).* 207–222.

[36] Sysbench 2018. Sysbench: a scriptable database and system performance benchmark. https://wiki.gentoo.org/wiki/Sysbench.

[37] John Whaley and Monica S Lam. 2002. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 2002 International Static Analysis Symposium (SAS).* 180–195.

[38] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-based selective flow-sensitive pointer analysis. In *Proceedings of the 2014 International Static Analysis Symposium (SAS).* 319–336.

[39] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. 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).* 218–229.