DLOS: Effective Static Detection of Deadlocks in OS Kernels

Jia-Ju Bai, Tuo Li, and Shi-Min Hu, *Tsinghua University*

https://www.usenix.org/conference/atc22/presentation/bai

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

DLOS: Effective Static Detection of Deadlocks in OS Kernels

Jia-Ju Bai
Tsinghua University

Tuo Li
Tsinghua University

Shi-Min Hu
Tsinghua University

Abstract

Deadlocks in OS kernels can cause critical problems like performance degradation and system hangs. However, detecting deadlocks in OS kernels is quite challenging, due to high complexity of concurrent execution and large code bases of OS kernels. In this paper, we design a practical static analysis approach named DLOS, to effectively detect deadlocks in OS kernels. DLOS consists of three key techniques: (1) a *summary-based lock-usage analysis* to efficiently extract the code paths containing distinct locking constraints from kernel code; (2) a *reachability-based comparison method* to efficiently detect locking cycles from locking constraints; (3) a *two-dimensional filtering strategy* to effectively drop false positives by validating code-path feasibility and concurrency. We have evaluated DLOS on Linux 5.10, and find 54 real deadlocks, with a false positive rate of 17%. We have reported these deadlocks to Linux kernel developers, and 31 of them have been confirmed.

1 Introduction

Concurrent execution improves the performance of OS kernels, but can inevitably introduce concurrency bugs. Some studies [37, 38, 51] have shown that a large part of reported OS bugs are related to kernel concurrency. Deadlock is a common kind of concurrency bugs, caused by a locking cycle in different threads. For example, one thread acquires the locks *A* and then *B*, while the other concurrent thread acquires the locks *B* and then *A*, and thus a deadlock caused by the ABBA locking cycle occurs. Deadlocks in OS kernels are dangerous, because they can infinitely block the involved threads, causing performance degradation and even system hangs.

To find deadlocks, many existing approaches [5, 9–11, 15, 21, 27, 28, 31, 33, 44, 45, 55] dynamically monitor thread execution and lock-related operations to detect locking cycles. These approaches have shown promising results in both user-level applications and OS kernels. For example, Lockdep [33] is a widely-used lock-usage validator integrated in the Linux kernel. It detects deadlocks, double locks and other locking issues, by dynamically tracking the state of each lock class and checking the dependencies between different lock classes. However, dynamic analysis approaches require well-constructed workloads or substantial test cases to cover the code containing bugs, and thus their detection coverage is often limited in runtime testing.

To improve detection coverage, some approaches [30, 41, 42, 46, 52] use static analysis to detect deadlocks in user-level applications. However, these approaches are ineffective in detecting deadlocks in OS kernels, for two main reasons. First, these approaches requires a fixed entry point (such as a `main` function) to start dataflow analysis; but an OS kernel consists of many kernel modules, each of which has no such a fixed entry point [4, 43]. Second, these approaches need to identify concurrent code and perform concurrency alias analysis according to thread-creation function calls (such as the calls to `pthread_create`) and related arguments; but the concurrency of OS kernel is often determined by the concurrent execution of specific interface functions in each kernel module [2], not explicitly calling thread-creation functions.

To our knowledge, RacerX [19] is the sole existing static analysis approach to systematically detect deadlocks in OS kernels. It uses *locking constraint* to describe the locking situation when each lock is acquired, e.g., if a code path acquires the locks *A* and then *B*, its locking constraint is $A \rightarrow B$. RacerX performs flow-sensitive and inter-procedural analysis to identify the code paths containing locking constraints (such code paths are referred to as *target code paths* subsequently) from OS kernel code, and then recursively compares between each two target code paths with their locking constraints to detect locking cycles as deadlocks.

However, RacerX still has some limitations. First, though using some heuristic techniques (like result ranking), RacerX still has a high false positive rate of 46%, due to neglecting the feasibility and concurrency of code paths. Second, RacerX neglects alias relationships, causing both false positives and negatives. Finally, RacerX simply compares between each two target code paths with their locking constraints in a recursive

way to detect locking cycles. This method works well for old OS kernels (like Linux 2.5.62 checked in its paper), but can be inefficient for modern OS kernels (like Linux 5.10 checked in our evaluation) that are much larger and more complex. Since the RacerX paper published in 2003, no new static approach has been proposed to systematically detect deadlocks in OS kernels. Thus, it is important to design a new static approach to perform effective deadlock detection in modern OS kernels.

In this paper, we design a practical static analysis approach named DLOS, to effectively detect deadlocks in OS kernels. DLOS consists of three key techniques:

(1) DLOS uses a *summary-based lock-usage analysis* to efficiently extract the code paths containing distinct locking constraints from kernel code. Our analysis uses function summaries to avoid repeated code analysis in the same functions, and it also drops the target code paths containing repeated locking constraints. To improve accuracy, our analysis is flow-sensitive and inter-procedural with the consideration of alias relationships, and it also uses a light-weight method to validate code-path feasibility with an SMT solver.

(2) DLOS uses a *reachability-based comparison method* to efficiently detect locking cycles from locking constraints. We observe that there are substantial target code paths containing distinct locking constraints in modern OS kernels (like Linux 5.10). Thus, when detecting locking cycles, simply comparing between each two target code paths with their locking constraints in a recursive way is quite time-consuming. To solve this problem, for each target code path, our method maintains a constraint reachability graph to store the locking constraints that are reachable starting the comparison from this code path, and the involved target code paths. By using constraint reachability graphs, our method can reduce repeated comparison of locking constraints, to improve the detection efficiency. If a locking cycle is found, it is considered as a possible deadlock, with the involved target code paths and locking constraints.

(3) DLOS uses a *two-dimensional filtering strategy* to effectively drop false positives, by validating the feasibilty and concurrency of target code paths for each possible deadlock. On the one hand, as using an SMT solver to completely validate the feasibility of each code path is quite time-consuming, our strategy validates code paths in a phased way. Specifically, during locking-constraint extraction, as substantial code paths are required to be validated, making the validation efficiency more important, our strategy uses a simple and light-weight path-condition checking method to drop obviously infeasible code paths containing locking constraints; and then after locking-cycle detection, as the code paths of possible deadlocks should occupy a very small proportion of all the code paths, making the validation accuracy more important, our strategy uses a complete and heavy-weight path-condition checking method to drop false deadlocks. On the other hand, for each possible deadlock, our strategy checks the concurrency of its code paths, by analyzing their call graphs and looking for common locks acquired in these code paths.

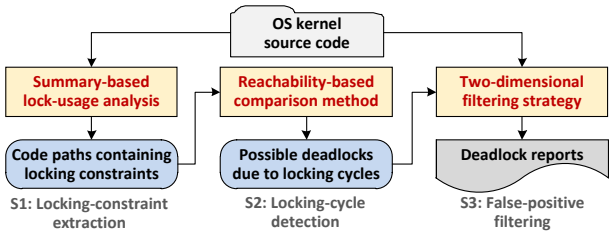

Figure 1: DLOS workflow.

Overall, DLOS has three main stages shown in Figure 1. In Stage 1, DLOS uses our summary-based lock-usage analysis to extract the code paths containing distinct locking constraints. In Stage 2, according to the extracted code paths and locking constraints, DLOS uses our reachability-based comparison method to detect possible deadlocks. In Stage 3, DLOS uses our two-dimensional filtering strategy to check possible deadlocks and drop false positives. After these stages, DLOS reports the final reports of the found deadlocks.

Compared to RacerX, DLOS has two main advantages. First, DLOS can achieve better accuracy than RacerX, by validating code-path feasibility with an SMT solver, considering alias relationships and checking the concurrency of the involved code paths for reported deadlocks. Second, DLOS can spend less time than RacerX, by extracting and comparing locking constraints more efficiently.

We have implemented DLOS with LLVM [32] and Z3 [54]. DLOS performs automated static analysis on the LLVM bytecode of the checked OS kernel. Overall, we make three main contributions in this paper:

- We analyze the challenges of static deadlock detection in OS kernels, and propose three key challenges to address these challenges: (1) a *summary-based lock-usage analysis* to efficiently extract the code paths containing distinct locking constraints from kernel code; (2) a *reachability-based comparison method* to efficiently detect locking cycles from locking constraints; (3) a *two-dimensional filtering strategy* to effectively drop false positives by validating code-path feasibility and concurrency.
- Based on these three key techniques, we design a practical static analysis approach named DLOS, to effectively detect deadlocks in OS kernels.
- We evaluate DLOS on Linux 4.9 and 5.10, and find 46 and 65 deadlocks, respectively. We manually check these deadlocks, and find that 39 and 54 deadlocks are real. 21 of the real deadlocks found in Linux 4.9 have been fixed in Linux 5.10. We have reported the 54 real deadlocks found in Linux 5.10 to Linux kernel developers, and 31 of them have been confirmed.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 introduces the challenges of static deadlock detection in OS kernels and our key techniques to address these challenges. Section 4 introduces DLOS. Section 5 shows our evaluation. Section 6

makes a discussion about DLOS. Section 7 presents related work, and Section 8 concludes this paper.

2 Background and Motivation

We first introduce deadlock and its detection, then explain the concurrency model of OS kernels, and finally motivate our work using a real deadlock in the Linux kernel.

2.1 Deadlock and Its Detection

To protect critical data from concurrent accesses, several kinds of synchronization primitives are designed and used. Locks are the most frequently-used synchronization primitives in real-world programs, to guarantee atomicity and prevent data races. However, if locks are incorrectly used, a deadlock can occur when one thread holds a lock that other concurrent threads want to acquire and vice versa.

Locking cycles in concurrent threads can cause deadlocks. The most common case is the ABBA lock in two threads, as shown in Figure 2(a). Namely, one thread acquires the locks *A* and then *B* (*A*→*B*), while the other concurrent thread acquires the locks *B* and then *A* (*B*→*A*), causing a locking cycle (*A*→*B*, *B*→*A*). In three or more threads, deadlocks can also occur due to such locking cycles, as shown in Figure 2(b).

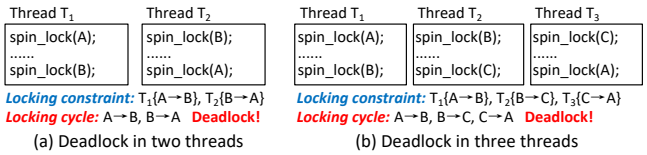

Figure 2: Deadlock examples.

For dynamic analysis, deadlock detection has two basic steps, namely extracting locking constraints in concurrent threads and then comparing these locking constraints to detect locking cycles. For static analysis, deadlock detection is similar but more complex. On the one hand, without exact runtime information about thread execution, static analysis has to identify locking constraints from each code path and validate the concurrency of code paths. On the other hand, without exact values of accessed variables, static analysis has to validate the feasibility of code paths using an SMT solver.

2.2 Concurrency Model of the OS Kernel

A modern OS kernel consists of many kernel modules, including filesystems, network modules, device drivers, etc. Each kernel module has some specific interface functions that are called by upper-level programs, including other kernel modules via function-pointer calls and user-level applications via system calls. Figure 3 shows some examples of interface functions that are assigned to function-pointer fields. These interface functions form the entry points of the kernel module,

```
FILE: linux-5.10/fs/gfs/file.c
1108. file_operations gfs2_file_fops = {
1109.     .llseek = gfs2_llseek,
1110.     .read_iter = generic_file_read_iter,
1111.     .write_iter = gfs2_file_write_iter,
1112.     .unlocked_ioctl = gfs2_ioctl,
          ......
1123. }
```

```
FILE: linux-5.10/drivers/net/wan/lmc_main.c
808. struct net_device_ops lmc_ops = {
809.     .ndo_open = lmc_open,
810.     .ndo_stop = lmc_close,
811.     .ndo_change_mtu = hdlc_change_mtu,
812.     .ndo_start_xmit = hdlc_start_xmit,
          ......
816. }
```

Figure 3: Examples of interface functions in kernel modules.

and all other functions defined in the kernel module are called by them [4, 43]. Due to this execution model, the concurrency of OS kernel is often determined by the concurrent execution of specific interface functions in each kernel module [2]. In fact, a kernel module can also explicitly call thread-creation functions (such as `kthread_create` in the Linux kernel), but such operations are not common in kernel module code.

Different from the OS kernel, each user-level application has a fixed entry point (like a `main` function) and explicitly calls thread-creation functions (like `pthread_create`) to start concurrent execution. Accordingly, to detect deadlocks in user-level applications, existing static approaches [30, 41, 42, 46, 52] start dataflow analysis from this fixed entry point, and identify concurrent code for concurrency alias analysis according to thread-creation function calls and related arguments. Due to the difference between the concurrency models of OS kernels and user-level applications, these approaches are ineffective in detecting deadlocks in OS kernels.

2.3 Motivating Example

Figure 4 presents a real and already fixed deadlock in the *btrfs* filesystem, and this bug is found by our approach DLOS in the evaluation of checking Linux 4.9. When the function `btrfs_read_chunk_tree` is executed on the code path *P1*, it acquires the locks `root->fs_info->chunk_mutex` and then `orig->device_list_mutex`; when the function `btrfs_remove_chunk` is executed on the code path *P2*, it acquires the locks `fs_devices->device_list_mutex` and then `root->fs_info->chunk_mutex`. During filesystem execution, the functions `btrfs_read_chunk_tree` and `btrfs_remove_chunk` are able to be concurrently executed at runtime, and the locks `orig->device_list_mutex` and `fs_devices->device_list_mutex` can be identical, and thus an ABBA deadlock can occur. This deadlock was introduced by the commit 57ba4cb85bff [16] in Linux 4.7, and it was found by Lockdep [33] and fixed by the commit 01d01caf19ff [17] in Linux 5.9, after over 4 years later. In fact, Lockdep is integrated in the Linux kernel for dynamic deadlock detection, but it took Lockdep such a long time to find this deadlock, because the interleaving of the code paths *P1* and *P2* are infrequent in real execution.

This example illustrates why deadlocks occur in OS kernels. First, determining concurrent code paths and identifying the same locks in these code paths require substantial knowledge of OS kernels. In the example, without deep understanding of filesystems and extensive testing, it may

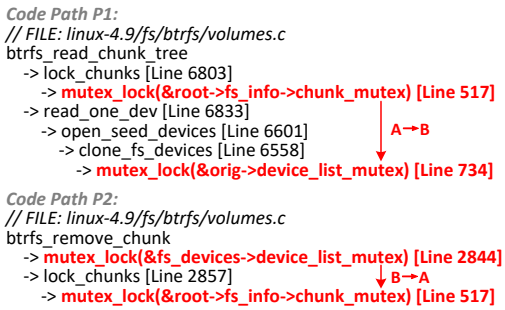

Figure 4: A real deadlock in Linux 4.9 *btrfs* filesystem.

be difficult to know code paths *P1* and *P2* can be concurrently executed, and the locks `orig->device_list_mutex` and `fs_devices->device_list_mutex` can be identical. Second, incorrect fixing of known bugs can introduce new and hard-to-find concurrency bugs. In the example, the commit 57ba4cb85bff introducing the deadlock aimed to fix a harmful data race in the functions `btrfs_remove_chunk` and `btrfs_dev_replace_finishing`, but this commit incautiously introduces a locking cycle in the functions `btrfs_remove_chunk` and `btrfs_read_chunk_tree`. Finally, multiple functions (including concurrent functions and the functions called by them) and variables in these functions need to be considered.

By scanning the reported deadlocks in the Linux kernel, we find that most of them are found in stress testing and kernel fuzzing. But the detection coverage of runtime testing heavily relies on the provided workloads, causing many real deadlocks to be missed. Static analysis can conveniently achieve high detection coverage without actual execution of the OS kernel. However, as the sole existing static approach of systematically detecting deadlocks in OS kernels, RacerX [19] still has many false positives, and its concurrency analysis can be inefficient to modern OS kernels (like Linux 5.10 checked in our evaluation) that are much larger and more complex than old OS kernels (like Linux 2.5.62 checked in the RacerX paper). Thus, it is important to design a new static approach to perform effective deadlock detection in modern OS kernels.

3 Challenges and Key Techniques

To detect deadlocks, static analysis needs to first extract the code paths containing distinct locking constraints (such code paths are referred to as *target code paths* subsequently) from kernel code, and then compare these code paths with their locking constraints to detect locking cycles as deadlocks. However, performing these steps for checking OS kernel code has three main challenges:

C1: Extracting locking constraints. A modern OS kernel is very large and complex, because it has many kernel modules and lots of functions with complicated call graphs. Thus, extracting locking contraints in OS kernel code can be quite time-consuming and inaccurate.

C2: Detecting locking cycles. Due to the large and complex code base of the OS kernel, there are substantial target code paths containing distinct locking constraints. Thus, when detecting locking cycles, simply comparing between each two target code paths with their locking constraints in a recursive way is quite inefficient.

C3: Dropping false bugs. On the one hand, without validating the feasibility of code paths, static analysis can extract many infeasible target code paths and thus report many false bugs. On the other hand, each deadlock involves two or more target code paths that should be able to concurrently executed. Thus, without validating the concurrency of these target code paths, static analysis can report many false bugs whose target code paths cannot be concurrently executed.

To solve the above challenges, we propose three key techniques. For *C1*, we propose a *summary-based lock-usage analysis* to efficiently extract the code paths containing distinct locking constraints from kernel code. For *C2*, we propose a *reachability-based comparison method* to efficiently detect locking cycles from locking constraints. For *C3*, we propose a *two-dimensional filtering strategy* to effectively drop false positives by validating code-path feasibility and concurrency. We will introduce these techniques as follows.

3.1 Summary-Based Lock-Usage Analysis

Our summary-based lock-usage analysis has two basic stages: (S1) performing a dataflow analysis to collect target code paths containing distinct lock-acquire/release operations; and then (S2) performing a static lockset analysis to compute locking constraints for each target code path.

S1: Collecting target code paths. In this stage, the dataflow analysis has some properties: 1) this analysis is flow-sensitive and inter-procedural with the consideration of alias relationships, which can improve the accuracy; 2) this analysis uses function summaries to reduce repeated analysis, which can improve the efficiency; 3) this analysis drops the target code paths containing repeated lock-acquire/release operations, which can reduce repeated comparison in locking-cycle detection; 4) this analysis uses a light-weight method to validate the feasibility of each analyzed code path, which can reduce false positives of deadlock detection. This dataflow analysis traverses the code paths in the analyzed function *func*.

Figure 5 shows the main procedure of this dataflow analysis, which is represented as *DataFlowAnalysis*. It creates a function summary *func_sum*, which stores basic information about *func* (like function name and function-definition location) and the target code paths in *func*. This function summary is initialized with no target code path (line 1). This analysis handles each code path *code_path* in *func* with three steps (lines 2-30). First, it creates a data structure *tar_path* to collect the lock-acquire/release function calls and analyzed basic blocks in *code_path* (line 3), and then checks each function call *call* in *code_path* (lines 4-26). If *call* is used to acquire

```
DataFlowAnalysis(func)
Input: func – the analyzed function
Output: func_sum – function summary storing basic information about func
                    and the target code paths in func
 1: func_sum->tar_path_set := ø;
 2: foreach code_path in GetCodePathSet(func) do
 3:   tar_path :=CreateTargetCodePath(code_path);
 4:   foreach call in GetCallSetInPath(code_path) do
 5:     called_func :=GetCalledFunction(call);
 6:     if CheckLockFunction(called_func) then
 7:       AddLockVector(call, tar_path->lock_vec);
 8:     else
 9:       // Use function summary to reduce repeated analysis
10:       called_func_sum :=FindFuncSummary(called_func);
11:       if called_func_sum == NULL then
12:         called_func_sum :=DataFlowAnalysis(called_func);
13:       end if
14:       // Top-down analysis of all target code paths in the callee
15:       called_tar_path_set :=called_func_sum->tar_path_set;
16:       foreach called_tar_path in called_tar_path_set do
17:         tar_path_tmp := SplicePathInfo(tar_path, called_tar_path);
18:         if LightPathCheck(tar_path_tmp) == TRUE then
19:           AddPathSet(tar_path_tmp, func_sum->tar_path_set);
20:         end if
21:       end foreach
22:       // Bottom-up analysis of one target code path selected in the callee
23:       rand_tar_path :=RandomSelect(called_tar_path_set);
24:       tar_path := SplicePathInfo(tar_path, rand_tar_path);
25:     end if
26:   end foreach
27:   if LightPathCheck(tar_path) == TRUE then
28:     AddPathSet(tar_path, func_sum->tar_path_set);
29:   end if
30: end foreach
31: DropRepeatTargetCodePath(func_sum->tar_path_set);
32: return func_sum;
```

Figure 5: Dataflow analysis of collecting target code paths.

or release a lock, it is added into the lock-operation vector *lock_vec* of *tar_path* (line 7); otherwise this analysis handles its called function *called_func*. If *called_func* has been already analyzed, its function summary is gotten and stored as *called_func_sum* (line 10); otherwise, *DataFlowAnalysis* is recursively used to compute its function summary as *called_func_sum* (line 12). Then, from *called_func_sum*, this analysis gets and handles the target code paths in *called_func* (lines 15-21) in a top-down manner. For each such target code path *called_tar_path*, this analysis splices it with *tar_path* to form a new and possible target code path *tar_path_tmp*. This analysis uses a light-weight method (will be explained in Section 3.3) to validate the code-path feasibility of *tar_path_tmp*; if the code path is feasible, *tar_path_tmp* is considered as a possibly real target code path in *func* and added into the function summary *func_sum->tar_path_set* (lines 18-20). To avoid the explosion of bottom-up code paths from the callee function *called_func*, this analysis randomly selects one of the target code paths in this function and splices it into *tar_path* (lines 23-24). Before the code path ends, this analysis uses the light-weight method again to validate the code-path feasibility of *tar_path*; if the code path is feasible, *tar_path* is considered as a possibly real target code path in *func* and added into the function summary *func_sum->tar_path_set* (lines 27-29). After handling each code path, this analysis checks *func_sum->tar_path_set* to drop the target code paths containing identical lock-operation vectors (line 31), which can reduce repeated comparison of target code paths in locking-cycle detection. Finally, this analysis returns the function summary *func_sum* (line 32), which can be used to analyze other functions that call *func*.

Besides the procedure shown in Figure 5, this dataflow analysis also performs an intra-procedural, flow-insensitive and Andersen-style alias analysis [1] to identify all variables aliased with the lock argument of each lock-acquire/release function call. The alias analysis can help to improve the accuracy of computing locking constraints in S2. Moreover, each function summary stores the information about arguments and global variables, and drops the information about local variables, which are never used outside the function.

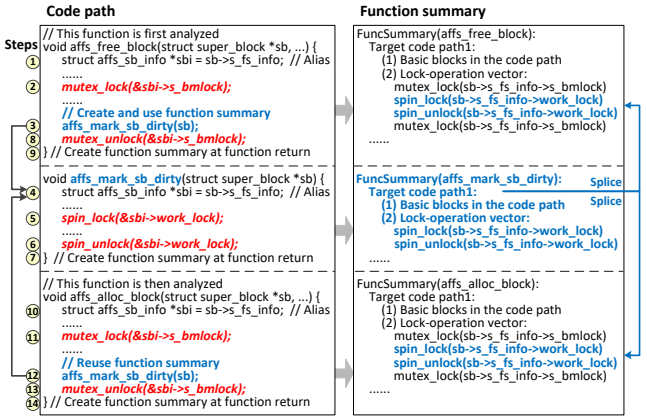

Figure 6: Example of summary-based dataflow analysis.

Example. We illustrate this dataflow analysis using the simplified code of the Linux *affs* filesystem in Figure 6. This figure shows three functions and partial code paths of them. According to the function position order, this analysis first analyzes the function `affs_free_block` and then `affs_alloc_block`, the analysis steps are represented as ⓝ. This dataflow analysis also considers the alias relationships at ①, ④ and ⑩. At ③, there is a function call to `affs_mark_sb_dirty`, so this dataflow analysis first handles this function, then creates its function summary, and finally splices the target code path of this function summary into the analyzed target code path of `affs_free_block`. At ⑫, the function `affs_mark_sb_dirty` is called again, so its function summary is reused, and the target code path of this function summary is spliced into the analyzed target code path of `affs_alloc_block`. By using the function summary, the analysis efficiency can be effectively improved.

Note that to avoid path explosion caused by bottom-up code paths of each callee function, this dataflow analysis uses a partial bottom-up analysis. Specifically, it randomly selects one of the target code paths in its function summary, and splices this path into the analyzed target code in the caller function, as shown at lines 23-24 in Figure 5. However, this method can miss other target code paths of the callee function, which can cause false negatives of deadlock detection. Even

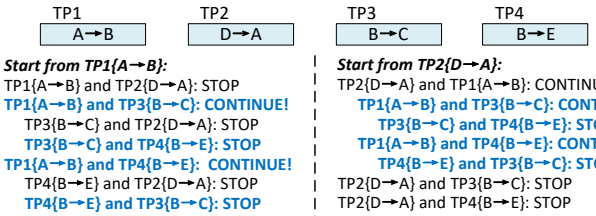

Figure 7: Example of the traditional comparison.

so, compared to RacerX [19] that only has top-down analysis without bottom-up analysis, this dataflow analysis is more accurate by using partial bottom-up analysis. In the future, we will implement a more complete and low-complexity bottom-up analysis, by referring to some existing approaches [39, 40].

S2: Computing locking constraints. This step uses a static lockset analysis to compute locking constraints in the target code paths collected in S1. This lockset analysis is similar to dynamic lockset analysis proposed in Eraser [47] for race detection, but in a static way. For each target code path, this lockset analysis maintains a lockset storing the held locks, and it handles lock-acquire and -release function calls.

When encountering a lock-acquire function call, this analysis first creates and adds related locking constraints in the analyzed target code path, according to the locks in the lockset and the acquired lock of this call; and then it adds this acquired lock into the lockset. For example, when this analysis handles the function call acquiring the lock X, if the lockset LS stores the held locks A and B, it first creates two locking constraints $A \rightarrow X$ and $B \rightarrow X$, then adds these locking constraints into the analyzed target code path, and finally adds X into LS. When encountering a lock-release function call, this analysis looks for and drops the involved lock in the lockset.

3.2 Reachability-Based Comparison Method

After extracting target code paths, we need to compare them to detect locking cycles as possible deadlocks. During comparison, we use a field-based analysis to identify the same locks in different code paths, if the lock variables' data structure types and fields are identical, which is similar to RacerX [19] and DCUAF [2]. Moreover, because a locking cycle can involve multiple target code paths (like the example deadlock in Figure 2(b)), the traditional method (used by existing static approaches like RacerX [19]) starts the comparison from each locking constraint in each target code path, and then recursively compares between each two target code paths with their locking constraints. Specifically, this method compares the current locking constraint with each locking contraint of each unhandled target code path. If they are matched, the current locking constraint is replaced with the matched locking constraint, and the comparison continues; if they are not matched, the comparison selects other target code paths. If all target code paths have been handled, the comparison stops. Once a locking cycle is found, this method reports a deadlock.

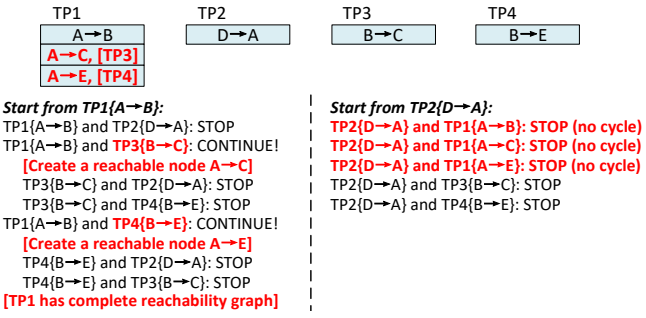

Figure 8: Example of our reachability-based comparison.

Example with the traditional comparison. We illustrate this traditional method using an example in Figure 7, containing four target code paths (*TP1*, *TP2*, *TP3* and *TP4*), each of which contains one locking constraint. This method first starts the comparison from the locking constraint $A \rightarrow B$ in *TP1* (namely *TP1{A→B}*), and then starts the comparison from the locking constraint $D \rightarrow A$ in *TP2* (namely *TP2{D→A}*). The detailed comparison steps are also shown in the figure.

In Figure 7, we find that when starting the comparison from *TP2{D→A}*, some steps (marked in blue and bold font) perform repeated comparison that has been done when starting from *TP1{A→B}*. In fact, such repeated comparison of locking constraints are common in locking cycle detection, because many code paths handle the same locks but have different locking orders, leading to different locking constraints. Thus, if such repeated comparison can be reduced, the locking cycle detection can be much more efficient.

Based on this idea, we propose a reachability-based method to efficiently compare locking constraints for locking-cycle detection. During comparison, this method maintains a constraint reachability graph for the target code path that the comparison starts from. This reachability graph contains some reachable nodes, each of which indicates an *indirect* locking constraint used for subsequent comparison:

$$\bigwedge_{i=1}^{n}(TP_i\{A_i \rightarrow A_{i+1}\}) \Rightarrow TP_{indirect}\{A_1 \rightarrow A_{n+1}, TP_{set}\}$$
$$TP_{set} = \{TP_1, TP_2, ..., TP_n\}$$

This indirect locking constraint is added in the handled target code path. When our method finishes the comparison starting from all the locking constraints in this target code path, its reachability graph is completely built. The indirect locking constraints in this reachability graph are used to reduce repeated comparison involving the handled target code path. Specifically, if any locking constraint (direct or indirect) in this target code path is matched, the comparison stops and checks whether there is a locking cycle. Note that our method assumes a target code path is never concurrently executed with itself, because static analysis has insufficient information to infer whether a code path can be concurrently executed with itself. This assumption is also followed by existing static approaches, such as RacerX [19] and DCUAF [2].

Example with our reachability-based comparison. To illustrate our method, we still use the example in Figure 7. The key steps performed by our method are marked in red and bold font. Our method still first starts the comparison from the locking constraint *TP1{A→B}*. Because *TP1{A→B}* matches *TP3{B→C}* and *TP4{B→E}*, our method creates two indirect locking constraints *TP1{A→C, [TP3]}* and *TP1{A→E, [TP4]}* and adds them in the target code path *TP1*. After *TP1* is handled, its reachability graph is completely built. When our method starts the comparison from the locking constraint *TP2{D→A}*, the three locking constraints (including two indirect ones) in *TP1* are matched. Because the reachability graph of *TP1* is complete, the comparison stops when these locking constraints are handled, which can avoid the repeated steps performed by the traditional comparison method in Figure 7. In this way, our method can effectively reduce the time usage of locking-cycle detection.

Besides, for each indirect locking constraint, our method also stores the related original locking constaints and target code paths. During comparison, if a locking cycle is found as a possible deadlock, our method can conveniently recover the information about the involved locks and their code paths, which is used for false-positive filtering in Section 3.3.

3.3 Two-Dimensional Filtering Strategy

For a possible deadlock, our strategy checks whether it is a false positive, in two dimensions, namely validating the feasibility and concurrency of its target code paths.

D1: Validating code-path feasibility. For a given code path, we can use an SMT solver to validate the satisfiability of all the branch conditions and variable accesses (including read and write operations) in this code path. For deadlock detection, there are two possible stages where code-path validation can be performed: (S1) during the dataflow analysis extracts target code paths in Section 3.1, we can validate the feasibility of each extracted target code path; (S2) after locking-cycle detection in Section 3.2, we can validate the feasibility of the target code paths for each possible deadlock.

In S1, because the dataflow analysis needs to handle substantial code paths, if we perform complete and accurate validation of these code paths, the time cost will be quite high. In S2, we believe that the code paths of possible deadlocks should occupy a very small proportion of all the target code paths extracted in the dataflow analysis, and thus it is acceptable to perform complete and accurate validation of these code paths. An alternative way is to just perform code-path validation in S2. However, without the validation in S1, lots of infeasible target code paths will be extracted for locking-cycle detection, which can also introduce high time cost.

Based on the above consideration, our strategy uses a staged and balanced way. In S1, our strategy uses a simple and light-weight path-condition checking method to efficiently check the extracted target code paths. This method checks only branch conditions in each code path, without handling variable accesses outside branch conditions. Thus, this method is fast but relatively inaccurate, and it can quickly drop many target code paths that are obviously infeasible. Figure 9(a) shows an example target code path that can be dropped by this method. In S2, our strategy uses a complete and heavy-weight path-condition checking method to accurately check the code paths of possible deadlocks. This method checks both branch conditions and variable accesses in each code path. Thus, this method is relatively slow but accurate, and it can effectively drop false deadlocks involving complex path conditions.

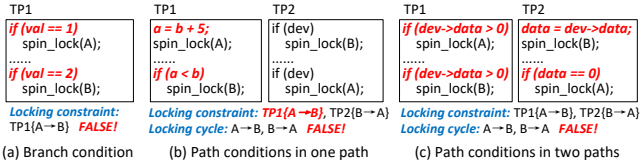

Figure 9: Examples of code-path feasibility validation.

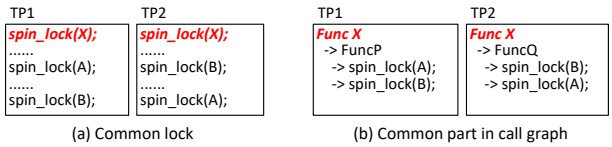

Figure 10: Examples of code-path concurrency checking.

In fact, because a deadlock contains two or more code paths that are interleaved in concurrent execution, these code paths may access some shared variables that should have identical values. Thus, for each possible deadlock, the heavy-weight method in S2 performs code-path validation in two ways. First, for each code path of this deadlock, the method validates its feasibility; if any code path is identified to be infeasible by an SMT solver, this deadlock is considered to be false and dropped. Second, the method extracts shared variables having identical data structure types and fields in each two code paths, then identifies the variable accesess and branch conditions that are related to these shared variables in the code paths, and finally translates the identified operations into SMT constraints of an SMT solver. If these SMT constraints are computed to be unsatisfiable, this deadlock is considered to be false and dropped. Figure 9(b) and Figure 9(c) show two example false deadlocks that can be dropped in these two ways, respectively.

D2: Checking code-path concurrency. For a deadlock, its target code paths should be able to be concurrently executed; otherwise, this deadlock is false. For each possible deadlock, our strategy checks the concurrency of its target code paths in two ways. First, our strategy checks whether there is a common lock acquired before the involved lock-acquire operations in any two of the target code paths. If so, these code paths cannot be concurrently executed, so this possible deadlock is considered to be false and dropped. Second, our strategy extracts the call graph of each target code path, and checks whether any two of these call graphs have common parts. If so,

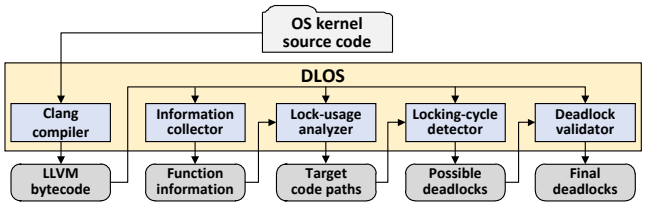

Figure 11: DLOS architecture.

it indicates that the related two code paths may be sequentially executed at different time points of the same thread, so this possible deadlock is considered to be false and dropped. Figure 10(a) and Figure 10(b) show two example false deadlocks that can be dropped in these two ways, respectively.

4 DLOS Approach

Based on the three key techniques in Section 3, we design a practical static approach named DLOS, to detect deadlocks in OS kernels. We have implemented DLOS with Clang [13] and Z3 [54]. DLOS automatically performs static analysis on the LLVM bytecode files of the OS kernel. Figure 11 shows the architecture of DLOS, which has four phases:

P1: Source-code compilation. The *Clang compiler* compiles the kernel source files into LLVM bytecode files, and then the *information collector* handles each function in LLVM bytecode to collect the function's information (including function name, function-definition position, etc.). The information is used for inter-procedural analysis across source files.

P2: Locking-constraint extraction. The *lock-usage analyzer* uses our summary-based lock-usage analysis to handle LLVM bytecode files. This analysis starts at the entry of each function in the kernel code, to extract target code paths containing distinct locking constraints.

P3: Locking-cycle detection. The *locking-cycle detector* uses our reachability-based comparison method to check the extracted target code paths with locking constraints, and detects locking cycles as possible deadlocks. We observe that a kernel module often acquires private locks that are not accessible for other kernel modules, and thus the detector focuses on checking target code paths in the same kernel module.

P4: False-positive filtering. The *deadlock validator* uses our two-dimensional filtering strategy to check possible deadlocks and drop false positives. Besides, two possible deadlocks may have identical problematic locking operations but differ in code paths. To drop such repeated bugs, for a new possible deadlock, the validator checks whether it has the same problematic locking operations with any already detected deadlock; if so, it is considered to be repeated and dropped.

Implementation details. DLOS performs lock-usage analysis and path validation from the entry of each function in OS code, so it can handle different execution contexts like interrupt handling. However, DLOS does not handle function-pointer calls at present, so it cannot detect deadlocks across kernel modules connected by function pointers. Besides, DLOS cannot analyze RCU locks, as RCU lock-acquiring/release functions (like `rcu_read_lock` and `rcu_read_unlock`) have no argument. Finally, to accelerate deadlock detection, DLOS can support the parallelism of handling multiple kernel modules using multi-thread execution.

5 Evaluation

To validate the effectiveness of DLOS, we evaluate it on the Linux kernel. To cover different kernel versions, we select an old version 4.9 and a recent version 5.10. Table 1 shows the basic information about these kernel versions, and source code lines are counted by CLOC [14]. We run the experiments on a regular x86-64 PC with eight Intel i7-3770@3.40G CPUs and 16GB memory. We use the kernel configuration *allyesconfig* to enable all kernel code for the x86-64 architecture.

Description	Linux 4.9	Linux 5.10
Release time	December 2016	December 2020
Source files (.c)	23.7K	29.4K
Source code lines (.c)	11.4M	14.7M

Table 1: Basic information about the checked OS kernels.

5.1 Bug Detection

We configure DLOS with common lock-acquiring/release functions (like `spin_lock` and `spin_unlock`) according to the Linux kernel documents [34], and then run DLOS to automatically check the kernel source code. We manually check all the deadlocks found by DLOS to identify real bugs. Table 2 shows the results, and we have the following findings:

Code analysis. DLOS can scale to large code bases of OS kernels. Specifically, it analyzes 8.5M and 11.7M source code lines in 14.5K and 19.9K source files in Linux 4.9 and 5.10, respectively, within 7 hours. The remaining 2.9M and 3.0M source code lines in 9.2K and 9.5K source files are not analyzed, because they are not enabled by *allyesconfig* for the x86-64 architecture.

Efficiency improvement. DLOS improves the analysis efficiency from two aspects:

First, when extracting target code paths containing locking constraints, our lock-usage analysis uses function summaries to reduce repeated code analysis in the same functions. Specifically, around 93% of the times DLOS handles a function call is able to reuse an existing function summary, without the need of analyzing the function's definition again.

Second, when detecting locking cycles, our reachability-based comparison method uses constraint reachability graphs to reduce repeated comparison of locking constraints. Specifically, with 196K and 222K indirect locking constraints created by our method, 851K and 946K times of repeated comparison are reduced in Linux 4.9 and 5.10, respectively.

Description		Linux 4.9	Linux 5.10
Lock-usage analysis	Source files (analyzed/all)	14.5K/23.7K	19.9K / 29.4K
	Source code lines (analyzed/all)	8.5M/11.4M	11.7M/14.7M
	Times of handling functions	4,102K	5,032K
	Times of reusing function summaries	3,816K	4,682K
	Extracted distinct target code paths	102K	117K
	Extracted locking constraints	323K	439K
Locking-cycle detection	Created indirect locking constraints	196K	222K
	Times of reducing comparison	851K	946K
	Possible deadlocks	465	539
False-positive filtering	Dropped infeasible target code paths	464K	524K
	False bugs due to infeasible paths	220	258
	False bugs due to common locks	78	94
	False bugs due to call graphs	101	122
	Total false bugs	419	474
Deadlock	Found bugs	46	65
	Real bugs	39	54
Time usage	Lock-usage analysis	265m	294m
	Locking-cycle detection	85m	96m
	False-positive filtering	22m	28m
	Total time	372m	418m

Table 2: Deadlock-detection results.

False-positive dropping. DLOS uses our two-dimensional filtering strategy to drop false positives from three aspects:

First, when extracting target code paths containing locking constraints, our strategy uses a simple and light-weight code-path validation method to drop 464K and 524K infeasible target code paths in Linux 4.9 and 5.10, respectively. In addition, by dropping these target code paths, the related unnecessary locking-constraint comparison can be avoided in locking-cycle detection, which also improves the efficiency of deadlock detection.

Second, after locking-cycle detection reports possible deadlocks, our strategy uses a complete and heavy-weight code-path validation method to drop 220 and 258 false bugs in Linux 4.9 and 5.10, respectively. Indeed, these false bugs' code paths are failed to be dropped in our lock-usage analysis, because the light-weight code-path validation method used in this analysis is efficient but relatively inaccurate. To improve accuracy, the heavy-weight code-path validation method completely checks both branch conditions and variable accesses in the code paths of each possible deadlock, and thus it successfully drops these false bugs after locking cycle detection.

Finally, our strategy checks the concurrency of possible deadlocks, and drops 179 and 216 false bugs in Linux 4.9 and 5.10, respectively, because their target code paths are considered to be non-concurrent. Specifically, 78 and 94 bugs are dropped due to holding a common lock in target code paths; 101 and 122 bugs are dropped due to containing common parts in the call graphs of target code paths.

Deadlock finding. DLOS reports 46 and 65 deadlocks in Linux 4.9 and 5.10, respectively. We spent eight hours on checking all these 111 reported deadlocks. We identify 39 and 54 deadlocks are real in Linux 4.9 and 5.10, respectively. 21 real deadlocks in Linux 4.9 have been fixed in Linux 5.10, including the deadlock in the *btrfs* filesystem shown in Figure 4. Thus, DLOS can find known deadlocks. Moreover, we have reported the 54 real deadlocks in Linux 5.10 to Linux kernel developers, and 31 of them have been confirmed. We are still waiting for the reply of the remaining ones. Thus, DLOS can find new deadlocks.

We infer that these real deadlocks are missed by Lockdep in extensive kernel testing, because their thread interleavings are infrequent to occur, and constructing workloads to cover these thread interleavings is difficult. Thus, DLOS can indeed find many deadlocks that are hard to find in runtime testing.

Besides, we believe that DLOS is helpful to deadlock reproduction, because it produces the detailed code paths of the found deadlocks. With these code paths, time delays can be strategically injected and carefully controlled, to cover specific thread interleavings and reproduce the found deadlocks. We have manually performed this way for several deadlocks in kernel modules that we can run, including the two deadlocks in Figure 4 and Figure 12(a), and these deadlocks can be successfully reproduced at runtime.

Deadlock details. Among the 93 real deadlocks in Linux 4.9 and 5.10, 78 (33 in Linux 4.9 and 45 in Linux 5.10) occur in device drivers, and 15 (6 in Linux 4.9 and 9 in Linux 5.10) occur in filesystems. This result indicates that device drivers remain a significant source of OS bugs [49]. Besides, for 86 deadlocks (35 in Linux 4.9 and 51 in Linux 5.10), DLOS reports two code paths for each of them, indicating it is caused by two locks in two threads; for the remaining 7 deadlocks, DLOS reports three code paths for each of them, indicating it is caused by three locks in three threads.

5.2 False Positives and Negatives

False positives. DLOS reports 7 and 11 false bugs in Linux 4.9 and 5.10, resulting the false positives rates of 15% and 17%, respectively. By manually checking these false bugs, we find that they are reported for three main reasons:

First, the field-based analysis in locking-cycle detection can make mistakes when identifying the same locks in different code paths. This analysis identifies the same locks if the locks variables have the same data structure types and fields; but two different lock variables can also have the same data structure types and fields, and their data structure variables are different. This analysis cannot handle such cases at present. This reason causes DLOS to report 3 and 5 false bugs in Linux 4.9 and 5.10, respectively.

Second, although DLOS uses Z3 to validate path feasibility, it can still make mistakes when handling some complex cases, such as complicated arithmetic conditions and data dependence across multiple functions. This reason causes DLOS to report 2 and 3 false bugs in Linux 4.9 and 5.10, respectively.

Finally, the alias analysis in our lock-usage analysis is intra-procedural and flow-insensitive, and thus can identify wrong alias relationships across function calls, causing mistakes in locking-constraint extraction. This reason causes DLOS to report 2 and 3 false bugs in Linux 4.9 and 5.10, respectively.

False negatives. DLOS may still miss some real deadlocks for three main reasons:

First, our lock-usage analysis performs incomplete bottom-up analysis of each callee function, to avoid path explosion of inter-procedural analysis. Specifically, it randomly selects one of the target code paths in the callee function, and splices it into the analyzed target code in the caller function. Although the other target code paths in the callee function are handled in top-down analysis, they are neglected in bottom-up analysis, causing some locking constraints in the target code paths of the caller function to be missed.

Second, DLOS does not analyze function-pointer calls, and thus it cannot build complete call graphs for inter-procedural analysis. As a result, DLOS may miss real deadlocks involving the code that is reached through function-pointer calls.

Finally, DLOS considers that a target code path is never concurrently executed with itself. Indeed, to reduce false positives, DLOS validates two code paths' concurrency by checking their common locks and call graphs. However, this validation is infeasible for two identical code paths, and thus DLOS does not detect deadlocks occuring in the same target code path of different execution contexts.

5.3 Case Studies of the Found Deadlocks

Figure 12 shows two deadlocks found by DLOS in Linux 5.10, and they have been confirmed by Linux kernel developers.

Deadlock in SysRq command handling for filesystems. In Figure 12(a), when the function `do_thaw_all_callback` is executed on the code path *P1*, it first acquires the read-write semaphore `sb->s_umount` and then the mutex lock `bdev->bd_fsfreeze_mutex`; when the function `freeze_bdev` is executed on the code path *P2*, it first acquires the mutex lock `bdev->bd_fsfreeze_mutex` and then the read-write semaphore `sb->s_umount`. During SysRq commands [50] are handled for filesystems, the functions `do_thaw_all_callback` and `freeze_bdev` can be concurrently executed, and thus an ABBA deadlock can occur.

Deadlock in the LPFC SCSI driver. In Figure 12(b), when the function `lpfc_nvmet_unsol_fcp_issue_abort` is executed on the code path *P1*, it acquires the spinlocks `ctxp->ctxlock` and then `phba->sli4_hba.abts_nvmet_buf_list_lock`; when the function `lpfc_sli4_nvmet_xri_aborted` is executed on the code path *P2*, it acquires the spinlocks `phba->sli4_hba.abts_nvmet_buf_list_lock` and then `ctxp->ctxlock`. During driver execution, the functions `lpfc_nvmet_unsol_fcp_issue_abort` and `lpfc_sli4_nvmet_xri_aborted` can be concurrently executed, and thus an ABBA deadlock can occur.

From the feedback of kernel developers, the confirmed deadlocks found by DLOS require infrequent and special test cases to find at runtime, which indicates that DLOS is useful to detecting hard-to-trigger deadlocks via static analysis.

```
Code Path P1:
// FILE: linux-5.10/fs/super.c
do_thaw_all_callback
  -> down_write(&sb->s_umount) [Line 1028]
  -> emergency_thaw_bdev [Line 1030]
    -> thaw_bdev [526]
      -> mutex_lock(&bdev->bd_fsfreeze_mutex) [Line 734]
Code Path P2:
// FILE: linux-5.10/fs/block_dev.c
freeze_bdev
  -> mutex_lock(&bdev->bd_fsfreeze_mutex) [Line 556]
  -> freeze_super [Line 576]
    -> down_write(&sb->s_umount) [Line 517]
```

(a) Deadlock in SysRq command handling for filesystems

```
Code Path P1:
// FILE: linux-5.10/drivers/scsi/lpfc/lpfc_nvmet.c
lpfc_nvmet_unsol_fcp_issue_abort
  -> spin_lock_irqsave(&ctxp->ctxlock, flags) [Line 3502]
  -> spin_lock(&phba->sli4_hba.abts_nvmet_buf_list_lock) [Line 3504]
Code Path P2:
// FILE: linux-5.10/drivers/scsi/lpfc/lpfc_nvmet.c
lpfc_sli4_nvmet_xri_aborted
  -> spin_lock(&phba->sli4_hba.abts_nvmet_buf_list_lock) [Line 1787]
  -> spin_lock(&ctxp->ctxlock) [1794]
```

(b) Deadlock in the LPFC SCSI driver

Figure 12: Two real deadlocks found by DLOS in Linux 5.10.

5.4 Comparison Experiment

We aim to experimentally compare to RacerX [19], which is the sole existing static approach to systematically detect deadlocks in OS kernels. However, RacerX is not open-source, and Linux kernel 2.5.62 checked in its paper is too old to be normally compiled by Clang. Thus, we have to try our best to implement a RacerX-like tool according to its paper.

RacerX [19] performs code analysis with summary caches, which seems similar to function summaries used in our lock-usage analysis. However, RacerX lacks the other two key techniques used in DLOS, namely the reachability-based comparison method to improve the efficiency of locking-cycle detection, and the two-dimensional filtering strategy to drop false positives. To validate the value of these two techniques in comparison, we implement three tools by modifying DLOS: (1) $DLOS_{!reach}$ that uses the traditional comparison method in RacerX for locking-cycle detection to replace the reachability-based comparison method in DLOS; (2) $DLOS_{!filter}$ that removes the two-dimensional filtering strategy in DLOS; (3) *RacerX-like* that both uses the traditional comparison method for locking-cycle detection and removes the two-dimensional filtering strategy in DLOS.

We run these three tools to check the whole Linux 5.10 code, but the $DLOS_{!reach}$ and RacerX-like tools run for over 60 hours, without finishing their detection. Thus, for more clear comparison, we select six kernel modules in Linux 5.10, and run these tools and DLOS to check the source code of these kernel modules. These six kernel modules include: two ones (*sb* and *lpfc*) that has real deadlocks found by DLOS, two ones (*fpga* and *ocfs2*) that has false deadlocks found by DLOS, and two ones (*jfs* and *bcache*) that has no deadlock found by DLOS. Table 3 shows the results, and we find that:

Description		DLOS$_{!reach}$	DLOS$_{!filter}$	RacerX-like	DLOS
sb	Found bugs (real/all)	6/6	6/14	6/14	6/6
	Time usage	30s	14s	27s	16s
lpfc	Found bugs (real/all)	7/7	7/25	7/25	7/7
	Time usage	524s	162s	501s	181s
fpga	Found bugs (real/all)	0/2	0/5	0/5	0/2
	Time usage	21s	9s	18s	11s
ocfs2	Found bugs (real/all)	0/2	0/10	0/10	0/2
	Time usage	936s	214s	892s	253s
jfs	Found bugs (real/all)	0/0	0/0	0/0	0/0
	Time usage	305s	101s	280s	122s
bcache	Found bugs (real/all)	0/0	0/3	0/3	0/0
	Time usage	78s	28s	71s	33s

Table 3: Comparison results of six Linux kernel modules.

First, the DLOS$_{!reach}$ tool achieves the same accuracy with DLOS, but it spends more time on locking-cycle detection. Thus, our reachability-based comparison method is more efficient than the traditional comparison method in RacerX, when performing locking-cycle detection.

Second, the DLOS$_{!filter}$ tool reports many more false bugs than DLOS, though it finds the real deadlocks found by DLOS. Thus, our two-dimensional filtering strategy is useful to dropping false positives in deadlock detection. Moreover, we observe that the DLOS$_{!filter}$ tool spends less time than DLOS. Indeed, without validating the feasibility or concurrency of target code paths, the DLOS$_{!filter}$ tool can decrease time usage; but this tool extracts many infeasible target code paths for locking-constraint comparison, which also increases the time usage of locking-cycle detection. As a whole, the decreased time usage is more than the increased time usage in the experiment, and thus the DLOS$_{!filter}$ tool has less time usage than DLOS.

Finally, the RacerX-like tool spends less time than the DLOS$_{!reach}$ tool, because it does not validate the feasibility or concurrency of target code paths, but causing more false bugs to be reported. The RacerX-like tool spends more time than the DLOS$_{!filter}$ tool, because it detects locking cycles with the traditional comparison method, which is less efficient than our reachability-based comparison method; but it achieves the same accuracy with the DLOS$_{!filter}$ tool, because neither of them drops false positives. Compared to the RacerX-like tool, DLOS achieves better accuracy in deadlock detection with less time usage.

6 Discussion

Interleaving model. DLOS identifies each target code path from the entry of each function in OS code, and then it considers that two different target code paths identified by our lock-usage analysis can be concurrently executed. To reduce false positives, DLOS validates their concurrency by checking common locks and call graphs using our two-dimensional filtering strategy. As this strategy is infeasible in handling the case that a target code path is concurrently executed with itself, DLOS cannot detect deadlocks occurring in this case.

Detecting deadlocks in other OS kernels. Besides the Linux kernel, DLOS can also check other OS kernels to detect their deadlocks. However, doing so has some practical difficulties. For example, some APIs used in DLOS have different usages between these OSes and Linux, and these OSes have different processes of kernel-code compilation from Linux. At present, we have preliminarily run DLOS in NetBSD to check its kernel source code, and found one real deadlock in the *sysmon* kernel module without false positive. This deadlock has been confirmed by NetBSD kernel developers.

Detecting deadlocks involving waiting queues. Besides the deadlocks caused by locking cycles, incorrect operations on waiting queues can also cause deadlocks in OS kernels. For example, one thread waits for the event E_1 and then triggers the event E_2, while the other concurrent thread waits for the event E_2 and then triggers the event E_1, so a deadlock can occur for these two threads. In kernel code, waiting queues and locks can be used together to cause deadlocks, which are more difficult to detect. At present, no static approach (including RacerX) can detect such deadlocks, and thus we plan to extend DLOS to detecting them.

Detecting other locking issues. We believe that DLOS can be extended to detecting other locking issues, such as double locks and using sleep-able locks while holding spinlocks. Indeed, computing locksets and validating code-path feasibility are two important steps in detecting locking issues, and our lock-usage analysis and filtering strategy can effectively perform these steps, respectively.

Limitations and future works. DLOS can be strengthened in some aspects. First, DLOS does not handle function-pointer calls in its lock-usage analysis, and thus it may miss deadlocks involving the code that is reached through function-pointer calls, especially the deadlocks across kernel modules connected by function pointers. To relieve this limitation, we plan to apply existing function-pointer analysis [3, 36] in DLOS to detect more deadlocks and reduce false negatives. Second, to reduce the complexity of analyzing loops and recursive calls, DLOS unrolls each loop and recursive call just once, causing soundness loss in static analysis. Such soundness loss can introduce both false positives and negatives, when DLOS analyzes the code involving loops and recursive calls. To relieve this limitation, we plan to adapt existing loop-oriented analysis [35, 48] in DLOS to soundly handle loops and recursive calls. Thirdly, DLOS does not handle some special cases at present, such as RCU locks, memory barriers, assembly instructions and concurrent execution of the same code path, which may also cause false positives and negatives in deadlock detection. To relieve this limitation, we plan to consider these special cases in our static analysis, to further improve analysis accuracy. Finally, we plan to port DLOS to detecting deadlocks in other OS kernels, and to extend DLOS to detecting deadlocks involving waiting queues as well as other locking issues.

7 Related Work

7.1 Dynamic Analysis of Deadlocks

Many approaches [5, 9–11, 15, 21, 27, 28, 31, 33, 44, 45, 55] dynamically monitor thread execution and lock-related operations to detect locking cycles. Most of them are used for user-level applications. For example, Pulse [31] is an operating system mechanism to detect deadlocks in applications. It periodically identifies long-sleeping application processes and the events they are waiting for, then uses high-level speculative execution to a general resource graph about each identified application process, and finally detects cycles in the graph as deadlocks. UnDead [55] is an efficient dynamic approach for deadlock detection. It uses several techniques to reduce runtime overhead, such as only recording unique lock dependencies (identical to locking constraints in this paper) for every thread during the execution and dropping unnecessary information in runtime recording.

Lockdep [33] is a kernel lock-usage validator, which can find different kinds of lock-related bugs, such as double locks and deadlocks. Lockdep performs runtime monitoring and checking based on the granularity of lock class, which describes a group of locks that are logically the same with respect to locking rules. Specifically, Lockdep dynamically tracks the state of each lock class and checks the dependencies between different lock classes. If any state or dependency is incorrect when lock-related operations are performed, Lockdep will report related bugs at runtime.

By using exact runtime information about thread execution and lock-related operations, dynamic analysis approaches can effectively reduce false positives in deadlock detection. However, dynamic analysis requires substantial test cases to achieve high testing coverage and reduce false negatives, and it also introduces runtime overhead for the tested programs.

7.2 Static Analysis of Deadlocks

Some approaches [30, 41, 42, 46, 52] use static analysis to detect deadlocks in user-level applications, without actually running the applications. Naik et al. [30] design a sound static approach to check deadlocks in C programs. For the checked program, this approach performs context-sensitive and thread-sensitive analysis on its inter-procedural control flows, based on abstract interpretation. During the analysis, this approach checks lock-related operations to extract lock dependencies and detect locking cycles as possible deadlocks. This approach also uses a non-concurrency analysis to drop false positives, by checking common locks and thread-creation/joining operations. Santhiar et al. [46] design a static approach to detect deadlocks in asynchronous C# programs. This approach uses a new representation of the mixed synchronous and asynchronous control flows, and constructs a deadlock detection graph based on this representation. However, OS kernels and user-level applications have different concurrency models (described in Section 2.2), and thus these approaches are ineffective in detecting deadlocks in OS kernels.

To our knowledge, RacerX [19] is the sole existing static analysis approach to systematically detect deadlocks in OS kernels. However, it has a high false positive rate of 46%, due to neglecting the feasibility and concurrency of code paths; and its locking-cycle detection method is simple and inefficient. Compared to RacerX, DLOS checks the feasibility and concurrency of code paths to achieve better accuracy, and uses a reachability-based comparison method in locking-cycle detection to achieve higher efficiency. Besides, we also note that Breuer et al. [6–8] have several works that focus on detecting deadlocks caused by sleeping whiling holding spinlocks. As these works have no systematic technique of detecting deadlocks caused by locking cycles, we do not particularly introduce and compare to these works in this paper.

7.3 Detection of Kernel Concurrency Bugs

Besides deadlocks, OS kernels also suffer from other kinds of concurrency bugs, such as data races and atomicity violations. To detect these kernel concurrency bugs, some approaches [2, 12, 18, 47] use static or dynamic lockset analysis to track shared variables and lock-related operations, and some approaches [20, 25, 29] perform sampling to monitor concurrent memory accesses. Moreover, to actually cover infrequent thread interleavings and detect hard-to-find concurrency bugs, some approaches [22–24, 26, 53] perform random thread scheduling or coverage-guided thread-interleaving exploration in runtime testing. Though these approaches do not target deadlocks, some of their techniques (like lockset analysis) are useful for DLOS in deadlock detection.

8 Conclusion

Deadlocks in OS kernels are dangerous and hard-to-find. To detect these bugs, we design a practical static analysis approach named DLOS. It has three key techniques, including a summary-based lock-usage analysis to efficiently extract code paths containing distinct locking constraints, a reachability-based comparison method to efficiently detect locking cycles, and a two-dimensional filtering strategy to effectively drop false positives. In the evaluation, DLOS finds 54 real deadlocks in Linux 5.10, and 31 of them have been confirmed.

Acknowledgment

We thank our shepherd and anonymous reviewers for their helpful advice on the paper. We also thank Linux kernel developers, who gave useful feedback and advice to us. This work was supported by the National Natural Science Foundation of China under Project 62002195.

References

[1] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[2] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 255–268, 2019.

[3] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Effective detection of sleep-in-atomic-context bugs in the Linux kernel. *ACM Transactions on Computer Systems (TOCS)*, 36(4):1–30, 2020.

[4] Jia-Ju Bai, Yu-Ping Wang, and Shi-Min Hu. AutoPA: automatically generating active driver from original passive driver code. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, pages 288–299, 2018.

[5] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Proceedings of the 2005 Haifa Verification Conference*, pages 208–223, 2005.

[6] Peter T Breuer and Simon Pickin. Checking for deadlock, double-free and other abuses in the Linux kernel source code. In *Proceedings of the 2006 International Conference on Computational Science*, pages 765–772, 2006.

[7] Peter T Breuer, Simon Pickin, and Maria Larrondo Petrie. Detecting deadlock, double-free and other abuses in a million lines of Linux kernel source. In *Proceedings of the 30th NASA Software Engineering Workshop*, pages 223–233, 2006.

[8] Peter T Breuer and Marisol Garciá Valls. Static deadlock detection in the Linux kernel. In *Proceedings of the 9th Ada-Europe International Conference on Reliable Software Technologies*, pages 52–64, 2004.

[9] Yan Cai and WK Chan. MagicFuzzer: scalable deadlock detection for large-scale applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 606–616, 2012.

[10] Yan Cai and Qiong Lu. Dynamic testing for deadlocks via constraints. *IEEE Transactions on Software Engineering (TSE)*, 42(9):825–842, 2016.

[11] Yan Cai, Ruijie Meng, and Jens Palsberg. Low-overhead deadlock prediction. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 1298–1309, 2020.

[12] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, 2019.

[13] Clang: a LLVM-based compiler for C/C++ program. https://clang.llvm.org/.

[14] CLOC: count lines of code. https://cloc.sourceforge.net.

[15] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. Dynamic deadlock verification for general barrier synchronisation. In *Proceedings of the 20th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 150–160, 2015.

[16] Linux commit 57ba4cb85bff. https://github.com/torvalds/linux/commit/57ba4cb85bff.

[17] Linux commit 01d01caf19ff. https://github.com/torvalds/linux/commit/01d01caf19ff.

[18] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pages 166–177, 2015.

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

[20] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 151–162, 2010.

[21] Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 353–365, 2014.

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

[23] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the 28th International Symposium on Operating Systems Principles (SOSP)*, pages 66–83, 2021.

[24] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: finding kernel race bugs through fuzzing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pages 754–768, 2019.

[25] Yunyun Jiang, Yi Yang, Tian Xiao, Tianwei Sheng, and Wenguang Chen. DRDDR: a lightweight method to detect data races in Linux kernel. *Journal of Supercomputing*, 72(4):1645–1659, 2016.

[26] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[27] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the 18th International Symposium on Foundations of Software Engineering (FSE)*, pages 327–336, 2010.

[28] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, pages 110–120, 2009.

[29] KCSAN: concurrency sanitizer for the Linux kernel. https://github.com/google/ktsan/wiki/KCSAN.

[30] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for C/Pthreads. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 379–390, 2016.

[31] Tong Li, Carla Schlatter Ellis, Alvin R Lebeck, and Daniel J Sorin. Pulse: a dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, pages 31–44, 2005.

[32] LLVM compiler infrastructure. https://llvm.org/.

[33] Lockdep: runtime locking correctness validator in the Linux kernel. https://www.kernel.org/doc/html/latest/locking/lockdep-design.html.

[34] Linux kernel locking documents. https://www.kernel.org/doc/html/latest/locking/index.html.

[35] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization (CGO)*, pages 136–146, 2009.

[36] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 26th International Conference on Computer and Communications Security (CCS)*, pages 1867–1881, 2019.

[37] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th International Conference on File and Storage Technologies (FAST))*, pages 31–44, 2013.

[38] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, 2008.

[39] Ivan Matosevic and Tarek S Abdelrahman. Efficient bottom-up heap analysis for symbolic path-based data access summaries. In *Proceedings of the 2012 International Symposium on Code Generation and Optimization (CGO)*, pages 252–263, 2012.

[40] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 9th International Symposium on Foundations of Software Engineering (FSE)*, pages 554–564, 2013.

[41] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 386–396, 2009.

[42] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, pages 174–184, 2016.

[43] Leonid Ryzhyk, Yanjin Zhu, and Gernot Heiser. The case for active device drivers. In *Proceedings of the 1st Asia-Pacific Workshop on Systems (APSys)*, pages 25–30, 2010.

[44] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 473–489, 2014.

[45] Malavika Samak and Murali Krishna Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Proceedings of the 19th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 29–42, 2014.

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

[47] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[48] Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue. Loop-oriented pointer analysis for automatic simd vectorization. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):1–31, 2018.

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

[50] Linux kernel SysRq documents. https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html.

[51] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

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

[53] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: data race fuzzing for kernel file systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 1643–1660, 2020.

[54] Z3: a theorem prover. https://github.com/Z3Prover/z3.

[55] Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. Undead: detecting and preventing deadlocks in production software. In *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE)*, pages 729–740, 2017.