DLOS: Effective Static Detection of Deadlocks in OS Kernels

In USENIX ATC 2022

Jia-Ju Bai, Tuo Li, Shi-Min Hu

Tsinghua University

https://baijiaju.github.io/

Tsinghua University

Motivation

- Deadlocks in OS kernels
  - Caused by locking cycles in concurrent threads
  - Hard to find due to the non-determinism of kernel concurrency
  - Can cause performance degradation and even system hangs

Motivation

- Example
  - ABBA deadlock in Linux 4.9 *btrfs* filesystem
  - Lifetime: Jul. 2016 ~ Oct. 2020
  - Fixed by the commit 01d01caf19ff in Linux 5.9

Code Path P1:
```
// FILE: linux-4.9/fs/btrfs/volumes.c
btrfs_read_chunk_tree
   -> lock_chunks [Line 6803]
      -> mutex_lock(&root->fs_info->chunk_mutex) [Line 517]
   -> read_one_dev [Line 6833]                              A→B
      -> open_seed_devices [Line 6601]
         -> clone_fs_devices [Line 6558]
            -> mutex_lock(&orig->device_list_mutex) [Line 734]
```

Code Path P2:
```
// FILE: linux-4.9/fs/btrfs/volumes.c
btrfs_remove_chunk
   -> mutex_lock(&fs_devices->device_list_mutex) [Line 2844]
   -> lock_chunks [Line 2857]                               B→A
      -> mutex_lock(&root->fs_info->chunk_mutex) [Line 517]
```

State of the art

- Basic steps of deadlock detection
  - S1: Extracting locking constraints in concurrent threads/code paths

    $T\{A \rightarrow B\}$ means thread T acquires lock B when lock A is held
  - S2: Detecting locking cycles in concurrent threads/code paths

    $T_1\{A \rightarrow B\}$, $T_2\{B \rightarrow C\}$ and $T_3\{C \rightarrow A\}$ form a locking cycle in three threads

Thread T_1

```
spin_lock(A);
......
spin_lock(B);
```

Thread T_2

```
spin_lock(B);
......
spin_lock(A);
```

Locking constraint: $T_1\{A \rightarrow B\}$, $T_2\{B \rightarrow A\}$

Locking cycle: $A \rightarrow B$, $B \rightarrow A$ **Deadlock!**

Deadlock in two threads

Thread T_1

```
spin_lock(A);
......
spin_lock(B);
```

Thread T_2

```
spin_lock(B);
......
spin_lock(C);
```

Thread T_3

```
spin_lock(C);
......
spin_lock(A);
```

Locking constraint: $T_1\{A \rightarrow B\}$, $T_2\{B \rightarrow C\}$, $T_3\{C \rightarrow A\}$

Locking cycle: $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$ **Deadlock!**

Deadlock in three threads

State of the art

- Dynamic analysis
  - Most approaches are designed for user-level applications
  - Advantages: low false positives + support reproduction
  - Weakness: limited testing coverage + runtime overhead
- LockDep [1]
  - Widely-used kernel lock-usage runtime validator
  - Runtime monitoring and checking
  - Based on the granularity of lock class

[1] Lockdep. https://www.kernel.org/doc/html/latest/locking/lockdep-design.html

State of the art

- Static analysis
  - Most approaches are designed for user-level applications
  - Advantages: good detection coverage + easy to use
  - Weakness: high false positives + hard to reproduce
- RacerX [2]
  - ***Sole*** static approach of detecting kernel deadlocks
  - Flow-sensitive and inter-procedural analysis
  - 46% false positive rate in its evaluation

We focus on improving static analysis in kernel deadlock detection!

[2] Engler et al. RacerX: effective, static detection of race conditions and deadlocks. In SOSP 03.

Challenges of static kernel deadlock detection

- ***C1: Extracting locking constraints***
  - How to ensure both the accuracy and efficiency when analyzing large kernel code?
- ***C2: Detecting locking cycles***
  - How to reduce the time usage of comparing numerous locking constraints in lots of code paths?
- ***C3: Dropping false bugs***
  - How to effectively drop false positives with short time usage?

Key techniques

- ***C1: Extracting locking constraints***
  - ***T1: Summary-based lock-usage analysis*** to extract target code paths containing distinct locking constraints
- ***C2: Detecting locking cycles***
  - ***T2: Reachability-based comparison method*** to detect locking cycles from locking constraints
- ***C3: Dropping false bugs***
  - ***T3: Two-dimensional filtering strategy*** to drop false positives by validating code-path feasibility and concurrency

T1: Summary-based lock-usage analysis

- S1: Collecting target code paths
  - *Target code path* means a code path having lock-related operations
  - Flow-sensitive, field-sensitive and inter-procedural analysis
  - Andersen-style [3] alias analysis to identify aliased lock variables
  - Create and reuse function summaries to reduce repeated analysis
  - Drop target code paths having repeated lock-related operations

[3] Lars Ole Andersen. Program analysis and specialization for the C programming language. PhD thesis, 1994.

T1: Summary-based lock-usage analysis

- S1: Collecting target code paths
  - Example: Linux *affs* filesystem code

Code path

Steps

```
    // This function is first analyzed
    void affs_free_block(struct super_block *sb, ...) {
1       struct affs_sb_info *sbi = sb->s_fs_info;  // Alias
        ......
2       mutex_lock(&sbi->s_bmlock);
        ......
        // Create and use function summary
3       affs_mark_sb_dirty(sb);
8       mutex_unlock(&sbi->s_bmlock);
9   } // Create function summary at function return
    ------------------------------------------------------------
    void affs_mark_sb_dirty(struct super_block *sb) {
4       struct affs_sb_info *sbi = sb->s_fs_info;  // Alias
        ......
5       spin_lock(&sbi->work_lock);
        ......
6       spin_unlock(&sbi->work_lock);
7   }  // Create function summary at function return
    ------------------------------------------------------------
    // This function is then analyzed
    void affs_alloc_block(struct super_block *sb, ...) {
10      struct affs_sb_info *sbi = sb->s_fs_info;  // Alias
        ......
11      mutex_lock(&sbi->s_bmlock);
        ......
        // Reuse function summary
12      affs_mark_sb_dirty(sb);
13      mutex_unlock(&sbi->s_bmlock);
14  } // Create function summary at function return
```

Function summary

```
FuncSummary(affs_free_block):
  Target code path1:
    (1) Basic blocks in the code path
    (2) Lock-operation vector:
      mutex_lock(sb->s_fs_info->s_bmlock)
      spin_lock(sb->s_fs_info->work_lock)
      spin_unlock(sb->s_fs_info->work_lock)
      mutex_lock(sb->s_fs_info->s_bmlock)
  ......
------------------------------------------------------------
FuncSummary(affs_mark_sb_dirty):                 Splice
  Target code path1:
    (1) Basic blocks in the code path            Splice
    (2) Lock-operation vector:
      spin_lock(sb->s_fs_info->work_lock)
      spin_unlock(sb->s_fs_info->work_lock)
  ......
------------------------------------------------------------
FuncSummary(affs_alloc_block):
  Target code path1:
    (1) Basic blocks in the code path
    (2) Lock-operation vector:
      mutex_lock(sb->s_fs_info->s_bmlock)
      spin_lock(sb->s_fs_info->work_lock)
      spin_unlock(sb->s_fs_info->work_lock)
      mutex_lock(sb->s_fs_info->s_bmlock)
  ......
```

T1: Summary-based lock-usage analysis

- S2: Computing locking constraints
  - Static lockset analysis [4] for each target code path
  - Handle the cases of acquiring and releasing locks

Case1: Acquiring lock X

Original lockset *LS = {A, B}*

(1) Create new locking constraints:

A→X, B→X

(2) Add X in the lockset:

LS = {A, B, X}

Case 2: Releasing lock X

Original lockset LS = {A, B, X}

(1) Find and drop X in the lockset:

LS = {A, B}

[4] Savage et al. Eraser: a dynamic data race detector for multithreaded programs. In TOCS 97.

T2: Reachability-based comparison method

- S1: Identifying the same locks in target code paths
  - Field-based analysis of data structure type and field

- S2: Comparing locking constraints in target code paths to detect possible deadlocks
  - Traditional comparison:

(1) Start the comparison from each locking constraint;

(2) Compare the current locking constraint with each locking constraint in other code paths;

(3) If matched, replace the current locking constraint with the matched one;

(4) If not matched, select another locking constraint for comparison

T2: Reachability-based comparison method

- Example of traditional comparison (4 target paths TP1~TP4)
  - Traditional method:

TP1	TP2	TP3	TP4
A→B	D→A	B→C	B→E

Start from TP1{A→B}:

TP1{A→B} and TP2{D→A}: STOP
TP1{A→B} and TP3{B→C}: CONTINUE!
    TP3{B→C} and TP2{D→A}: STOP
    TP3{B→C} and TP4{B→E}: STOP
TP1{A→B} and TP4{B→E}: CONTINUE!
    TP4{B→E} and TP2{D→A}: STOP
    TP4{B→E} and TP3{B→C}: STOP

T2: Reachability-based comparison method

- Example of traditional comparison (4 target paths TP1~TP4)
  - Traditional method:

TP1	TP2	TP3	TP4
A→B	D→A	B→C	B→E

Start from TP1{A→B}:

TP1{A→B} and TP2{D→A}: STOP
TP1{A→B} and TP3{B→C}: CONTINUE!
    TP3{B→C} and TP2{D→A}: STOP
    TP3{B→C} and TP4{B→E}: STOP
TP1{A→B} and TP4{B→E}: CONTINUE!
    TP4{B→E} and TP2{D→A}: STOP
    TP4{B→E} and TP3{B→C}: STOP

Start from TP2{D→A}:

TP2{D→A} and TP1{A→B}: CONTINUE!
    TP1{A→B} and TP3{B→C}: CONTINUE!
        TP3{B→C} and TP4{B→E}: STOP
    TP1{A→B} and TP4{B→E}: CONTINUE!
        TP4{B→E} and TP3{B→C}: STOP
TP2{D→A} and TP3{B→C}: STOP
TP2{D→A} and TP4{B→E}: STOP

T2: Reachability-based comparison method

- Example of traditional comparison (4 target paths TP1~TP4)
  - Traditional method:

TP1	TP2	TP3	TP4
A→B	D→A	B→C	B→E

Start from TP1{A→B}:

TP1{A→B} and TP2{D→A}: STOP
TP1{A→B} and TP3{B→C}: CONTINUE!
TP3{B→C} and TP2{D→A}: STOP
TP3{B→C} and TP4{B→E}: STOP
TP1{A→B} and TP4{B→E}: CONTINUE!
TP4{B→E} and TP2{D→A}: STOP
TP4{B→E} and TP3{B→C}: STOP

Start from TP2{D→A}:

TP2{D→A} and TP1{A→B}: CONTINUE!
TP1{A→B} and TP3{B→C}: CONTINUE!
TP3{B→C} and TP4{B→E}: STOP
TP1{A→B} and TP4{B→E}: CONTINUE!
TP4{B→E} and TP3{B→C}: STOP
TP2{D→A} and TP3{B→C}: STOP
TP2{D→A} and TP4{B→E}: STOP

Repeated comparison

T2: Reachability-based comparison method

- New structure: *indirect locking constraint*
  - Combine multiple locking constraints for a reachable node
  - Can reduce repeated comparison

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

$$TP_{set} = \{TP_1, TP_2, \ldots, TP_n\}$$

T2: Reachability-based comparison method

- Example of traditional comparison (4 target paths TP1~TP4)
  - Our method:

TP1	TP2	TP3	TP4
A→B	D→A	B→C	B→E
A→C, [TP3]			
A→E, [TP4]			

Start from TP1{A→B}:

TP1{A→B} and TP2{D→A}: STOP

TP1{A→B} and **TP3{B→C}**: CONTINUE!

  [Create a reachable node A→C]

  TP3{B→C} and TP2{D→A}: STOP

  TP3{B→C} and TP4{B→E}: STOP

TP1{A→B} and **TP4{B→E}**: CONTINUE!

  [Create a reachable node A→E]

  TP4{B→E} and TP2{D→A}: STOP

  TP4{B→E} and TP3{B→C}: STOP

[TP1 has complete reachability graph]

T2: Reachability-based comparison method

- Example of traditional comparison (4 target paths TP1~TP4)
  - Our method:

TP1	TP2	TP3	TP4
A→B	D→A	B→C	B→E
A→C, [TP3]			
A→E, [TP4]			

Start from TP1{A→B}:

TP1{A→B} and TP2{D→A}: STOP

TP1{A→B} and **TP3{B→C}**: CONTINUE!

  [Create a reachable node A→C]

  TP3{B→C} and TP2{D→A}: STOP

  TP3{B→C} and TP4{B→E}: STOP

TP1{A→B} and **TP4{B→E}**: CONTINUE!

  [Create a reachable node A→E]

  TP4{B→E} and TP2{D→A}: STOP

  TP4{B→E} and TP3{B→C}: STOP

[TP1 has complete reachability graph]

Start from TP2{D→A}:

TP2{D→A} and TP1{A→B}: STOP (no cycle)

TP2{D→A} and TP1{A→C}: STOP (no cycle)

TP2{D→A} and TP1{A→E}: STOP (no cycle)

TP2{D→A} and TP3{B→C}: STOP

TP2{D→A} and TP4{B→E}: STOP

T3: Two-dimensional filtering strategy

- D1: Validating code-path feasibility (using Z3 [5] SMT solver)
  - Lock-usage analysis for numerous code paths:

    Light-weight and imprecise code-path checking — for efficiency
  - False-positive filtering for some possible deadlocks:

    Heavy-weight and precise code-path checking — for accuracy

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

T3: Two-dimensional filtering strategy

- D2: Validating code-path concurrency
  - Checking common lock:

    Whether the two code paths have a common lock?
  - Checking call graph:

    Whether the two code paths have common parts in call graphs?

TP1

```
spin_lock(X);
......
spin_lock(A);
......
spin_lock(B);
```

TP2

```
spin_lock(X);
......
spin_lock(B);
......
spin_lock(A);
```

Common lock

TP1

```
Func X
 -> FuncP
   -> spin_lock(A);
   -> spin_lock(B);
```

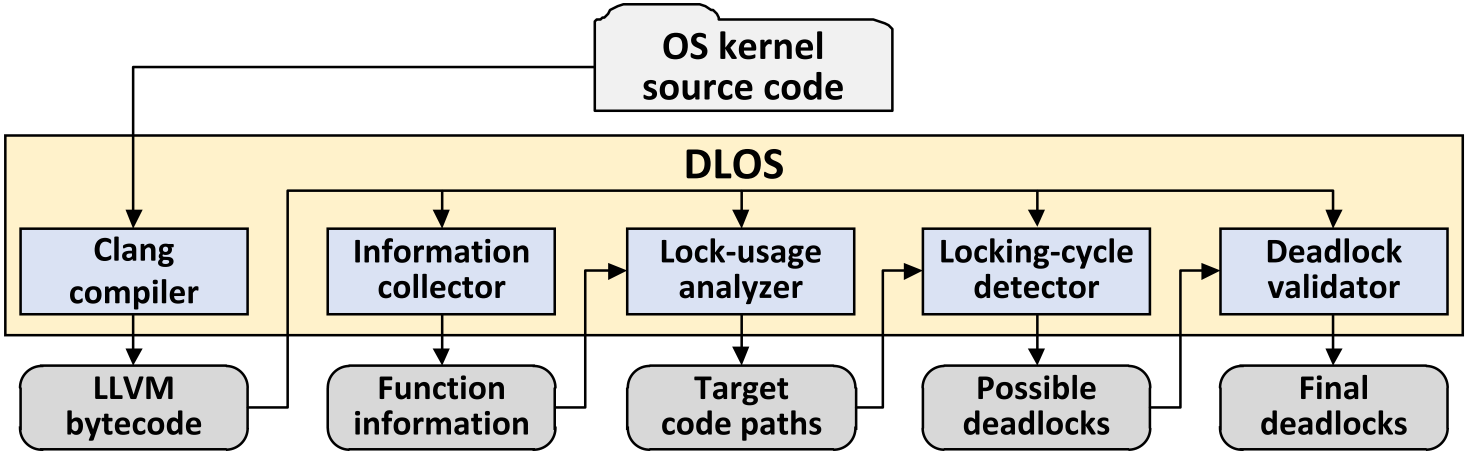
TP2

```
Func X
 -> FuncQ
   -> spin_lock(B);
   -> spin_lock(A);
```

Common part in call graph

Approach

- **DLOS** (<u>D</u>ead<u>L</u>ocks in <u>OS</u> kernels)
  - Integrate the three key techniques
  - Statically detect deadlocks in OS kernels
  - LLVM-based static analysis



Evaluation

- Linux 4.9 and 5.10
  - Use a regular PC with eight CPUs and 16GB memory
  - Use Clang-9.0
  - Make *allyesconfig* of x86-64

Evaluation

- Deadlock detection

Description		Linux 4.9	Linux 5.10
Code handling	Analyzed source files (.c)	23.7K	29.4K
	Analyzed source code lines	11.4M	14.7M
Lock-usage analysis	Distinct target code paths	102K	117K
	Locking constraints	323K	439K
Lock-cycle detection	Created indirect locking constraints	196K	222K
	Times of reducing comparison	851K	946K
	Possible deadlocks	465	539
Deadlock detection	Dropped false bugs	419	474
	Found bugs (real / all)	39 / 46	54 / 65
Time usage		372m	418m

Evaluation

- Linux 4.9
  - Find 46 deadlocks, and 39 of them are real
  - 21 deadlocks have been fixed in Linux 5.10
- Linux 5.10
  - Find 65 deadlocks, and 54 of them are real
  - 31 deadlocks have been confirmed

Some confirmed deadlocks:

- https://github.com/torvalds/linux/commit/7418e6520f22
- https://github.com/torvalds/linux/commit/7740b615b666
- https://github.com/torvalds/linux/commit/f10f582d2822

Limitations

- False positives
  - Field-based analysis is not accurate enough
  - Alias analysis is intra-procedural and flow-insensitive
  - Path validation can make mistakes in complex cases
  - ……
- False negatives
  - Incomplete bottom-up analysis of called functions
  - No analysis of function-pointer calls
  - Assume that a code path is never concurrently executed with itself
  - ……

Conclusion

- Deadlocks are dangerous and hard-to-find in OS kernels
- DLOS: static detection of deadlocks in OS kernels
  - ***T1: Summary-based lock-usage analysis*** to extract target code paths containing distinct locking constraints
  - ***T2: Reachability-based comparison method*** to detect locking cycles from locking constraints
  - ***T3: Two-dimensional filtering strategy*** to drop false positives by validating code-path feasibility and concurrency
- Find 39 and 54 real deadlocks in Linux 4.9 and 5.10
- DLOS can be extended to detecting other locking issues

Thanks for listening!

Jia-Ju Bai

E-mail: baijiaju@tsinghua.edu.cn

https://baijiaju.github.io/