Auto-detecting sleeping lock calls in non-preemptible context via static analysis

(LPC 2024 talk)

Tomas Glozar
Software Engineer, Red Hat

September 18, 2024

Outline

The problem

Proof-of-concept tool

- Algorithm design
- Implementation
- Demo

Improving the approach

The problem

The problem

- real-time kernel (PREEMPT_RT) changes the semantics of spin locks into sleeping locks
- sleeping locks, unlike standard spin locks, cannot be used when preemption is disabled
- affects all subsystems and drivers
- leads to kernel panic when the locks actually sleeps (i.e. is taken)

Example

- detected when running BPF map selftests on RT kernel[1]

```
BUG: sleeping function called from invalid context at
kernel/locking/spinlock_rt.c:35
in_atomic(): 1, irqs_disabled(): 0, non_block: 0,
pid: 17709, name: test_sockmap
Preemption disabled at:
sock_map_update_elem_sys+0x8f/0x2a0
Call Trace:
dump_stack+0x5c/0x80
 ___might_sleep.cold.95+0xf5/0x109
rt_spin_lock+0x3d/0xd0
___slab_alloc+0xc8/0x8d0
kmem_cache_alloc_trace+0xe7/0x220
sock_hash_update_common+0x54/0x4d0
sock_map_update_elem_sys+0x25a/0x2a0
```

[1] https://lore.kernel.org/lkml/ZMOrEi3cNWGXp9ZS@krava/t/

Solution?

Manually fix each case when encountered

Solution? (2)

~~Manually fix each case when encountered~~

Alternative: **Automatic detection** of sleeping functions called from invalid context

Proof-of-concept tool

What pattern to look for?

- simplest case:

```
preempt_disable();
...
spin_lock(...);
...
```

- more general case:

```
f1(); // f1 calls preempt_disable()
...
f2(); // f2 calls spin_lock()
```

- both calls may be several levels down in the call stack

General case (1)

- second case from previous slide is as general as possible (if control flow is taken into account)
  - = there is a corresponding sequence of that form for all cases of sleeping function called from invalid context
- proof:
  - to trigger the bug, there must be a call of spin_lock() or another sleeping lock inside a function; we name it f
  - now, either there is a preceeding call to a function calling preempt_disable (we name it g) in f preceeding the call to spin_lock
  - or a function calling preempt_disable (again called g) is called before a call to f upper in the stack, in another function; we name it h

General case (2)

- first case:

```
f(...) { ...
    g();            // f1 from pattern
    ...
    spin_lock(...); // f2 from pattern
... }
```

- second case:

```
h(...) { ...
    g(...); // f1 from pattern
    ...
    f'(...); // f2 from pattern; calls f eventually
... }
```

- → it is enough to look for this pattern

Graph algorithm

- to detect the pattern in a function (without any control statements), one needs to do two things:
    - know how its callees behave regarding scheduling and preemption (this we call *preemption* and *scheduling semantics*)
    - whether a callee that calls schedule() is present at a place in the function where preemption is disabled
- this can be easily done by recursion on a function call graph
    - assigning semantics to functions done at the same time as finding violations
    - if semantics are unknown for a callee, the assignment procedure is called recursively on it
    - requires having a set of functions with pre-assigned semantics

Graph algorithm example (1)

example of the graph algorithm (modeled roughly on the bpf sockmap BUG):

```
int map_lock() {
    preempt_disable();
}

int map_unlock() {
    preempt_enable();
}

int access_map() {
    map_lock();
    spin_lock(&another_lock);
    _access_map();
    spin_unlock(&another_lock);
    map_unlock();
}
```

```
known semantics:
    preempt_disable:
        disables preemption
    preempt_enable:
        enables preemption
    spin_lock:
        sleeps
```

Graph algorithm example (2)

example of the graph algorithm (modeled roughly on the bpf sockmap BUG):

```
int map_lock() {
    preempt_disable();
}

int map_unlock() {
    preempt_enable();
}

int access_map() {
    map_lock();
    spin_lock(&another_lock);
    _access_map();
    spin_unlock(&another_lock);
    map_unlock();
}
```

```
known semantics:
    preempt_disable:
        disables preemption
    preempt_enable:
        enables preemption
    spin_lock:
        sleeps
    access_map:
       no change
```

Graph algorithm example (3)

example of the graph algorithm (modeled roughly on the bpf sockmap BUG):

```
int map_lock() {
    preempt_disable();
}

int map_unlock() {
    preempt_enable();
}
int access_map() {
    map_lock();
    spin_lock(&another_lock);
    _access_map();
    spin_unlock(&another_lock);
    map_unlock();
}
```

```
known semantics:
    preempt_disable:
        disables preemption
    preempt_enable:
        enables preemption
    spin_lock:
        sleeps
    access_map:
      no change
```

Graph algorithm example (4)

example of the graph algorithm (modeled roughly on the bpf sockmap BUG):

```
int map_lock() {
    preempt_disable();
}

int map_unlock() {
    preempt_enable();
}

int access_map() {
    map_lock();
    spin_lock(&another_lock);
    _access_map();
    spin_unlock(&another_lock);
    map_unlock();
}
```

```
known semantics:
    preempt_disable:
        disables preemption
    preempt_enable:
        enables preemption
    spin_lock:
        sleeps
    access_map:
        no change
    map_lock:
        no change
```

Graph algorithm example (5)

example of the graph algorithm (modeled roughly on the bpf sockmap BUG):

```
int map_lock() {
    preempt_disable();
}

int map_unlock() {
    preempt_enable();
}

int access_map() {
    map_lock();
    spin_lock(&another_lock);
    _access_map();
    spin_unlock(&another_lock);
    map_unlock();
}
```

```
known semantics:
    preempt_disable:
        disables preemption
    preempt_enable:
        enables preemption
    spin_lock:
        sleeps
    access_map:
        no change
    map_lock:
        disables preemption
```

Graph algorithm example (6)

example of the graph algorithm (modeled roughly on the bpf sockmap BUG):

```
int map_lock() {
    preempt_disable();
}

int map_unlock() {
    preempt_enable();
}

int access_map() {
    map_lock();
    spin_lock(&another_lock);
    _access_map();
    spin_unlock(&another_lock);
    map_unlock();
}
```

```
known semantics:
    preempt_disable:
        disables preemption
    preempt_enable:
        enables preemption
    spin_lock:
        sleeps
    access_map:
        disables preemption
    map_lock:
        disables preemption
```


Red Hat

Graph algorithm example (7)

example of the graph algorithm (modeled roughly on the bpf sockmap BUG):

```
int map_lock() {
    preempt_disable();
}

int map_unlock() {
    preempt_enable();
}

int access_map() {
    map_lock();
    spin_lock(&another_lock);
    _access_map();
    spin_unlock(&another_lock);
    map_unlock();
}
```

```
known semantics:
    preempt_disable:
        disables preemption
    preempt_enable:
        enables preemption
    spin_lock:
        sleeps
    access_map:
        disables preemption, sleeps
    map_lock:
        disables preemption
```

Graph algorithm example (8)

Violation detected!

Graph algorithm example (9)

Output from PoC tool (more about it on next slide):

```
$ bin/rtlockscope /tmp/artificial_case/ kb/default-rt.yaml
...
Sleeping lock called at:
access_map at case.c:9
spin_lock at case.c:11

preemption disabled at:
access_map at case.c:9
map_lock at case.c:10
preempt_disable at case.c:2

Statistics:
- 1 cases found
```

rtlockscope tool[2]

- prototype tool for auto-detecting scheduling with preemption disabled
- accepts two arguments: the source code tree and a *knowledge base*
  - the KB is in YAML format and represents the initial knowledge about semantics
- makes use of graph algorithm described above (plus call stack backtracking, as seen on previous slide)
- currently quite simple (under 400 lines of Python code), makes use of external tools
  - `ctags` is used to find all functions in a source tree
  - `cscope` is used to look up callees of functions

[2] `https://gitlab.com/tglozar/rtlockscope`

rtlockscope knowledge base

```
$ cat kb/default-rt.yaml
# Direct preemption setting functions
preempt_disable:
  preempt-semantics: preempt-disabling
preempt_enable:
  preempt-semantics: preempt-enabling
...
# Sleeping locks
schedule:
  lock-semantics: sleeping
spin_lock:
  lock-semantics: sleeping
```


Red Hat

The important question (1)

Does rtlockscope work *reasonably well* on the Linux kernel?

The important question (2)

Does rtlockscope work *reasonably well* on the Linux kernel?

Kind of.

rtlockscope false positive example

```
if (a)
    preempt_disable();
if (!a)
    schedule();
if (a)
    preempt_enable();
```

rtlockscope false negative example

```
preempt_enable();
while (...) {
    schedule();
    preempt_enable();
    ...
    preempt_disable();
}
```

Current rtlockscope limitations

- rtlockscope ignores the *control flow* of the program
- the sequence of function calls ordered by the source code is not necessarily the one occuring at runtime
- e.g. loops, conditional statements, recursion
- usually, this is not a problem, since the enabling and disabling of preemption usually wraps a block of code
- an issue is that one wrongly assigned semantics can generate or suppress thousands of violations

Simple workaround

- = add functions which are labeled wrong to KB with correct labeling
- requires manual updates to the KB, but keeps the algorithm simple
- theoretically sound and complete (in worst case, label all functions)

Demo

rtlockscope run on a Linux kernel source tree (6.10.3)[3]

[3] asciinema, result.log

Improving the approach

How to improve rtlockscope further?

- one way is to keep refining the knowledge base (adding new annotations for problematic functions)
- another one is to improve the algorithm
- idea: view the problem as finding and checking possible sequences of events
- this separates it into two parts:
  - find the set of all possible sequences of relevant events ("traces")
  - determine from the set if a violation can happen
- this turns out to be a useful way of viewing the problem

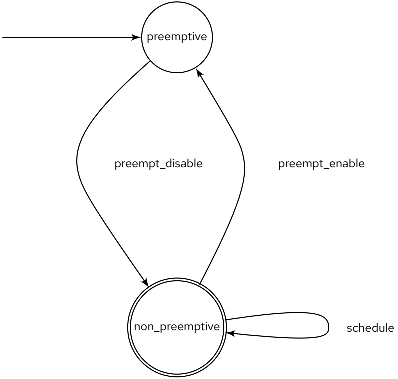
Formal representation of the problem

- let $E = \{PE, PD, S\}$ be a set of events we are interested in
  - PE ... `preempt_enable`
  - PD ... `preempt_disable`
  - S ... `schedule`
- let $A \subseteq E^*$ be the set of all sequences of events that can happen on a system
- let $V \subseteq E^*$ be the set of all violations, i.e. sequences where `schedule` is called under non-preemptible context
- statement of problem: prove $A \cap V = \emptyset$

Properties of sequence sets *A*, *V*

- *V* is either a regular language or context-free language, depending on whether nesting is allowed
- thus, it is representable with a finite-state automaton or a push-down automaton
- automaton representation is used, for example, in kernel rv subsystem[1] for runtime verification
- *A* is more complicated, since it derives from the program itself (the Linux kernel)
    - in rv, one concrete sequence of events is observed on a live kernel
    - here in static analysis, we have to account for all such possible sequences

rtlockscope problem as an automaton



rtlockscope as event sequence generator

- instead of determining semantics, attach a set of event sequences to each function to the kernel
- then check each sequence for violations
- practically, this is the same as the original graph algorithm: the semantics assigned by rtlockscope are just a function between states of the automaton given by the corresponding sequences
- however, this point of view gives us intermediate representation that can be processed further

Sequence assignment example

same program as seen before, but with assigned sequences (as regexes) instead of semantics

```
int map_lock() {
    preempt_disable();
}

int map_unlock() {
    preempt_enable();
}

int access_map() {
    map_lock();
    spin_lock(&another_lock);
    _access_map();
    spin_unlock(&another_lock);
    map_unlock();
}
```

```
event sequences:
    preempt_disable:
        PD
    preempt_enable:
        PE
    spin_lock:
        S
    map_lock:
        [preempt_disable] = PD
    map_unlock:
        [preempt_enable] = PE
    access_map:
        [map_lock] [spin_lock] [map_unlock]
         = PD S PE
```

Advantages of sequence representation

- sequences closely resemble the original source code but drop all details unnecessary for the analysis
- regular expressions can be used to represent sequence sets
- sequence representation can be extended to include control sequences:

```
if (a) PD
if (!a) S
if (a) PE
```

- the extended representation can be seen and manipulated as superset of C, and transformations may be applied to common patterns in code[4]

[4] For another static analysis project using patterns on the Linux kernel, see DiffKemp[2]

Sequence transformation pattern example

```
if (EXPR)
    PD;
if (!EXPR)
    S;          ---->  (PD PE) + (S)
if (EXPR)
    PE;
```

Some challenges

- implementing patterns efficiently without blow-up of the number of event sequences
- creating a database of patterns that is efficient on the Linux kernel in addition to a knowledge base

Conclusion

- static analysis can be used to detect at least some scheduling while atomic issues in the Linux kernel
- although static analysis is theoretically hard, real-world programs have only a limited degree of complexity that can be resolved with specialized algorithms
- the rtlockscope approach is connected to the kernel’s rv subsystem and the formalism under it as well as the approach of DiffKemp
- possible generalization to other verification problems

Questions?

References I

- Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira.
  Efficient formal verification for the linux kernel.
  In Peter Csaba Ölveczky and Gwen Salaün, editors, Software Engineering and Formal Methods, pages 315–332, Cham, 2019. Springer International Publishing.
- Viktor Malík, Petr Šilling, and Tomáš Vojnar.
  Applying custom patterns in semantic equality analysis.
  In Mohammed-Amine Koulali and Mira Mezini, editors, Networked Systems, pages 265–282, Cham, 2022. Springer International Publishing.