COBET - Incorrect usage of atomic operations bug pattern and Waiting for an already finished thread bug pattern

Shin Hong and Moonzoo Kim
Computer Science Department
KAIST, South Korea
hongshin@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

1 Incorrect usage of atomic operations

Most computer architectures provide special instruction sets for synchronization operations called *atomic instructions*. ‘test-and-set’ and ‘compare-and-swap’ are popular forms of these instructions. The distinguishable feature of atomic instructions is that computer architectures guarantee the atomic execution of read and consequent update on operands. These instructions are utilized for implementing higher synchronization mechanisms such as lock or semaphore, or used directly by programmers for light-weight synchronizations.

Linux kernel provides special variable type `atomic_t` and the library functions for atomic instructions (e.g., `atomic_read`, `atomic_set`). These are used for lock free synchronization on shared variables for simple and frequent updates. frequent and simple updates. Many counter variables on shared data structures are implemented by `atomic_t` and its associated operations.

Atomic instructions synchronize two threads by executing read and update operations atomically. For this reason, the programmers should use proper combined atomic instructiosn for two related read and write operations. However, programmers mistakenly use two separate atomic operations instead of one combined instruction, and it may lead to data race bugs.

We characterize this incorrect atomic instruction usages as a bug pattern shown in Figure 1. `pattern 1` represents two consecutive atomic instructions whose executions are intended to be atomic. `pattern 2` expresses another thread which can be scheduled between two atomic instructions inf `pattern 1` and interfere their executions.

The semantic condition checking examines the following conditions:

1. A lockset at `3l` must be disjoint with the set of held locks at `3m`. Otherwise, `pattern 1` and `pattern 2` do not interfere with each other.
2. `$a1` and `$a3` should be shared variables and may alias to each other.
3. `$a2` should contain a variable which is equal to or aliased to `$a1`.

```
1l:pattern 1 {             1m:pattern 2 {
2l:  fun $f1 {             2m: fun $f2 {
3l:    call $atomic1 $a1 ; 3m:  call $atomic2 $a3 ;
4l:    if ($a2) { }        4m: }}
5l:    }}
```

Figure 1: Incorrect usage of atomic operations pattern

4. `$atomic1` is an atomic instruction which updates `$a1` (e.g., `atomic_inc`, `atomic_set`, etc). And `$atomic2` is an atomic instruction which updates `$a3`.
5. `$a2` should contains a function call to an atomic instruction which reads `$a1` (e.g, `atomic_read()`, etc).

We refine the bug pattern detector code manually to supplement expression level conditions. The bug pattern detector refers the type of the associated variables to check whether or not related variables are `atomic_t` variables. We provide the function names of library functions for atomic instructions and the bug pattern detector utilizes these information at checking the fourth and the fifth semantic conditions.

Using the bug pattern detector, we found undiscovered bugs in Linux kernel, one of them is shown in Figure 2. This code updates `&cp->in_pkts` at line 5 and then examines its value at line 10 and these might not be executed atomically. We repored this issue according to the result from the bug pattern detector, and the maintainers immediately patch the code that the two separate atomic instructions are replaced to a proper combined atomic instruction.

COBET found a new bug of this pattern (see Figure 2) in the `netfilter` network module (see Linux ChangeLog 2.6.32.).

2 Waiting for an Already Finished Thread

A Linux kernel can create and execute a child thread by calling `kthread_run()` and terminate the child thread by calling `kthread_stop()`. `kthread_stop()` sends a special message to a child thread and waits until the child thread is terminated. A child thread should regularly call `kthread_should_stop()` which returns true if the message is received and terminates respondingly. Otherwise, the parent thread waits indefinitely at `kthread_stop()` as no other thread can make the handshaking with the parent thread.

One programming pattern of a child thread is a function with a loop conditioned by `kthread_should_stop()`. The child thread processes task until `kthread_should_stop()` returns true. In this situation, the child thread should not terminate although the task is finished or the task may not proceed for errors, because the parent thread may invoke `kthread_stop()` after the child thread terminates and it results in indefinite waiting of the parent thread.

```
unsigned int ip_vs_in(unsigned int hooknum,
 sk_buff *skb, net_device *in, net_vice *out,
 int(*okfn)(sk_buff *)) {
 ...
 atomic_inc(&cp->in_pkts);
 if (af == AF_INET &&
     (ip_vs_sync_state & IP_VS_STATE_MASTER)
     && (((cp->protocol != IPPROTO_TCP ||
      && cp->state == IP_VS_TCP_S_ESTABLISHED)
       (atomic_read(&cp->in_pkts) %
        sysctl_ip_vs_sync_threshold[1]
        == sysctl_ip_vs_sync_threshold[0])) ||
      ((cp->protocol == IPPROTO_TCP) &&
       (cp->old_state != cp->state) &&
       ((cp->state == IP_VS_TCP_S_FIN_WAIT) ||
        (cp->state == IP_VS_TCP_S_CLOSE_WAIT) ||
        (cp->state == IP_VS_TCP_S_TIME_WAIT)))))
  ip_vs_sync_conn(cp);
  cp->old_state = cp->state ;
  ...
```

Figure 2: Incorrect usage of atomic operations bug detected at net/netfilter/ipvs/ip_vs_core.c, Linux 2.6.30.4

```
1p:pattern 1 {                 1q:pattern 2 {
2p: fun $f1 {                  2q: fun $f2 {
3p:  call "kthread_run" $a1 ;  3q:  loop $c1 {
4p:  call "kthread_stop" $a2 ; 4q:      if $c2 {
5p: }}                         5q:        break ; }
                               6q:  }}}
```

Figure 3: Waiting for an Already Finished Threads

```
1r: static int cleaner_kthread(void *arg) {
2r:   btrfs_root *root = arg;
3r:   do {
4r:     smp_mb() ;
5r:       if (root->fs_info->closing)
6r:         break ;
7r:       ...
8r:   } while(!kthread_should_stop()) ;
9r:   return 0; }
```

Figure 4: Waiting for an already finished thread bug detected at `fs/btrfs/async-thread.c` Linux 2.6.30.4

However, if a child thread has a path which escape the loop regardless of `kthread_should_stop()`, the child thread can terminate earlier than the parent thread's `kthread_stop()` and result in the deadlock situation. We observe an actual bug of this type reported at Linux Change Log 2.6.28, where the loop is escaped by `break` statement for an error condition.

We specify this bug into a PDL pattern as shown in Figure 3. `pattern 1` represents a parent thread which creates a child thread and invokes `kthread_stop()` for the child thread. `pattern 2` specifies the function for child thread which has a loop with `kthread_should_stop()`. Lines `5r-6r` represent the branch which escape the loop illegally. We manually supplemented the detailed conditions related to expressions. For this pattern, the semantics condition checking does not have any additional condition.

In Linux 2.6.30.4, we found the undiscovered bugs from `btrfs` file system code, one of them shown in Figure 4. The child thread unintentionally terminated when the error handling branch is taken (line 5r-6r). The `btrfs` developer confirmed these bugs. And after Linux 2.6.32, the Linux kernel developers modified the semantics of `kthread_stop()` not to wait if the child thread is already finished, so that the bug of this categories may not induce error.