Improving Linux development with better tools

Andi Kleen

Oct 2013
Intel Corporation
ak@linux.intel.com
Linux complexity growing

Source lines in Linux kernel

All source code

Linux kernel source lines IO

net/ fs/ block/

Source lines Linux Kernel core

kernel/ lib

Kernel version

M-LOC

Kernel version

M-LOC

Kernel version

M-LOC
Do we have a problem?

- If we assume number of bugs stays constant per line there would be more and more bugs.

- If we assume programmers don't get cleverer some code may become too complex to change/debug.
Or we can use better tools to find bugs

- Static code checker tools
- Dynamic runtime checkers
- Fuzzers/test suites
- Tracers to understand code
- Tools to understand the source
Static checkers

- sparse, smatch, coccinelle, clang checker, checkpatch, gcc -W/LTO, stanse

- Can check a lot of things, simple mistakes, complex problems

- Generic C and kernel specific rules
Static checker challenges

- Some are very slow
- False positives
  - Often only can do new warnings
  - Otherwise too many false positives
- May need concentrated effort to get false positives down
  - Only done for gcc/sparse/checkpatch so far
  - Needs both changes to Linux and to checkers
Study bug fixes

• “At least 14.8%~24.4% of the sampled bug fixes are incorrect. Moreover, 43% of the incorrect fixes resulted in severe bugs that caused crash, hang, data corruption or security problems.”

  • “How do fixes become bugs” Yin/Yuan et.al.
  • http://opera.ucsd.edu/~zyin2/fse11.pdf
  • Great paper, every kernel programmer should read it

• Can new rules for static checkers help?
Coccinelle checker

// Find &&/|| operations that include the same argument more than once
// A common source of false positives is when the argument performs a side
// effect.
@r expression@
expression E;
position p;
@@
(* E@p
 || ... || E
 |
(* E@p
 && ... && E
)
@script:python depends on org@
p <= r.p;
@@
cocci.print_main("duplicated argument to && or ||",p)
Challenge: global checks

- No static checker I found can follow indirect calls ("OO in C", common in kernel)
  
  ```c
  struct foo_ops {
      int (*do_foo)(struct foo *obj);
  }
  foo->do_foo(foo);
  ```

- Can be done by using type information

- Misses a lot of potential bugs
Lock ordering: lockdep

- Deadlock from lock ordering ("ABBA" bugs) used to be common
  
  T1                             T2
  lock(a);                     lock(b);
  lock(b);                     lock(a);

- Lockdep basically eliminated this problem
- Checks lock ordering, interrupt flags violations at runtime
- Unfortunately scaling problems on large systems
Kmemcheck / AddressSanitizer

- Check uninitialized/freed/out of bounds data
- Kmemcheck based on page faults
  - Quite slow
- AddressSanitizer using compiler instrumentation
  - Much faster
  - Kernel library seems to exist, but not released yet
Thread checkers

• Find data races:
  – Shared data accesses not protected by locks

• User space: helgrind, ThreadSanitizer, ..
  – ThreadSanitizer compiler based and could be used in kernel

• Problem: kernel does not mark lock-less accesses, which would be false positives.
  User lock less code:
  ```c
  __atomic_store_n(&foo, 1, __ATOMIC_SEQ_CST);
  ```
  Kernel:
  ```c
  foo = 1;
  mb();
  ```
Undefined behavior checker

• UBSan: New gcc/LLVM feature

• Checks undefined C behavior at runtime
  – e.g. x << 100, signed integer overflows, …

• Needs special runtime library

• Would need to be ported to kernel
Fuzzers

- Use random input data to find bugs
- Trinity is a great tool
  - Finds many bugs
- Needs manual model for each syscall
  How do we cover all the ioctls/sys/proc files?
- Modern fuzzers around using automatic feedback by instrumenting code
  - But not for kernel yet
The biggest challenge

• How to run all these tools on every new patch:
  – Cannot ask every developer to use all of them

• Static checkers are relatively easy
  – But can we get beyond just deltas for new code?

• But how to run the dynamic tools?
Test suites

- Ideally all kernel code would come with a test suite
  - Then someone could run all the dynamic checkers
- Difficult for hardware drivers
- LKP, kernel unit tests, tools/* limited
- Need a real unit testing framework
Coverage

• Kernel gcov can be used to test coverage of test suites
• Should be used much more widely
Tracers

• Long beyond “real men don't use debuggers”
  – Linux has good debuggers these days (kgdb etc.)

• But how to debug hard to reproduce bugs
  – Ideal enough information to debug on first trigger

• Tracing:
  – Low overhead instrumentation
  – When problem triggers dump data
ftrace: function tracer

• Trace all functions in the kernel for PID
  # trace-cmd record -p function -e sched_switch -P $(pidof firefox-bin)
     plugin function
    disable all
  enable sched_switch
  path = /sys/kernel/debug/tracing/events/sched_switch/enable
  path = /sys/kernel/debug/tracing/events/*/sched_switch/enable
  path = /sys/kernel/debug/tracing/events/sched_switch/enable
  path = /sys/kernel/debug/tracing/events/*/sched_switch/enable
  Hit Ctrl^C to stop recording
...
  # trace-cmd report
...
    firefox-bin-13822 [002] 36628.537061: function:
    sys_poll
    poll_select_set_timeout
      ktime_get_ts
        timekeeping_get_ns
          set_normalized_timespec
            timespec_add_safe
              set_normalized_timespec
    do_sys_poll
      copy_from_user
        might_fault
          _cond_resched
            should_resched
              need_resched
                test_ti_thread_flag
    ...

All kernel functions executed
Ftrace / kernelshark

- Can dump on events / oops / custom triggers
- But still too much overhead in many cases to run always during testing

- Lots of other tracers not mentioned here
  - systemtap, perf, k/uprobes, ...
Intel Processor Trace (PT)

- Upcoming Intel CPU feature
- Traces all branches with low overhead
- Will be supported in perf and gdb
- Can be used as “Flight Recorder”

- Tells you “how you got there” on a problem
Biggest challenge with tracers

- They generate too much data
- Need better tools to analyze the data
- Can machine learning/analytics help?
Understanding source code

- Often first problem is finding the code
- grep/cscope work great for many cases
- But do not understand indirect pointers (OO in C model used in kernel): Give me all “do_foo” instances

```c
struct foo_ops {
    int (*do_foo)(struct foo *obj);
} = { .do_foo = my_foo };
foo->do_foo(foo)
```

- Would be great to have a cscope like tool that understands this based on types/initializers
Conclusion

- Linux has a lot of great tools for making kernel development easier
- We need them to keep up with the growing complexity
- But still many improvements possible

- Questions?