OS Bugs

Required reading: Bugs as deviant behavior

Overview

Operating systems must obey many rules for correctness and performance. Examples rules:

- Do not call blocking functions with interrupts disabled or spin lock held
- check for NULL results
- Do not allocate large stack variables
- Do no re-use already-allocated memory
- Check user pointers before using them in kernel mode
- Release acquired locks

In addition, there are standard software engineering rules, like use function results in consistent ways.

These rules are typically not checked by a compiler, even though they could be checked by a compiler, in principle. The goal of the meta-level compilation project is to allow system implementors to write system-specific compiler extensions that check the source code for rule violations.

The results are good: many new bugs found (500-1000) in Linux alone. The paper for today studies these bugs and attempts to draw lessons from these bugs.

Are kernel error worse than user-level errors? That is, if we get the kernel correct, then we won't have system crashes?

Errors in JOS kernel

What are unstated invariants in the JOS?

- Interrupts are disabled in kernel mode
- Only env 1 has access to disk
- All registers are saved & restored on context switch
- Application code is never executed with CPL 0
- Don't allocate an already-allocated physical page
- Propagate error messages to user applications (e.g., out of resources)
- Map pipe before fd
- Unmap fd before pipe
- A spawned program should have open only file descriptors 0, 1, and 2.

- Pass sometimes size in bytes and sometimes in block number to a given file system function.
- User pointers should be run through TRUP before used by the kernel

Could these errors have been caught by metacompilation? Would metacompilation have caught the pipe race condition? (Probably not, it happens in only one place.)

How confident are you that your code is correct? For example, are you sure interrupts are always disabled in kernel mode? How would you test?

Metacompilation

A system programmer writes the rule checkers in a high-level, state-machine language (metal). These checkers are dynamically linked into an extensible version of g++, xg++. Xg++ applies the rule checkers to every possible execution path of a function that is being compiled.

An example rule from the OSDI paper:

```
sm check_interrupts {
   decl { unsigned} flags;
   pat enable = { sti(); } | {restore_flags(flags);} ;
   pat disable = { cli(); };

   is_enabled: disable ==> is_disabled | enable ==> { err("double
      enable")};
   ...
```

A more complete version found 82 errors in the Linux 2.3.99 kernel.

Common mistake:

```
get_free_buffer ( ... ) {
   ....
   save_flags (flags);
   cli ();
   if ((bh = sh->buffer_pool) == NULL)
      return NULL;
   ....
}
```

(Figure 2 also lists a simple metarule.)

Some checkers produce false positives, because of limitations of both static analysis and the checkers, which mostly use local analysis.

How does the **block** checker work? The first pass is a rule that marks functions as potentially blocking. After processing a function, the checker emits the function's flow graph to a file (including, annotations and functions called). The second pass takes the merged flow graph of all function calls, and produces a file with all functions that have a path in the control-flow-graph to a blocking function call. For the Linux kernel this results in 3,000 functions that potentially could call sleep. Yet another checker like check_interrupts checks if a function calls any of the 3,000 functions with interrupts disabled. Etc.

This paper

Writing rules is painful. First, you have to write them. Second, how do you decide what to check? Was it easy to enumerate all conventions for JOS?

Insight: infer programmer "beliefs" from code and cross-check for contradictions. If *cli* is always followed by *sti*, except in one case, perhaps something is wrong. This simplifies life because we can write generic checkers instead of checkers that specifically check for *sti*, and perhaps we get lucky and find other temporal ordering conventions.

Do we know which case is wrong? The 999 times or the 1 time that *sti* is absent? (No, this method cannot figure what the correct sequence is but it can flag that something is weird, which in practice useful.) The method just detects inconsistencies.

Is every inconsistency an error? No, some inconsistency don't indicate an error. If a call to function *f* is often followed by call to function *g*, does that imply that f should always be followed by g? (No!)

Solution: MUST beliefs and MAYBE beliefs. MUST beliefs are invariants that must hold; any inconsistency indicates an error. If a pointer is dereferences, then the programmer MUST believe that the pointer is pointing to something that can be dereferenced (i.e., the pointer is definitely not zero). MUST beliefs can be checked using "internal inconsistencies".

An aside, can zero pointers pointers be detected during runtime? (Sure, unmap the page at address zero.) Why is metacompilation still valuable? (At runtime you will find only the null pointers that your test code dereferenced; not all possible dereferences of null pointers.) An even more convincing example for Metacompilation is tracking user pointers that the kernel dereferences. (Is this a MUST belief?)

MAYBE beliefs are invariants that are suggested by the code, but they maybe coincidences. MAYBE beliefs are ranked by statistical analysis, and perhaps augmented with input about functions names (e.g., alloc and free are important). Is it computationally feasible to check every MAYBE belief? Could there be much noise?

What errors won't this approach catch?

Paper discussion

This paper is best discussed by studying every code fragment. Most code fragments are pieces of code from Linux distributions; these mistakes are real!

Section 3.1. what is the error? how does metacompilation catch it?

Figure 1. what is the error? is there one?

Code fragments from 6.1. what is the error? how does metacompilation catch it?

Figure 3. what is the error? how does metacompilation catch it?

Section 8.3. what is the error? how does metacompilation catch it?