How to find lots of bugs in real code with system-specific static analysis

Dawson Engler
Stanford University

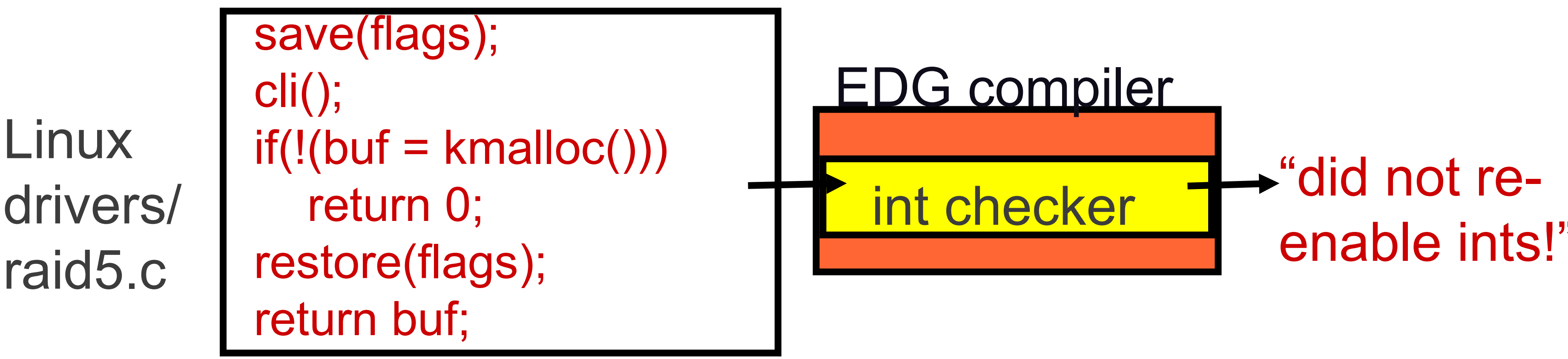
Ben Chelf, Andy Chou, Seth Hallem
Coverity

The core static bug finding intuition

- Systems software has many ad-hoc restrictions:
  - "acquire lock L before accessing shared variable X"
  - "disabled interrupts must be re-enabled"
  - Error = crashed system. How to find?
- Observation: rules can be checked with a compiler
  - scan source for "relevant" acts, check that correct.
  - E.g., to check "disabled interrupts must be re-enabled:" scan for calls to disable()/enable(), check matching, not done twice
- Main problem:
  - compiler has machinery to check, but not knowledge
  - implementor has knowledge but not machinery
- System-specific static analysis:
  - give implementors a framework to add easily-written, system-specific compiler extensions

System-specific static analysis

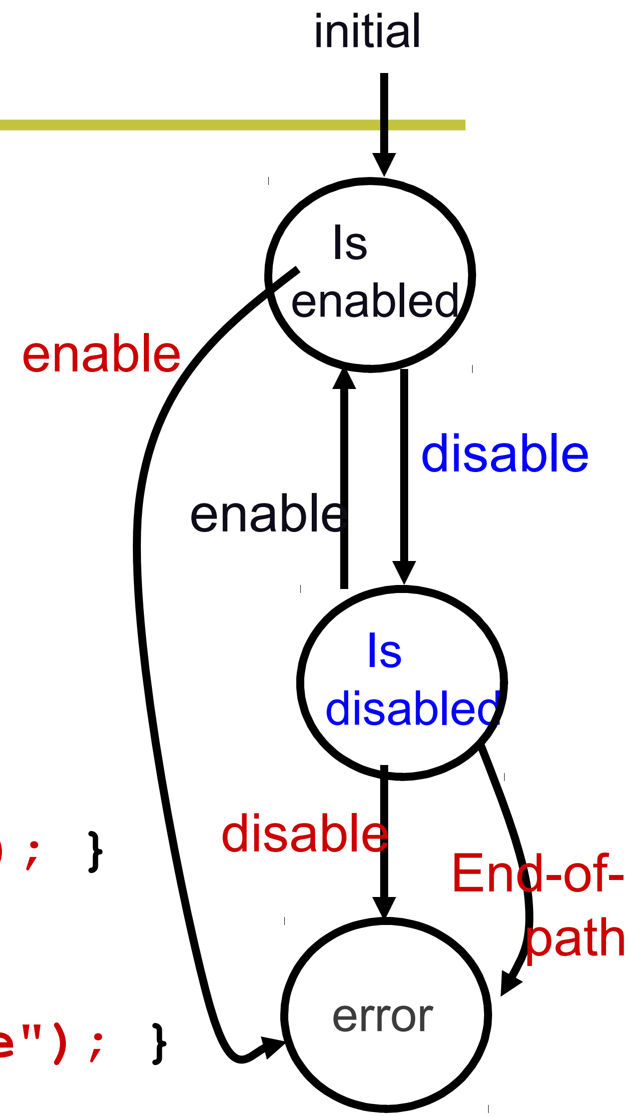
- Implementation:
  - Extensions dynamically linked into EDG C compiler
  - Applied down all paths ("flow sensitive"), across all procedures ("interprocedural") in input program source at compile time.



  - Scalable: handles millions of lines of code
  - Precise: says exactly what error was
  - Immediate: finds bugs without having to execute path
  - Effective: 1000s of bugs in Linux, OpenBSD, Commercial

A bit more detail

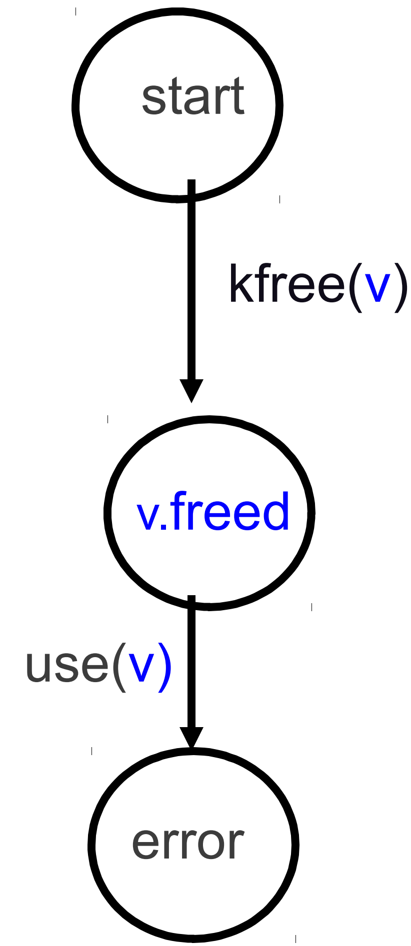
```
{ #include "linux-includes.h" }
sm chk_interrupts {
 decl { unsigned } flags;
 // named patterns
 pat enable = { sti(); }
       | { restore_flags(flags); };
 pat disable = { cli(); };

 // states
 is_enabled: disable ==> is_disabled
   | enable ==> { err("double enable"); }
   ;
 is_disabled: enable ==> is_enabled
   | disable ==> { err("double disable"); }
   | $end_of_path$ ==>
    { err("exiting w/intr disabled!"); }
   ; }
```



No X after Y: do not use freed memory

```
sm free_checker {
 state decl any_pointer v;
 decl any_pointer x;

 start: { kfree(v); } ==> v.freed
   ;
 v.freed:
    { v != x } || { v == x }
              ==> { /* do nothing */ }
 | { v } ==> { err("Use after free!"); }
    ;
}
```

```
/* 2.4.1: fs/proc/generic.c */
ent->data = kmalloc(…)
if(!ent->data) {
     kfree(ent);
     goto out;
…
out: return ent;
```



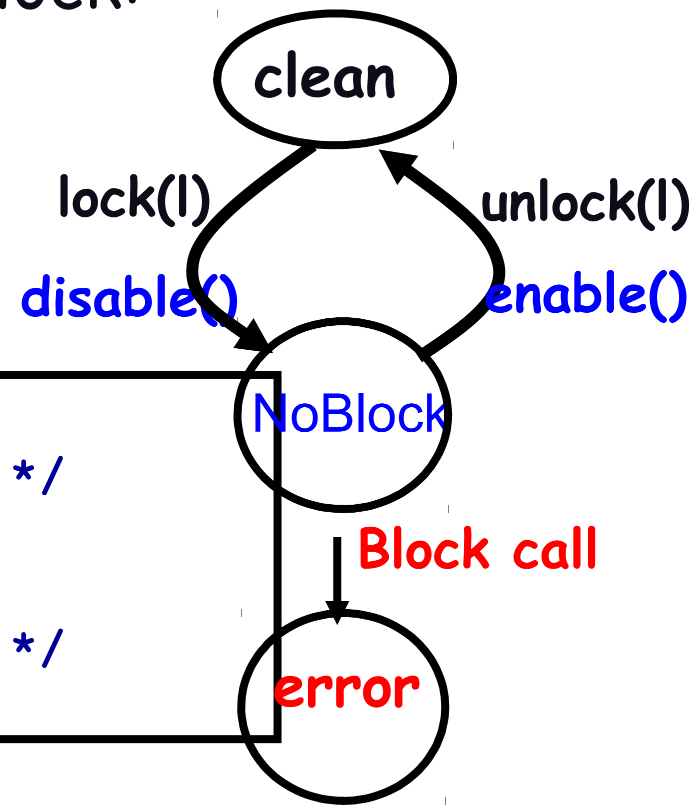
“In context Y, don’t do X”: blocking

- Linux: if interrupts are disabled, or spin lock held, do not call an operation that could block:

  Compute transitive closure of all potentially blocking fn’s

  Hit disable/lock: warn of any calls

  123 errors, 8 false pos



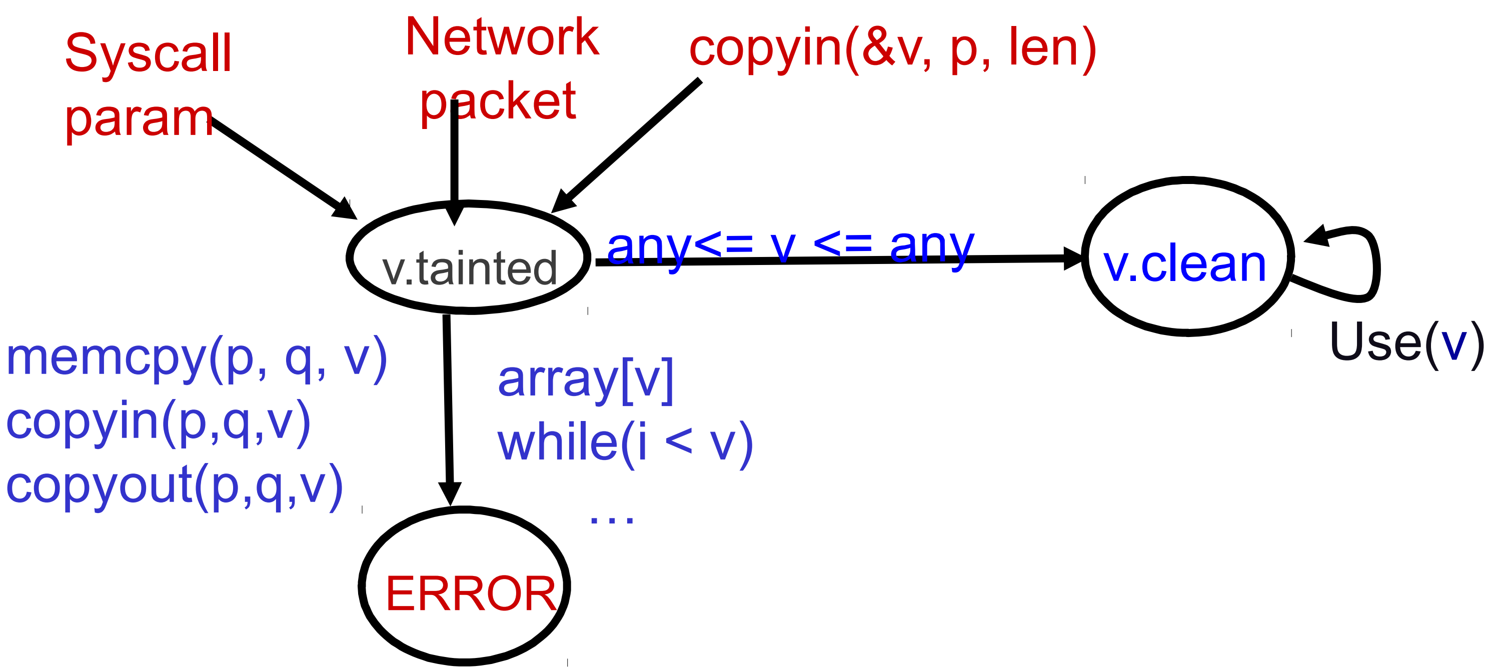

```
 drivers/net/pcmcia/wavelan_cs.c */
n_lock_irqsave (&lp->lock, flags);/* 1889 */
tch(cmd)

case SIOCGIWPRIV:                    /* 2304 */
 if(copy_to_user(wrq->u.data.pointer, …))
```

Heavy clustering:

net/atm: 152 checks, 22 bugs (exp 1.9) P =3.1x10^-15

"X before Y": sanitize integers before use

- Security: OS must check user integers before use
- Checker: Warn when unchecked integers from untrusted sources reach trusting sinks



Global; simple to retarget (text file with 2 srcs&12 sinks)

Linux: 125 errors, 24 false; BSD: 12 errors, 4 false

Some big, gaping security holes.

- Remote exploit, no checks

```
/* 2.4.9/drivers/isdn/act2000/capi.c:actcapi_dispatch */
isdn_ctrl cmd;
...
while ((skb = skb_dequeue(&card->rcvq))) {
   msg = skb->data;
   ...
   memcpy(cmd.parm.setup.phone,msg->msg.connect_ind.addr.num,
                             msg->msg.connect_ind.addr.len - 1);
```

- Missed lower-bound check:

```
/* 2.4.5/drivers/char/drm/i810_dma.c */
if(copy_from_user(&d, arg, sizeof(arg)))
   return -EFAULT;
if(d.idx > dma->buf_count)
   return -EINVAL;
buf = dma->buflist[d.idx];
Copy_from_user(buf_priv->virtual, d.address, d.used);
```

Enforcing subtle rules

- Unexpected overflow

```
copy_from_user(&wrthdr, addr, sizeof wrthdr);
if ( wrthdr.size + wrthdr.offset > FST_MEMSIZE )
        return -ENXIO;
copy_from_user(card->mem+wrthdr.offset,data,wrthdr.size)
```

```
/* 2.4.9-ac7/fs/intermezzo/psdev.c */
error = copy_from_user(&input, arg, sizeof(input));
input.path = kmalloc(input.path_len + 1, GFP_KERNEL);
if ( !input.path )
      return -ENOMEM;
error =copy_from_user(input.path,user_path, input.path_len);
```

- Weird security implications

```
get_user(len, oldlenp); /* 2.4.1/kernel/sysctl.c */
if (len > table->maxlen)
   len = table->maxlen;
copy_to_user(oldval, table->data, len);
```

Results for BSD 2.8 & 4 months of Linux

All bugs released to implementors; most serious fixed

Violation	Linux Bug	Linux Fixed	BSD Bug	BSD Fixed
Gain control of system	18	15	3	3
Corrupt memory	43	17	2	2
Read arbitrary memory	19	14	7	7
Denial of service	17	5	0	0
Minor	28	1	0	0
Total	125	52	12	12

Local bugs	109	12
Global bugs	16	0
Bugs from inferred ints	12	0
False positives	24	4
Number of checks	~3500	594

Talk Overview

- System-specific static analysis:
  - Correctness rules map clearly to concrete source actions
  - Check by making compilers aggressively system-specific
  - Easy: digest sentence fragment, write checker.
  - One person writes checker, imposed on all code.
  - Result: precise, immediate error diagnosis. Found errors in every system looked at
- Next: Belief analysis
  - Using programmer beliefs to infer state of system, relevant rules
  - Key: Find bugs without knowing truth.

Goal: find as many serious bugs as possible

- Problem: what are the rules?!?!
  - 100-1000s of rules in 100-1000s of subsystems.
  - To check, must answer: Must a() follow b()? Can foo() fail? Does bar(p) free p? Does lock l protect x?
  - Manually finding rules is hard. So don't. Instead infer what code believes, cross check for contradiction
- Intuition: how to find errors without knowing truth?
  - Contradiction. To find lies: cross-examine. Any contradiction is an error.
  - Deviance. To infer correct behavior: if 1 person does X, might be right or a coincidence. If 1000s do X and 1 does Y, probably an error.
  - Crucial: we know contradiction is an error without knowing the correct belief!

Cross-checking program belief systems

- MUST beliefs:

  Inferred from acts that imply beliefs code *must* have.

```
x = *p / z; // MUST belief: p not null
            // MUST: z != 0
unlock(l);  // MUST: l acquired
x++;        // MUST: x not protected by l
```

  Check using internal consistency: infer beliefs at different locations, then cross-check for contradction

- MAY beliefs: could be coincidental

  Inferred from acts that imply beliefs code *may* have

```
A();   A();  A();
......  ...   ...  // MAY: A() and B()
B();   B();  B();  // must be paired
```

```
B(); // MUST: B() need not
     // be preceded by A()
```

Trivial consistency: NULL pointers

- *p implies MUST belief:
  - **p is not null**
- A check (p == NULL) implies two MUST beliefs:
  - **POST: p is null on true path, not null on false path**
  - **PRE: p was unknown before check**
- Cross-check these for three different error types.
- Check-then-use (79 errors, 26 false pos)

```
/* 2.4.1: drivers/isdn/svmb1/capidrv.c */
if(!card)
  printk(KERN_ERR, "capidrv-%d: …", card->contrnr…)
```

Null pointer fun

- Use-then-check: 102 bugs, 4 false

```
/* 2.4.7: drivers/char/mxser.c */
struct mxser_struct *info = tty->driver_data;
unsigned flags;
if(!tty || !info->xmit_buf)
    return 0;
```

- Contradiction/redundant checks (24 bugs, 10 false)

```
/* 2.4.7/drivers/video/tdfxfb.c */
fb_info.regbase_virt = ioremap_nocache(...);
if(!fb_info.regbase_virt)
    return -ENXIO;
fb_info.bufbase_virt = ioremap_nocache(...);
if(!fb_info.regbase_virt) {
    iounmap(fb_info.regbase_virt);
```

Internal Consistency: finding security holes

- Applications are bad:
  - Rule: “do not dereference user pointer <p>”
  - One violation = security hole
  - Detect with static analysis if we knew which were “bad”
  - Big Problem: which are the user pointers???
- Sol’n: forall pointers, cross-check two OS beliefs
  - “*p” implies safe kernel pointer
  - “copyin(p)/copyout(p)” implies dangerous user pointer
  - Error: pointer p has both beliefs.
  - Implemented as a two pass global checker
- Result: 24 security bugs in Linux, 18 in OpenBSD
  - (about 1 bug to 1 false positive)

An example

- Still alive in linux 2.4.4:

```
/* drivers/net/appletalk/ipddp.c:ipddp_ioctl */
case SIOCADDIPDDPRT:
      return ipddp_create(rt);
case SIOCDELIPDDPRT:
      return ipddp_delete(rt);
case SIOCFCINDIPDDPRT:
   if(copy_to_user(rt, ipddp_find_route(rt),
                    sizeof(struct ipddp_route)))
        return -EFAULT;
```

Tainting marks "rt" as a tainted pointer, checker warns that rt is passed to a routine that dereferences it

2 other examples in same routine…

Cross checking beliefs related abstractly

- Common: multiple implementations of same interface.
  - **Beliefs of one implementation can be checked against those of the others!**
- User pointer (3 errors):

If one implementation taints its argument, all others must

```
foo_write(void *p, void *arg,…){
   copy_from_user(p, arg, 4);
   disable();
   … do something …
   enable();
   return 0;
}
```

```
bar_write(void *p, void *arg,…){
    *p = *(int *)arg;
    … do something …
    disable();
    return 0;
}
```

```
write_fp = foo_write;
…
write_fp = bar_write;
```

How to tell? Routines assigned to same function pointer

MAY beliefs

- Separate fact from coincidence? General approach:
  - **Assume MAY beliefs are MUST beliefs.**
  - **Check them**
  - **Count number of times belief passed check (S=success)**
  - **Count number of times belief failed check (F=fail)**
  - **Expect: valid beliefs = high ratio of S to F.**

  - **Use S and F to compute confidence that belief is valid.**
  - **Rank errors based on this confidence.**
  - **Go down list, inspecting until false positives are too high.**
- How to weigh evidence?

How to weigh MAY beliefs

- Wrong way: percentage. (Ignores population size)
  - Success=1, Failure=0, Percentage = 1/1 * 100= 100%
  - Success=999, Failure=10,Percentage =999/1000 = 99.9%
- A better way: "hypothesis testing."
  - Treat each check as independent binary coin toss
  - Pick probability p0 that coin "coincidently" comes up S.
  - For a given belief, compute how "unlikely" that it coincidently got S successes out of N (N=S+F) attempts

$$Z = (\text{observed} - \text{expected}) / \text{stderr}$$
$$= (S - N*p0) / \text{sqrt}(N*p0*(1-p0))$$

- HUGE mistake: pick T, where Z>T implies MUST
  - Becomes very sensitive to T.

Statistical: Deriving deallocation routines

- Use-after free errors are horrible.
  - Problem: lots of undocumented sub-system free functions
  - Soln: derive behaviorally: pointer "p" not used after call "foo(p)" implies MAY belief that "foo" is a free function
- Conceptually: Assume all functions free all arguments
  - (in reality: filter functions that have suggestive names)
  - Emit a "check" message at every call site.
  - Emit an "error" message at every use

foo(p);	foo(p);	foo(p);	bar(p);	bar(p);	bar(p);
*p = x;	*p = x;	*p = x;	p = 0;	p = 0;	*p = x;

  - Rank errors using z test statistic: z(checks, errors)
  - E.g., foo.z(3, 3) < bar.z(3, 1) so rank bar's error first
  - Results: 23 free errors, 11 false positives

Ranked free errors

```
kfree[0]: 2623 checks, 60 errors, z= 48.87
  2.4.1/drivers/sound/sound_core.c:sound_insert_unit:
      ERROR:171:178: Use-after-free of 's'! set by 'kfree`
  ...
kfree_skb[0]: 1070 checks, 13 errors, z = 31.92
  2.4.1/drivers/net/wan/comx-proto-fr.c:fr_xmit:
     ERROR:508:510: Use-after-free of 'skb'! set by 'kfree_skb`
  ...
[FALSE] page_cache_release[0] ex=117, counter=3, z = 10.3
dev_kfree_skb[0]: 109 checks, 4 errors, z=9.67
   2.4.1/drivers/atm/iphase.c:rx_dle_intr:
      ERROR:1321:1323: Use-after-free of 'skb'! set by 'dev_kfree_skb_any`
   ...
cmd_free[1]: 18 checks, 1 error, z=3.77
    2.4.1/drivers/block/cciss.c:667:cciss_ioctl:
      ERROR:663:667: Use-after-free of 'c'! set by 'cmd_free[1]'
drm_free_buffer[1] 15 checks, 1 error, z = 3.35
     2.4.1/drivers/char/drm/gamma_dma.c:gamma_dma_send_buffers:
         ERROR:Use-after-free of 'last_buf'!
[FALSE] cmd_free[0] 18 checks,  2 errors, z = 3.2
```

Recall: deterministic free checker

```
sm free_checker {
 state decl any_pointer v;
 decl any_pointer x;

 start: { kfree(v); } ==> v.freed
  ;
 v.freed:
    { v != x } || { v == x }
                  ==> { /* do nothing */ }
 | { v } ==> { err("Use after free!"); }
    ;
}
```

A statistical free checker

```
sm free_checker local {
 state decl any_pointer v;
 decl any_fn_call call;
 decl any_pointer x;

 start: { call(v) } ==> v.freed,
       {
           v.data = call.name();
           printf("checking [POP=%s]", v.data);
       }
   ;
 v.freed:
    { v != x } || { v == x }  ==> { /* do nothing */ }
 | { v } ==> { err("Use after free! [FAIL=%s]", v.data); }
    ;
}
```

A bad free error

```
/* drivers/block/cciss.c:cciss_ioctl  */
if (iocommand.Direction == XFER_WRITE){
   if (copy_to_user(...)) {
        cmd_free(NULL, c);
        if (buff != NULL) kfree(buff);
        return( -EFAULT);
  }
}
if (iocommand.Direction == XFER_READ) {
     if (copy_to_user(...)) {
         cmd_free(NULL, c);
         kfree(buff);
     }
}
cmd_free(NULL, c);
if (buff != NULL) kfree(buff);
```

Deriving "A() must be followed by B()"

- "a(); … b();" implies MAY belief that a() follows b()
  - Programmer may believe a-b paired, or might be a coincidence.
- Algorithm:
  - Assume every a-b is a valid pair (reality: prefilter functions that seem to be plausibly paired)
  - Emit "success" for each path that has a() then b()
  - Emit "error" for each path that has a() and no b()

foo(p, …) bar(p, …);	→ "check foo-bar"	x(); y();	→ "check x-y"	foo(p, …); …	→ "error:foo, no bar!"

  - Rank errors for each pair using the test statistic $z(foo.success, foo.error) = z(2, 1)$
- Results: 23 errors, 11 false positives.

Checking derived lock functions

- Evilest:
```
/* 2.4.1: drivers/sound/trident.c:
                        trident_release:
    lock_kernel();
    card = state->card;
    dmabuf = &state->dmabuf;
    VALIDATE_STATE(state);
```
- And the award for best effort:
```
/* 2.4.0:drivers/sound/cmpci.c:cm_midi_release: */
lock_kernel();
if (file->f_mode & FMODE_WRITE) {
      add_wait_queue(&s->midi.owait, &wait);
         ...
         if (file->f_flags & O_NONBLOCK) {
             remove_wait_queue(&s->midi.owait, &wait);
             set_current_state(TASK_RUNNING);
             return -EBUSY;
… unlock_kernel();
```

Statistical: deriving routines that can fail

- Traditional:
  - **Use global analysis to track which routines return NULL**
  - **Problem: false positives when pre-conditions hold, difficult to tell statically ("return p->next"?)**
- Instead: see how often programmer checks.
  - **Rank errors based on number of checks to non-checks.**
- Algorithm: Assume *all* functions can return NULL
  - **If pointer checked before use, emit "check" message**
  - **If pointer used before check, emit "error"**

```
P = foo(…);    p = bar(…);        p = bar(…);        p = bar(…);        p = bar(…);
*p = x;        If(!p) return;     If(!p) return;     If(!p) return;     *p = x;
               *p = x;            *p = x;            *p = x;
```

  - **Sort errors based on ratio of checks to errors**
- Result: 152 bugs, 16 false.

The worst bug

- Starts with weird way of checking failure:

```
/* 2.3.99: ipc/shm.c:1745:map_zero_setup */
if (IS_ERR(shp = seg_alloc(...)))
    return PTR_ERR(shp);

static inline long IS_ERR(const void *ptr)
 { return (unsigned long)ptr > (unsigned long)-1000L; }
```

- So why are we looking for “seg_alloc”?

```
/* ipc/shm.c:750:newseg: */
if (!(shp = seg_alloc(...))
  return -ENOMEM;
id = shm_addid(shp);
```

```
int ipc_addid(…* new…) {
    ...
    new->cuid = new->uid =…;
    new->gid = new->cgid = …
    ids->entries[id].p = new;
```

Summary

- Effective static analysis of real code
  - **Write small extension, apply to code, find 100s-1000s of bugs in real systems**
  - **Result: Static, precise, immediate error diagnosis**
  - **One person writes, imposes on all code.**

- Belief analysis: broader checking
  - **Using programmer beliefs to infer state of system, relevant rules**
  - **Key feature: find errors without knowing truth**
- Found lots of serious bugs everywhere.

Assertion: Soundness is often a distraction

- Soundness: Find all bugs of type X.
  - **Not a bad thing. More bugs good.**
  - **BUT: can only do if you check weak properties.**
- What soundness really wants to be when it grows up:
  - **Total correctness: Find all bugs.**
  - **Most direct approximation: find as many bugs as possible.**
- Opportunity cost:
  - **Diminishing returns: Initial analysis finds most bugs**
  - **Spend time on what gets the next biggest set of bugs**
  - **Easy experiment: bug counts for sound vs unsound tools.**
- Soundness violates end-to-end argument:
  - **"It generally does not make much sense to reduce the residual error rate of one system component (property) much below that of the others."**

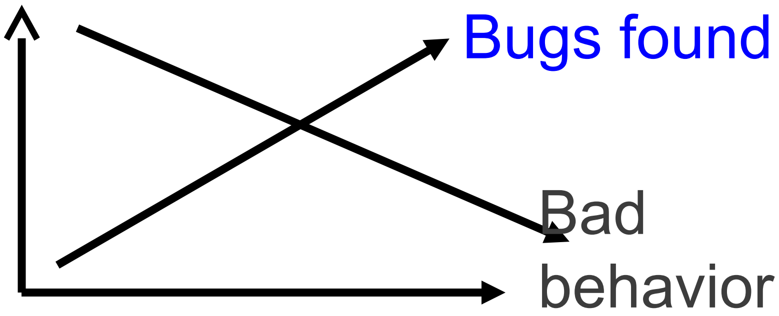
Static vs dynamic bug finding

- Static: precondition = compile (some) code.
  - **All paths + don't need to run + easy diagnosis.**
  - **Low incremental cost per line of code**
  - **Can get results in an afternoon.**
  - **10-100x more bugs.**
- Dynamic: precondition = compile all code + run
  - **What does code do? How to build? How to run?**
  - **Runs code, so can check implications.**
  - **Good: Static detects ways to cause error, dynamic can check for the error itself.**
- Result:
  - **Static better at checking properties visible in source, dynamic better at properties implied by source.**

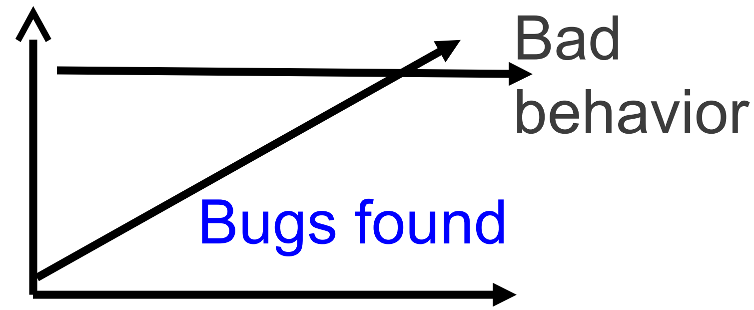
Open Q: how to get the bugs that matter?

- Myth: all bugs matter and all will be fixed
  - *FALSE*
  - Find 10 bugs, all get fixed. Find 10,000…
- Reality
  - All sites have many open bugs (observed by us & PREfix)
  - Myth lives because state-of-art is so bad at bug finding
  - What users really want: The 5-10 that "really matter"
- General belief: bugs follow 90/10 distribution
  - Out of 1000, 100 (10? or 1?) account for most pain.
  - Fixing 900+ waste of resources & may make things worse
- How to find worst? No one has a good answer to this.
  - Possibilities: promote bugs on executed paths or in code people care about, …

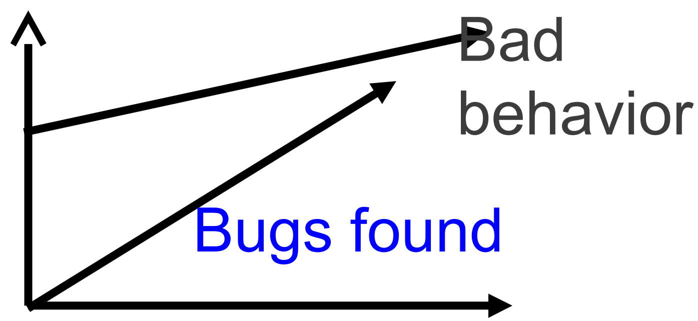
Open Q: Do static tools really help?



The optimistic hope



The null hypothesis



An Ugly Possibility

Danger: Opportunity cost.

Danger: Deterministic canary bugs to non-deterministic.

Laws of static bug finding

- Vacuous tautologies that imply trouble
  - **Can't find code, can't check.**
  - **Can't compile code, can't check.**
- A nice, balancing empirical tautology
  - **If can find code**
  - **AND checked system is big**
  - **AND can compile (enough) of it**
  - **THEN: will *always* find serious errors.**
- A nice special case:
  - **Check rule never checked? Always find bugs. Otherwise immediate kneejerk: what wrong with checker???**