How to find lots of bugs by checking program belief systems

Dawson Engler
David Chen, Seth Hallem, Ben Chelf, Andy Chou
Stanford University

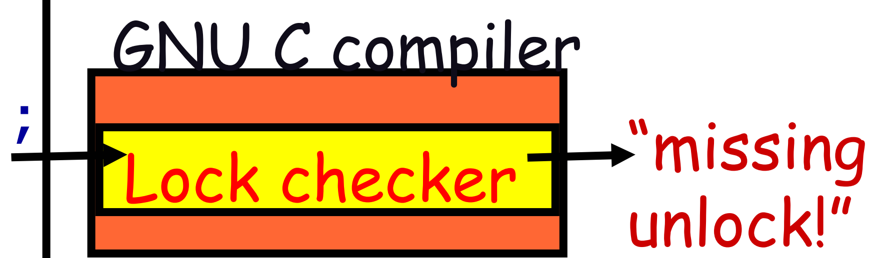
Presentedy by Baohua Wu

Context: finding OS bugs w/ compilers

- Systems have many ad hoc correctness rules
  - "acquire lock l before modifying x", "cli() must be paired with sti()," "don't block with interrupts disabled"
  - One error = crashed machine
- If we know rules, can check with extended compiler
  - Rules map to simple source constructs
  - Use compiler extensions to express them

Linux fs/proc/inode.c

```
lock_kernel();
if (!de->count) {
  printk("free!\n");
  return;
}
unlock_kernel();
```



Nice: scales, precise, statically find 1000s of errors

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 contradiction

- MAY beliefs: could be coincidental

  Inferred from acts that imply beliefs code *may* have

```
A();   A();   A();  A();
                                      B(); // MUST: B() need not
…      …      …     …                      // be preceded by A()
B();   B();   B();  B();  // MAY: A() and B()
                          // must be paired
```

  Check as MUST beliefs; rank errors by belief confidence.

Two techniques

- Internal Consistency
  - **Must beliefs**
- Statistical Analysis
  - **May beliefs**

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(...);
/* [META: meant fb_info.bufbase_virt!] */
if(!fb_info.regbase_virt) {
   iounmap(fb_info.regbase_virt);
```

Redundancy checking

- Assume: code supposed to be useful

  Useless actions = conceptual confusion. Like type systems, high level bugs map to low-level redundancies

- Identity operations: "x = x", "1 * y", "x & x", "x | x"

```
/*  2.4.5-ac8/net/appletalk/aarp.c  */
 da.s_node = sa.s_node;
 da.s_net = da.s_net;
```

- Assignments that are never read:

```
for(entry=priv->lec_arp_tables[i];entry != NULL; entry=next){
   next = entry->next;
   if (…)
      lec_arp_remove(priv->lec_arp_tables, entry);
   lec_arp_unlock(priv);
   return 0;
}
```

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 SIOFCINDIPDDPRT:
      if(copy_to_user(rt, ipddp_find_route(rt),
                          sizeof(struct ipddp_route)))
            return -EFAULT;
```

Tainting marks "rt" as a tainted pointer, checking 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

  How to tell? Routines assigned to same function pointer

```
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;
}
```

More general: infer execution context, arg preconditions…

Interesting q: what spec properties can be inferred?

Handling MAY beliefs

- MUST beliefs: only need a single contradiction
- MAY beliefs: need many examples to separate fact from coincidence
- Conceptually:

  Assume MAY beliefs are MUST beliefs

  Record every successful check with a "check" message

  Every unsuccessful check with an "error" message

  Use the test statistic to rank errors based on ratio of checks (n) to errors (err)

  $$z(n, err) = ((n-err)/n-p0)/sqrt(p0*(1-p0)/n)$$

  Intuition: the most likely errors are those where n is large, and err is small.

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

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);
```

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 = x;
```

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

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

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

```
p = bar(…);
*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;
```

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 "check" 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.check, 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();
```

Summary: Belief Analysis

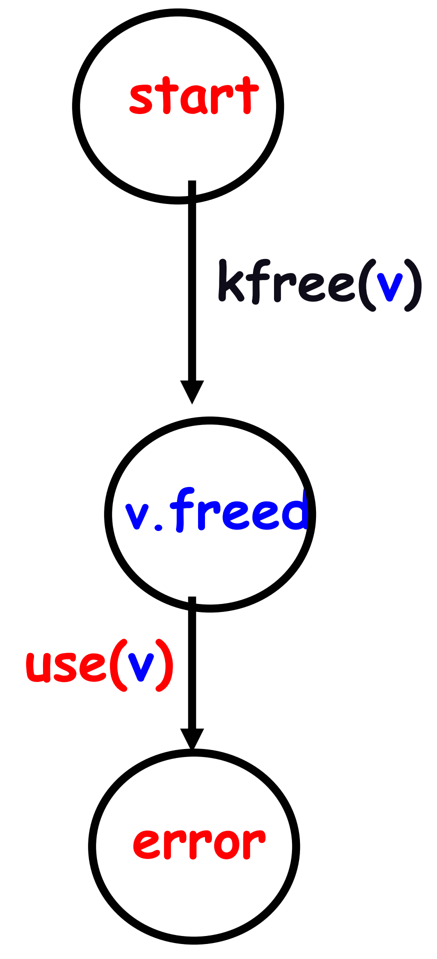
- Key ideas:
  - Check code beliefs: find errors without knowing truth.
  - Beliefs code MUST have: Contradictions = errors
  - Beliefs code MAY have: check as MUST beliefs and rank errors by belief confidence

- Secondary ideas:
  - Check for errors by flagging redundancy.
  - Analyze client code to infer abstract features rather than just implementation.
  - Spec = checkable redundancy. Can use code for same.

Example 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 } ➔ { /* suppress fp */ }
   | { v } ➔ { err("Use after free!");
   ;
}
```



Example inferring free checker

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

 start: { call(v) } ➔ {
      char *n = mc_identifier(call);
      if(strstr(n, “free”) || strstr(n, “dealloc”) || … ) {
         mc_v_set_state(v, freed);
         mc_v_set_data(v, n);
         note(“NOTE: %s”, n);
      }
  };
 v.freed: { v == x } | { v != x } ➔ { /* suppress fp */ }
   | { v } ➔ { err(“Use after free %s!”, mc_v_get_data(v));
   ;
```

Conclusion

- Two Techniques:
  - **internal consistency**
  - **statistical analysis**
- Found hundreds of bugs automatically in real system code:
  - **Linux**
  - **OpenBSD**