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

Dawson Engler

Andy Chou, Ben Chelf, Seth Hallem, Ken Ashcraft

Stanford University

Checking systems software

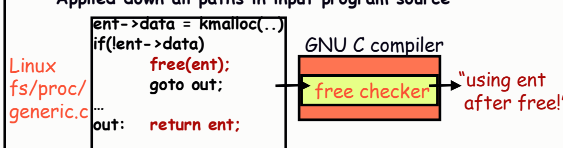
- Systems software has many ad-hoc restrictions:
  - "acquire lock L before accessing shared variable X"
  - "do not allocate large variables on 6K kernel stack"
- Error = crashed system. How to find errors?
  - Formal verification
    - + rigorous
    - - costly + expensive. *Very* rare to do for software
  - Testing:
    - + simple, few false positives
    - - requires running code: doesn't scale & can be impractical
  - Manual inspection
    - + flexible
    - - erratic & doesn't scale well.
  - What to do??

Another approach

- Observation: rules can be checked with a compiler
  - scan source for "relevant" acts check if they make sense
  - E.g., to check "disabled interrupts must be re-enabled:" scan for calls to disable()/enable(), check that they match, not done twice
- Main problem:
  - compiler has machinery to automatically check, but not knowledge
  - implementor has knowledge but not machinery
- Metacompilation (MC):
  - give implementors a framework to add easily-written, system-specific compiler extensions

Metacompilation (MC)

- Implementation:
  - Extensions dynamically linked into GNU gcc compiler
  - Applied down all paths in input program source



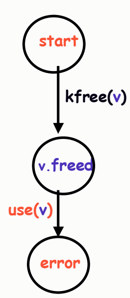
- Scalable: handles millions of lines of code
- Precise: says exactly what error was
- Immediate: finds bugs without having to execute path
- Effective: 1500+ errors in Linux source code

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

```
/* 2.4.4:drivers/isdn/isdn_ppp.c */
if (!(ippp_table[i] = kmalloc(…))
      for (j = 0; j < i; j++)
             kfree(ippp_table[i]);
```

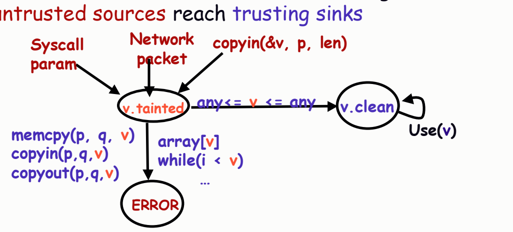


Talk Overview

- Overview: metacompilation [OSDI '00, ASPLOS '00]
- Next: three examples
  - Temporal rule: sanitize user data before use
  - Contextual rule: don't block with interrupts off
  - Moving dynamic to static: assert checking
- Broader checking: Inferring rules [SOSP '01]
  - Find inconsistencies in program belief systems
  - Great lever: find errors without knowing truth
- Deeper checking [ISCA '01]
  - Extract formal model from raw C code
  - Run through model checker

"X before Y": sanitize integers before use

- Security: OS must check user integers before use
- MC 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.

- No checks

```
2.4.5-ac8/drivers/usb/se401.c:
copy_from_user(&frame, arg, sizeof(int));
ret=se401_newframe(se401, frame);
se401->frame[frame].grabstate = FRAME_UNUSED;
```

- Unexpected overflow

```
/* 2.4.9: drivers/net/wan/farsync.c */
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)
```

- Weird security implications

```
/* 2.4.1/kernel/sysctl.c:455:do_sysctl_strategy */
get_user(len, oldlenp);
if (len > table->maxlen)
     len = table->maxlen;
copy_to_user(oldval, table->data, len);
```

Some more 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);
```

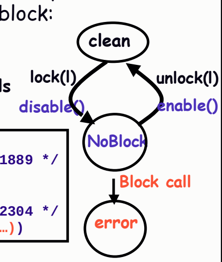
- A more subtle overflow

```
/* 2.4.9-ac7/fs/intermezzo/psdev.c:presto_psdev_ioctl */
error = copy_from_user(&input, (char *)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);
```

"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

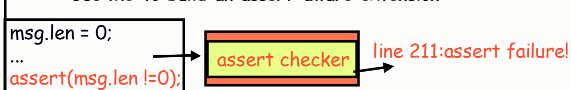



```
/* drivers/net/pcmcia/wavelan_cs.c */
spin_lock_irqsave (&lp->lock, flags);/* 1889 */
switch(cmd)
...
  case SIOCGIWPRIV:                     /* 2304 */
    if(copy_to_user(wrq->u.data.pointer, …))
```

Heavy clustering:
net/atm: 152 checks, 22 bugs (exp 1.9) P =3.1x10^-15
drivers/i2o: 692 checks, 35 bugs (exp 8.8) P= 2.6x10^-10

Example: statically checking assert

- Assert(x) used to check "x" at runtime. Abort if false
  - **compiler oblivious, so cannot analyze statically**
  - **Use MC to build an assert-aware extension**



- Result: found 5 errors in FLASH.
  - **Common: code cut&paste from other context**
  - **Manual detection questionable: 300-line path explosion between violation and check**

General method to push dynamic checks to static

Summary

- Metacompilation:
  - **Correctness rules map clearly to concrete source actions**
  - **Check by making compilers aggressively system-specific**
  - **Easy: digest sentence fragment, write checker.**
  - **Result: precise, immediate error diagnosis**
  - **As outsiders found errors in every system looked at**
  - **1000s bugs, many capable of crashing system**
- Next:
  - **Inferring errors by checking program belief systems**
  - **Deeper checking**

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
…     …     …    …    // MAY: A() and B()          // be preceded by A()
B();  B();  B(); B(); // must be paired
```

  - Check as MUST beliefs; rank errors by belief confidence.

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);
```

Aside: redundancy checking

- Assume: code supposed to be useful
  - Like types: 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 never read (126 bugs, 26 fp, 1.8K uninsp):

```
/* 2.4.5-ac8/net/decnet/af_decnet.c:dn_wait_run */
do {
  if (signal_pending(current)) {
     err = -ERESTARTSYS;
     break;
  }
  ...
} while(scp->state != DN_RUN);
return 0;
```

Redundancy checking

- Dead code (66 bugs, 26 false):

```
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;
}
```

- Detect incomplete specifications:
  - Detect missed sinks in range checker: flag when data read from untrusted source, sanitized, but then not used for any dangerous operation.
  - Lock checker: critical section with no shared state, lock with no bound variables

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 SIOCFINDIPDDPRT:
        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
  - 3 other examples in same routine
- Can combine with earlier range checker (12 errors):

```
/* 2.4.9/drivers/telephony/ixj.c:ixj_ioctl */
  case IXJCTL_INIT_TONE:
    copy_from_user(&ti, (char *) arg, sizeof(ti)); ...
  case IXJCTL_INTERCOM_START:
    ... ixj[arg]->intercom = board;
```

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,…){        bar_write(void *p, void *arg,…){
   copy_from_user(p, arg, 4);              *p = *(int *)arg;
   disable();                              … do something …
   … do something …                        disable();
   enable();                               return 0;
   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. General approach:
  - Assume MAY beliefs are MUST beliefs & check them
  - Count number of times belief passed check
  - Count number of times belief failed check
  - 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.
  - BAD idea: pick threshold t, if z(n,c) > t treat as MUST

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
```

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);
```

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));
   ;
```

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;      *p = x;        if(!p) return;
               *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;
```

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, …)      → "check      |  x();  → "check  |  foo(p, …);  → "error:foo,
bar(p, …);       foo-bar"    |  y();    x-y"    |  …              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:
  - High-level errors map to low-level redundancies
  - Specification is a checkable redundancy: code has many redundant uses that can be leveraged in same way.
  - Can use statistical ranking to help traditional analysis!

Deeper checking

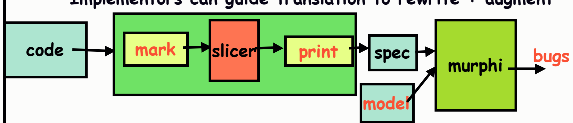
- We'd like real assurances of correctness
  - Verification? Coders don't write docs, much less specs…
  - Observation: spec clearly mirrors code. Auto-extract!

```
void PILocalGet(void) {
  /* ... Boilerplate setup code ... */
  nh.len = LEN_CACHELINE;
  if (!hl.Pending) {
    if (!hl.Dirty) {
      /* ... 37 lines deleted ... */
      ASSERT(!hl.IO);
      // The commented out ASSERT is
      // true 99.99% of the time,
      // but is not always
      // ASSERT(hl.Local);
      /*... deleted 15 lines ... */
      PI_SEND(F_DATA, F_FREE, F_SWAP,
              F_NOWAIT, F_DEC, 1);
      hl.Local = 1;
```

```
Rule "PI Local Get"
Cache.State = Invalid
& ! Cache.Wait
& ! DH.Pending
& ! DH.Dirty  ==>
 Begin
  Assert !DH.Local;
  DH.Local := true;
  CC_Put(Home, Memory);
EndRule;
```

Overview: Automatic extraction

- Key: abstract models are clearly embedded in code
  - Implementors use extensions to mark these features
  - System rips them out & translates to formal model
  - Implementors can guide translation to rewrite + augment



- Example: verifying FLASH protocol
  - Hard core, asm strewn C.
  - Tested for 6+ years, manually "verified"
  - We found 8 errors.
  - Bonus: Automatically found bugs in manual spec (it's code)

A simple abstraction function

```
sm len slicer {
  /* wildcard variables for pattern matching */
  decl any_expr  type, data, keep, swp, wait, nl;

  /* match all uses of the length field. */
  pat length = { HG_header.nh.len } ;
  /* match sends */
  pat sends =
      { NI_SEND(type, data, keep, swp, wait, nl) }
    | { PI_SEND(type, data, keep, swp, wait, nl) }
    ;
  /* match accesses to directory entries */
  pat entries = { HG_h.hl.Local } | { HG_h.hl.Dirty } ;
 /* mark patterns for MC slicer */
  all: length | sends | entries ==> { mgk_tag(mgk_s); } ;
}
```

Related work

- Tool-based checking
  - PREfix/PREfast
  - Slam
  - ESP
- Higher level languages
  - TypeState, Vault
  - Foster et al's type qualifier work.
- Derivation:
  - Houdini to infer some ESC specs
  - Ernst's Daikon for dynamic invariants
  - Larus et al dynamic temporal inference
- Spec extraction
  - Bandera
  - Slam

Summary

- MC: 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
- Belief analysis: broader checking
  - Infer system rules and state using code beliefs
  - Key feature: find errors without knowing truth.
- Model extraction: deeper checking
  - Common: abstract models clearly embedded in C code
  - Automatically extract these using extensions
  - Model check result