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??
"X before Y": sanitize integers before use

- Security OS must sanitize integers before use
- MC checker: Warn when unchecked integers from untrusted sources reach trusting sinks

System call parameter
Network packet

copyin(&p, v, len)

Use(v)

Global: simple to retarget (text file with 2 srcs & 12 sinks)
Linux: 125 errors, 24 false: BSD: 12 errors, 4 false

Some more big, gaping security holes.

- Remote exploit, no checks

```c
/* 2.4.3/drivers/ixsl/ixs2000/capi.c:shcapi_dispatch */
iosd_strl cmd;
while ((ssh = ssh_decode(card->rcvq))) {
  mesg = ssh->data;
  ... 
  memcpy(cmd.parm.connect->addr_num, mesg->mesg_connect->addr.len = 1);
}
```

- A more subtle overflow

```c
/* 2.4.3/flo/internmem/pddev.c:preto_pddev_ioctl */
int error = copy_from_user(input, (char *)arg, sizeof(input));
... 
input_path = kstrlcpy(input_path_len + 1, GFP_KERNEL);
if (!input_path)
  return -ENOMEM;
error = copy_from_user(input_path.user_path, input_path_len);
```

Some big, gaping security holes.

- No checks

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

- Unexpected overflow

```c
/* 2.4.3/drivers/net/wn/asyncn.c */
copy_from_user(wrthdr, addr, sizeof wrthdr);
if (wrthdr.size + wrthdr.offset > FST_MEMSIZE)
  return -ENOMEM;
copy_from_user(card->mem[wrthdr.offset, data, wrthdr.size];
```

- Weird security implications

```c
/* 2.4.1/kernel/asym1.c:455:do_async1_strategy */
get_user(len, oldlen);
if (len > table-maxlen)
  len = table-maxlen;
copy_to_user(oldval, table->data, 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 to
    - Linux: 123 errors, 8 false pos
    - BSD: 152 checks, 22 bugs (exp 1.9) P = 3.1x10^-15
    - drivers/usb: 692 checks, 35 bugs (exp 8.6) 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

```c
msg.len = 0;
... 
assert(msg.len != 0);
```

- 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() free p? Does lock() protect x?
- 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:
    \[ x = \text{"p"} / x; \] // MUST belief: p not null
    \[ x = \text{"z"} / 0; \] // MUST: z = 0
    unlock(); // MUST: l acquired
    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
  \[ \text{AO} : \text{AO} \rightarrow \text{BQ} ; \] // MUST: BQ need not
  \[ \text{BQ} : \text{BQ} \rightarrow \text{AO} ; \] // MAY: AO and BQ
  Check as MUST beliefs: rank errors by belief confidence.

Trivial consistency: NULL pointers

- *p implies MUST belief: p is 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/lsdn/evmsl/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/mexer.c */
 struct mexer_struc *info = tty->driver_data;
 if(!tty) |
   info->wait_buf)
   return 0;`

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

`/* 2.4.7: drivers/video/tidxfb.c */
 if(!fb_info.regbase_virt)
   return -ENOD;`

Aside: redundancy checking

- Assume: code supposed to be useful
- Like types: high-level bugs map to low-level redundancies
- Identity operations: \[ x = x^n, 1 \leq y \leq x, x \& x = x \]

`/* 2.4.5:asl/net/desct/vd4/demct.c */
 da_x_code = aa_x_code;
 da_x_net = da_x_code;`

- Assignments never read (126 bugs, 26 fp, 1.8K uninsp):

`/* 2.4.5-ssl/net/desct/vd4/demct.c */
 do {
   if (signal_pending(current)) {
     err = ERESTARTSYS;
     break;
   }
   ...`

Redundancy checking

- Dead code (66 bugs, 26 false):

`for(entry=priv->lec_app_tables;entry!=NULL;entry=entry->next){`
  `if (...)`
  `lec_app_remove(priv->lec_app_tables, entry);`
  `lec_app_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???
- Son: for all pointers, cross-check two OS beliefs
  - “[p]” implies safe kernel pointer
  - “copy(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)

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

```c
foo_write(void *p, void *arg,...)
  copy_from_user(p, arg, 4);
  disable();
  ...do something ...
  enable();
  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):
    $$z(n, err) = \left(\frac{\text{err}}{\sqrt{n+\text{err}}} - \frac{1}{2}\right) \times \sqrt{n+\text{err}}$$
  - 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
  - Same: 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

```c
foo(p); foa(p); foa(p); bar(p); bar(p);
```

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

Ranked free errors

<table>
<thead>
<tr>
<th>ERRORS</th>
<th>DETAILS</th>
</tr>
</thead>
<tbody>
<tr>
<td>kfree[0]: 2623 checks, 60 errors, z = 48.87</td>
<td></td>
</tr>
<tr>
<td>2.4.1/drivers/sound/sound_core.c: sound_insert_unit:</td>
<td></td>
</tr>
<tr>
<td>ERROR:17:17: Use-after-free of ‘s’ set by ‘kfree’</td>
<td></td>
</tr>
<tr>
<td>kfree_skb[0]: 1070 checks, 13 errors, z = 31.92</td>
<td></td>
</tr>
<tr>
<td>2.4.1/drivers/net/mac/mac-ipv6-fr-c: fr_unit:</td>
<td></td>
</tr>
<tr>
<td>ERROR:550:510: Use-after-free of skb’ skb’ set by ‘kfree_skb’</td>
<td></td>
</tr>
<tr>
<td>![FALSE] page_cache_release[0] sm=17, counterr3, z = 10.3</td>
<td></td>
</tr>
<tr>
<td>![ERROR] kfree_skb[0] t: 109 checks, 6 errors, x = 9.67</td>
<td></td>
</tr>
<tr>
<td>2.4.1/drivers/ata/phase.c:ra_disint:</td>
<td></td>
</tr>
<tr>
<td>ERROR:1311:1325: Use-after-free of skb set by ‘dev_kfree_skb.asm’</td>
<td></td>
</tr>
<tr>
<td>![FALSE] cmd_free[1] 18 checks, 1 error, x = 3.77</td>
<td></td>
</tr>
<tr>
<td>2.4.1/drivers/block/queue.c:647:9: Use-after-free of ‘c’ set by ‘cmd_free[1]’</td>
<td></td>
</tr>
<tr>
<td>![ERROR] dm_free_buffer[1] 15 checks, 1 error, x = 3.55</td>
<td></td>
</tr>
<tr>
<td>2.4.1/drivers/block/drm/drm_buffer.c:25:3: Use-after-free of ‘last_buf’</td>
<td></td>
</tr>
<tr>
<td>![FALSE] cmd_free[1] 18 checks, 2 errors, x = 3.2</td>
<td></td>
</tr>
</tbody>
</table>
A bad free error

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

Example inferring free checker

```c
sm_free_checker ()
state decl any_pointer v;
state decl any_pointer x;
state decl any_fn_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"
  Sort errors based on ratio of checks to errors

Result: 152 bugs, 16 false.

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() 
  Rank errors for each pair using the test statistic
  z(check, error) = z(t, 1)

Results: 23 errors, 11 false positives.

The worst bug

- Starts with weird way of checking failure:
  /* 2.3.89: ipc/shm.c:1745:map_extra_setup */
  if (IS_ERR(shm = seg_alloc(...)))
  return PTR_ERR(shm);

- So why are we looking for "seg_alloc"?
  /* ipc/shm.c:750:newseg */
  if (! (shp = seg_alloc(...)));
  int ipo_addid(... 
  id = shm_addid(shp);
  new->uid = new->uid =;
  new->gid = new->gid =
  id->entries[id].p = new;
```

Checking derived lock functions

- Evilest:
  /* 2.4.1: drivers/sound/sound:trident.c:
  trident_release:
  lock_kernel();
  card = state->card;
  dmbuf = state->dmbuf;
  VALIDATE_STATE(state);

- And the award for best effort:
  /* 2.4.0: drivers/sound/empoi.c:cm_midi_release: */
  lock_kernel();
  if (file->f_mode & FMODE_WRITE) {
    add_wait_queue(is->midi_cwait, wait);
    ...
  if (file->f_flags & O_NONBLOCK) {
    remove_wait_queue(is->midi_cwait, 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!

```c
void PLocalGet(void) {
    /* ... Boilerplate setup code ... */
    Cache.State = Invalid
    nh.len = LEN_CACHELINE;
    ! Cache.Wait
    if (!hl.Pending) {
        ! DH.Pending
        if (hl.Dirty) {
            // 37 lines deleted ...
            ! DH.Dirty
            ASSERT(!hl.IO);
            // The commented out ASSERT is true 99.99% of the time,
            // but is not always
            // ASSERT(hl.Local);
            // Begin
            // The commented out ASSERT is true;
            DH.Local = true;
            CC.Put(Some, Memory);
        }
        ASSERT(!hl.Local);
        /* ... deleted 15 lines ... */
        PI_SEND(F_DATA, F_FREE, F_SWAP, F_NOWAIT, F_DEC, 1);
    } else
        hl.Local = 1;
}
```

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

#### Example: verifying FLASH protocol
- Hard core, asm stream 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
```c
asm len slicer {
    /* wildcard variables for pattern matching */
    decl any_expr type, data, keep, swap, wait, nl;
    /* match all uses of the length field. */
    pat length = ( HG_header.nh.len ) ;
    /* match sends */
    pat sends =
        ( NL_SEND(type, data, keep, swap, wait, nl) )
        | ( PI_SEND(type, data, keep, swap, wait, nl) )
    ;
    /* match accesses to directory entries */
    pat entries = ( HGh.hl.Local ) | ( HGh.hl.Dirty ) ;
    /* mark patterns for MC slicer */
    all: length | sends | entries => ( mpg_tag(mpg_a); ) ;
}
```

### Related work
- **Tool-based checking**
  - PReFx/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