Finding bugs with system-specific static analysis

Dawson Engler
Ken Ashcraft, Ben Chelf, Andy Chou, Seth Hallem,
Yichen Xie, Junfeng Yang
Stanford University

Context: finding bugs w/ static analysis

- Systems have many ad hoc correctness rules
  "sanitize user input before using it"; "check permissions
  before doing operation X"
  One error = compromised system
- If we know rules, can check with extended compiler
  Rules map to simple source constructs
  Use compiler extensions to express them

A bit more detail

```c
struct free_checker {
  state decl any_pointer v;
  decl any_pointer x;
  start: { kfree(v); } --> v.freed;
  v.freed:
    { v != x } | | ( v == x )
    | v } --> { /* do nothing */ }
  use(v) --> error;
}
```

A quick analysis example

```c
foo(int *x) {
  kfree(x);
  if(y)
    freeit(x);
}
bar(int *y) {
  freeit(y);
  /* y */
}
```

A quick analysis example

```c
foo(int *x) {
  kfree(x);
}
bar(int *y) {
  freeit(y);
  /* y */
}
```
A quick analysis example

```c
foo(int *x) {
    freeit(x);
}
```

ERROR: use after free!
A quick analysis example

```c
freenit(int *z) {
    x: z->freed
}
foo(int *x) {
    x: free(x);
    y: freeit(x);
    z: freeit(z);
    if (y)
        ...
}
bar(int *y) {
    t: freeit(y);
}
frqeeit(int *x) {
    v: x->freed
}
```

```
ERROR: use after free!
```

"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

```
/* 2.4.9-ae7/fs/interrupts/psdev.c */
error = copy_from_user(input, (char *)arg2, sizeof(input));
if ((input.path) )
    return -EINVAL;
error = copy_from_user(input.path, user_path, input.path_len);
```

```
while ((skb = skb_dequeue(skb->queue)) ) {
    mesg = skb->data;
    ...
    memcpy(cmd.parm.setupt.phone, mesg->mesg.connect_ind.addr.num,
            mesg->mesg.connect_ind.addr.len - 1);
```

Talk Overview

- Metacompletion [OSDI’00, ASPLOS’00]:
  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. Found errors in every system look
  - Next: A deeper look at a security checker [S&P'01]
  - Flags when untrusted input is not sanitized before use
  - Broader checking: Inferring rules [SOSP ’01]
    - Great lever: find errors without knowing truth
  - Some practical issues

Results for BSD 2.8 & 4 months of Linux

<table>
<thead>
<tr>
<th>Violation</th>
<th>Bug Fixed</th>
<th>BSD Bug Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gain control of system</td>
<td>18 15 3 3</td>
<td></td>
</tr>
<tr>
<td>Corrupt memory</td>
<td>43 17 2 2</td>
<td></td>
</tr>
<tr>
<td>Read arbitrary memory</td>
<td>19 14 7 7</td>
<td></td>
</tr>
<tr>
<td>Denial of service</td>
<td>17 5 0 0</td>
<td></td>
</tr>
<tr>
<td>Minor</td>
<td>28 1 0 0</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>125 52 12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Violation</th>
<th>Bug Fixed</th>
<th>BSD Bug Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local bugs</td>
<td>109 12</td>
<td></td>
</tr>
<tr>
<td>Global bugs</td>
<td>16 0</td>
<td></td>
</tr>
<tr>
<td>Bugs from inferred ints</td>
<td>12 0</td>
<td></td>
</tr>
<tr>
<td>False positives</td>
<td>24 4</td>
<td></td>
</tr>
<tr>
<td>Number of checks</td>
<td>~3500 594</td>
<td></td>
</tr>
</tbody>
</table>

Many other checkers

- Concurrency
  - Deadlock
  - Missing unlock or enable interrupt call
  - Prototype race detection
- Memory errors
  - Null pointer bug
  - Not checking allocation result
  - Using freed pointers
  - Not deallocating memory on return paths.
- General temporal properties
  - A then B, A then NOT B, etc

Security checkers

- Unsafe uses of unset input: integers, strings, pointers
- Exploitable errors
- Statistically inferring
  - Parsed functions
  - Functions that deallocate arguments
  - Functions that return null pointers
  - Variables that are unsafe
  - Which locks protect which variables
Talk Overview

- Metacompilation
  Correctness rules map clearly to concrete source actions
  Check by making compilers aggressively system-specific
  One person writes checker, imposed on all code

- Next: Belief analysis
  Using programmer beliefs to infer state of system, relevant rules

- Managing false positives
- Some experience

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() 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 I
  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 facts that imply beliefs code "must" have.
  x = p / z: // MUST belief: p not null
  unlock(): // 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 facts that imply beliefs code "may" have
  A0: A0 (A0): B0: // MUST: B0 need not
  B0: B0: B0: // must be paired
  Check as MUST beliefs: rank errors by belief confidence.

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:

```c
/* drivers/net/appltalk/ipddp.c:ipddp_ioctl */
case SIOCFindIPDPart:
  if(copy_to_user(g, ipddp_find_route(c)),
      sizeof(struct ipddp_route)))
    return -EFAULT;

Tainting marks "p" as a tainted pointer, checking warns that "p" is passed to a routine that dereferences it
3 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 foists its argument, all others must
  How to tell? Routines assigned to same function pointer

- More general: infer execution context, arg preconditions
  Interesting q: what spec properties can be inferred?
Belief analysis to find missed sources/sinks
- Detect missed sinks:
  Usual: (1) read tainted input, (2) check, (3) pass to sink
  If we see (1) & (2) but not (3) implies missed sink
  Expected
  copy_from_user(&x, arg, sz);
  if(x == MAX || x < 0)
    return -EINVAL;
  array[x] = 10;
  ...

Belief analysis to find missed sources/sinks
- Detect missed sinks:
  Usual: (1) read tainted input, (2) check, (3) pass to sink
  If we see (1) & (2) but not (3) implies missed sink
  Expected
  copy_from_user(&x, arg, sz);
  if(x == MAX || x < 0)
    return -EINVAL;
    array[x] = 10;
    ...

- Suspicious
  copy_from_user(&x, arg, sz);
  if(x == MAX || x < 0)
    return -EINVAL;
    array[x] = 10;
    ...

Belief analysis to find missed sources/sinks
- Detect missed sinks:
  Usual: (1) read tainted input, (2) check, (3) pass to sink
  If we see (1) & (2) but not (3) implies missed sink
  Expected
  copy_from_user(&x, arg, sz);
  if(x == MAX || x < 0)
    return -EINVAL;
    array[x] = 10;
    ...

- Suspicious
  copy_from_user(&x, arg, sz);
  if(x == MAX || x < 0)
    return -EINVAL;
    array[x] = 10;
    ...

- Detect missed sources of information
  Similar to pointers: if variable used to specify user addr
  implies it is untrusted. Taint it and flag.

MAY beliefs
- Separate fact from coincidence? General approach:
  Assume MAY beliefs are MUST beliefs & check them
  Count number of times belief passed check (success)
  Count number of times belief failed check (fail)
  Rank errors based on ratio of successes to failures
- How to weigh evidence?
  Treat as independent "binomial trials"
  Pr(k, n) = n! / (k! * (n-k)!)
  Expected = n*p. Stddev = sqrt(n*p*(1-p)). Typical p = 0.8
  Compute degree of skew in terms of stddev:
  Z = (observed - expected) / stddev

Statistical: Deriving deallocation routines
- Use-after free errors are horrible.
- Problem: lots of undocumented sub-system free functions
- Goal: derive behaviorally: pointer "p" not used after call
  "free(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 entry to each call site
- Emit an "error" message at every use:
  foo(p), foo(p), bar(p), bar(p)
  p = x; p = x; p = x; p = 0;
- Rank errors using z test statistic: z(checks, errors)
- E.g., foo(x), foo(x) < bar(x), bar(x) so rank bar's error first
- Results: 23 free errors, 11 false positives

Recall: deterministic free checker

```
sfree_checker { 
  state decl any_pointer v; 
  decl any_pointer x; 
  start: { kfree(v); } \rightarrow v.freed 
  v.freed: 
    { v != x } \land \{ v == x \} \rightarrow \{ /* do nothing */ \}
    \{ v \} \rightarrow \{ err("Use after free!"); \}
} 
```
A statistical free checker

```c
A free_checker local { state decl any_pointer v; decl any_fn_call; decl any_pointer x;

start: { call(v) } ==> v.freed,
    { mc_v_set_data(v, mc_identifier(call));
    v_note("checking [POV=Data]", v);
    }
    v.freed:
    { v != x } || (v == x) ==> { /* do nothing */ }
    | (v) ==> { v_err("Use after free! [FAIL=Data]", v); }
    }
```  

Ranked free errors

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kfree(0)</td>
<td>2623 checks, 60 errors, z = 48.87</td>
</tr>
<tr>
<td>Kfree_skb(0)</td>
<td>1070 checks, 13 errors, z = 31.92</td>
</tr>
<tr>
<td>dev_Kfree_skb(0)</td>
<td>109 checks, 4 errors, z = 9.67</td>
</tr>
<tr>
<td>cmd_free(1)</td>
<td>18 checks, 1 error, z = 3.77</td>
</tr>
<tr>
<td>cmd_free_skb(1)</td>
<td>15 checks, 1 error, z = 3.35</td>
</tr>
<tr>
<td>cmd_free_skb(0)</td>
<td>18 checks, 2 errors, z = 3.2</td>
</tr>
</tbody>
</table>

A bad free error

```c
A bad_free_error

```  

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()  
  Check [A] || Check [B] => error => no bar

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/cmplc.c: cm_mid_release: */
  lock_kernel();
  if (file->f_mode & PMODE_WRITE) {
    add_wait_queue(as->midi.wait, &wait);
    ...  
  if (file->f_flags & O_NONBLOCK) {
    remove_wait_queue(as->midi.wait, &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” message
  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);

- System-specific: suppress impossible paths
  // Mark paths containing non-returning function as dead.
  start: { call(arge) | == { 
    if(mc_is_name(call, "panic")
      mc_kill_path(mc_stmt);
    } // or conditionals that check "user" for "kernel"
    | (v != 0) => { 
      if(mc_name_contains(v, "kernel")
        mc_kill_true_path(mc_stmt);
      else if(mc_name_contains(v, "user")
        mc_kill_false_path(mc_stmt);
      } }

Managing false positives

- Deterministic ranking
  Short distance over long, local over global.
  Important over less important

- System-specific: suppress impossible paths
  // Mark paths containing non-returning function as dead.
  start: { call(arge) | == { 
    if(mc_is_name(call, "panic")
      mc_kill_path(mc_stmt);
    } // or conditionals that check "user" for "kernel"
    | (v != 0) => { 
      if(mc_name_contains(v, "kernel")
        mc_kill_true_path(mc_stmt);
      else if(mc_name_contains(v, "user")
        mc_kill_false_path(mc_stmt);
      } }

Z-ranking Example: rank paired locks

- Intraprocedural lock checker false positives
  Confused role of semaphores
  Apply z-ranking:
    Failure: acquisition, no release
    Success: correct release
  Related: all messages for same acquisition site

Talk Overview

- Metacompilation Overview
  - Beliefs analysis: broader checking
    - Beliefs code MUST have: Contradictions = errors
    - Beliefs code MAY have: check as MUST beliefs and rank
      errors by belief confidence
    - Key feature: find errors without knowing truth

- Next: Managing false positives
  - Some experience

Statistical ranking: z-ranking

- Which analysis decisions to trust?
  - Valid analysis decision: many successful checks, one error
  - Classic false positive: few successful checks, many errors
  - Use the z-test statistic to rank!

- How?
  - Decide what constitutes a success or failure
    - Group related failures and successes into eqv class eq[i]
    - Rank errors by z-rank of their class z(eq[i].s, eq[i].f)

- Used to rank locking errors, freed pointers, security
  errors, ...

Some cursory experiences

- Bugs are everywhere
  - Initially worried we'd resort to historical data...
  - 100 checks? You'll find bugs (if not, bug in analysis)
  - People don't fix all the bugs

- Often simple analysis works well.
  - Easy for programmer? Easy for analysis. Hard for
    analysis. Hard for person.

- Soundness not needed for good results
  - Must: Doesn't compile? Delete it.

- Finding errors often easy, saying why is hard
  - Have to track and articulate all reasons.

- More analysis a mixed blessing
  - Has to be replicated by programmer. Exhausting. We
    denote errors for each analysis step.
Two big open questions

- How to find the most important bug?
  Main metric is bug counts or type
  How to flag the 2-3 bugs that will really kill system?

- Do static tools really help?
  Bugs found
  Bugs that mattered
  The hope
  Bugs found
  Bugs that mattered
  The null hypothesis

A Possibility

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

- Deeper checking
  Bandera

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
  Using programmer beliefs to infer state of system,
  relevant rules
  Key feature: find errors without knowing truth

- Managing false positives
  System-specific techniques
  Use statistical analysis