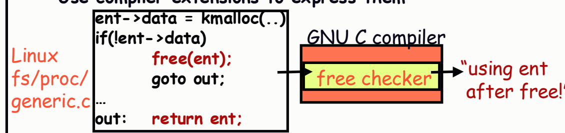
Finding bugs with system-specific static analysis

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



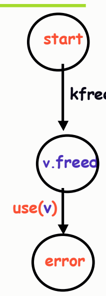
- Nice: scales, precise, statically find 1000s of errors

A bit more detail

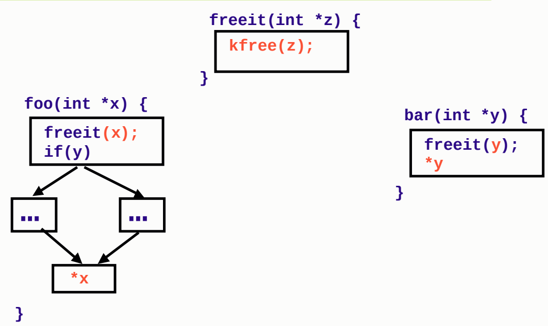
```
sm free_checker {
 state decl any_pointer v;
 decl any_pointer x;

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

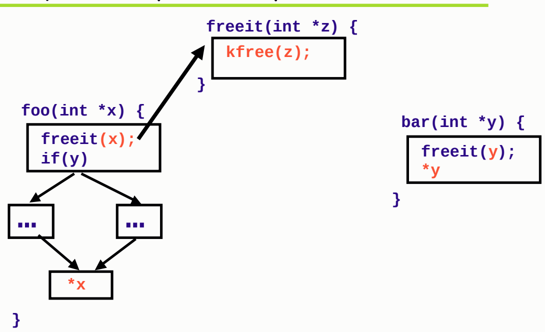
```
/* 2.4.1: fs/proc/generic.c */
ent->data = kmalloc(…)
if(!ent->data) {
     kfree(ent);
     goto out;
…
out: return ent;
```



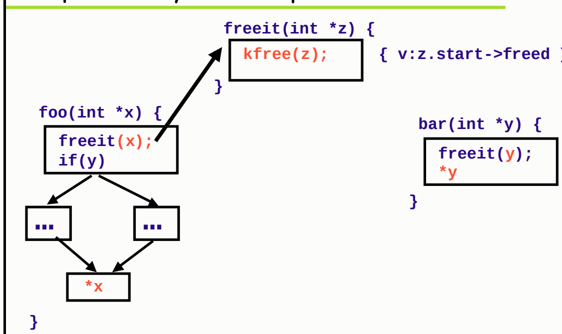
A quick analysis example



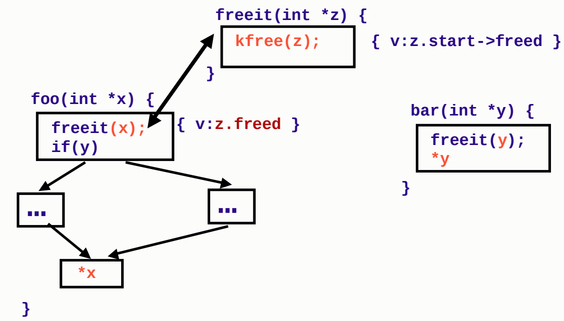
A quick analysis example



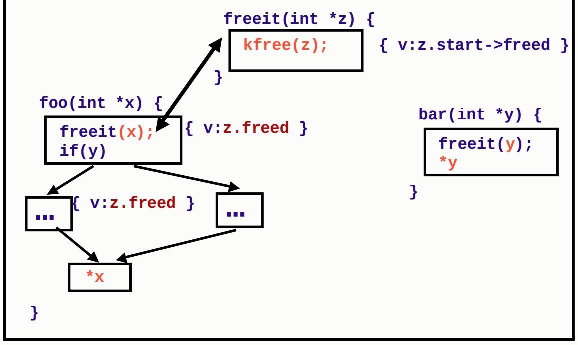
A quick analysis example



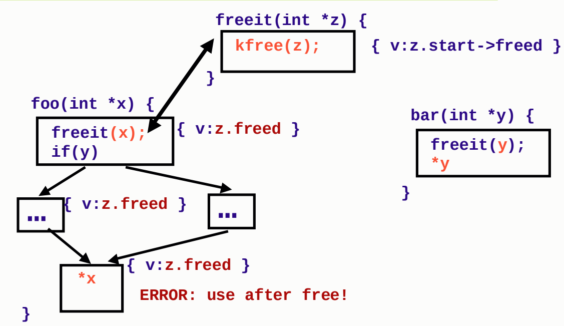
A quick analysis example



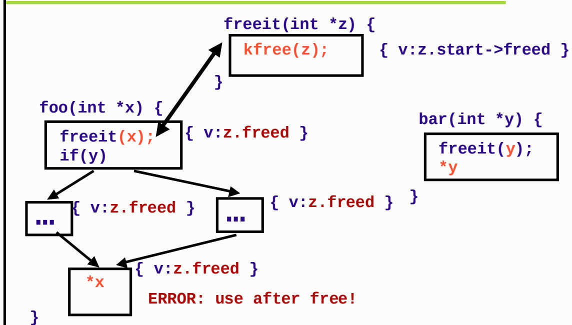
A quick analysis example



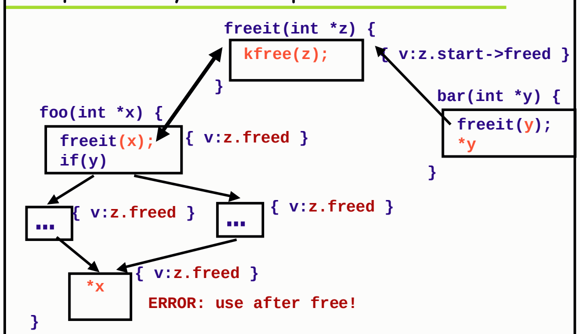
A quick analysis example



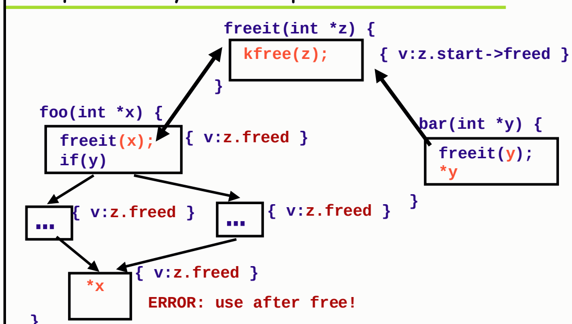
A quick analysis example



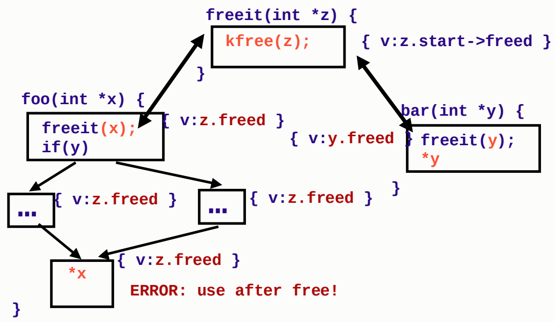
A quick analysis example



A quick analysis example



A quick analysis example

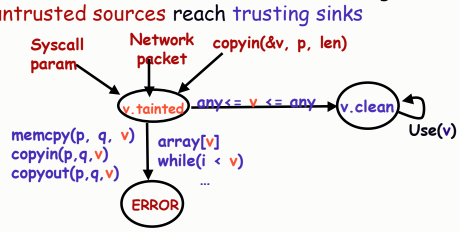


Talk Overview

- Metacompilation [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 looked at
- 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

"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.

- 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);
```

- Unexpected overflow:

```
/* 2.4.9-ac7/fs/intermezzo/psdev.c */
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);
```

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
Denaial 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

Many other checkers

- Concurrency
  - Deadlock
  - Missing unlock or enable interrupt call
  - Prototype race detection
- Memory errors
  - Null pointer bugs
  - 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 unvetted input: integers, strings, pointers
- Exploitable errors

- Statistically inferring
  - Paired 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 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.

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

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?

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 -EINVALID;
array[x] = 10;
…
```

Suspicious

```
copy_from_user(&x, arg, sz);
if(x >= MAX || x < 0)
    return -EINVALID;
… no dangerous use…
```

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 -EINVALID;
array[x] = 10;
…
array[arg] = 11;
```

Suspicious

```
copy_from_user(&x, arg, sz);
if(x >= MAX || x < 0)
    return -EINVALID;
… no dangerous use…
```

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

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 -EINVALID;
array[x] = 10;
…
array[arg] = 11;
```

Suspicious

```
copy_from_user(&x, arg, sz);
if(x >= MAX || x < 0)
    return -EINVALID;
… no dangerous use…
```

- 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 chose k) * p^k * (1-p)^(n-k)
  - Expected = n*p. Stddev= sqrt(n*p*(1-p)). Typical p = .8
  - Compute degree of skew in terms of stddevs:
    - Z = (observed – expected) / stddev
    - = (k – n*p) / sqrt(n*.8*.2)

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

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

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);
```

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();
```

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;
```

Talk Overview

- Metacompilation Overview
- Belief 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

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(args) } ==> {
        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);
    }
```

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, …

Z-ranking Example: rank paired locks

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

```
contrived(lock_t l) {
    spin_lock(l);
    if(!(p = malloc(…))
        return -ENOMEM;
    spin_unlock(l);
```

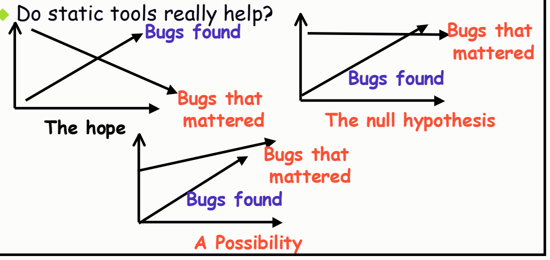
Z	S:F	Bugs:FP	Cum Z	Cum Rand
4.9	5:1	1:0	1:0	0:1
4.3	4:1	2:1	3:1	1:3
2.7	2:1	7:5	10:6	2:14
2.1	2:2	2:0	12:6	2:16
1.5	1:1	3:15	15:21	5:31
…				
-.4	0:1	0:93	18:118	12:124

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
  - Most extreme: 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 demote 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?



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