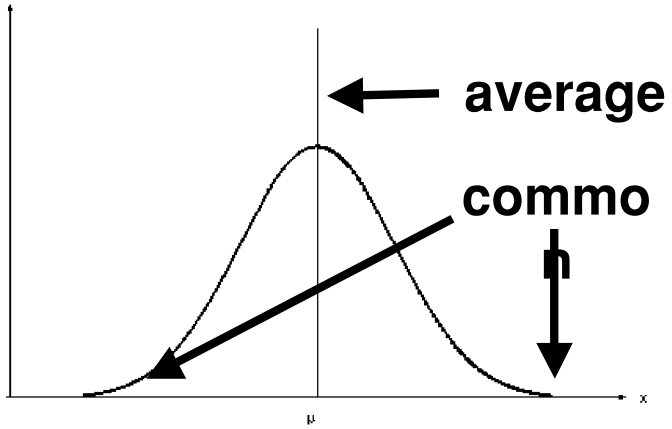


A couple billion lines of code later: static checking in the real world

Andy Chou, Ben Chelf, Seth Hallem
Scott McPeak, Bryan Fulton, Charles Henri-Gros,
Ken Block, Anuj Goyal, Al Bessey
Chris Zak
& many others
Coverity

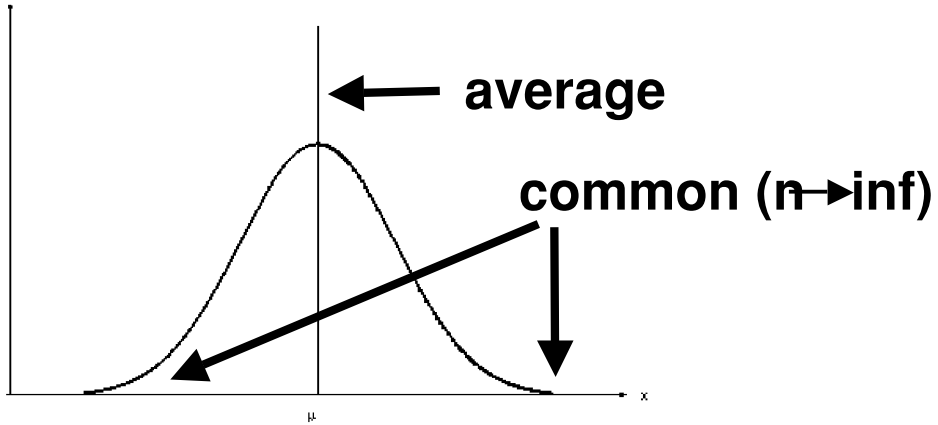
Dawson Engler
Associate Professor
Stanford

One-slide of background.

- Academic Lineage:
  - **MIT: PhD thesis = new operating system ("exokernel")**
  - **Stanford ('99--): techniques that find as many serious bugs as possible in large, real systems.**
- Main religion = results.
  - **System-specific static bug finding [OSDI'00, SOSP'01…]**
  - **Implementation-level model checking [OSDI'02, '04, '06].**
  - **Automatic, deep test generation [Spin'05,Sec'06,CCS'06, ISSTA'07]**
- Talk:
  - **Experiences commercializing our static checking work**
  - **Coverity: 400+ customers, 100+ employees.**
  - **Caveat: my former students run company; I am a voyeur.**

Many stories, two basic plots.

- Fun with normal distributions



- Social vs Technical: “What part of NO! do you not understand?”

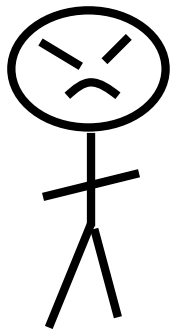
No: you cannot touch the build.

No: we will not change the source.

No: this code is not illegal C.

No: we will not understand your tool.

No: we do not understand static analysis.

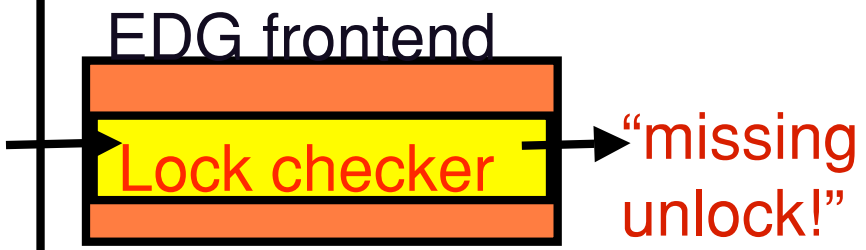


Context: system-specific static analysis

- 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

The high bit: works well.

- Lots of papers.
  - **System specific static checking [OSDI'00] (Best paper),**
  - **Security checkers [Oakland'02,CCS'03, FSE'03]**
  - **Race condition and deadlock [SOSP '03],**
  - **Others checkers: [ASPLOS'00,PLDI'02,FSE'02 (award)]**
  - **Infer correctness rules [SOSP'01, OSDI'06]**
  - **Statistical ranking of analysis decisions [SAS'03,FSE'03]**
- PhDs, tenure, award stuff.
- Commercialization: Coverity.
  - **Successful enough to have a marketing dept.**
  - **Proof: next few slides.**
  - **Useful for where data came from & to see story settled on after N iterations.**

Our Mission

coverity

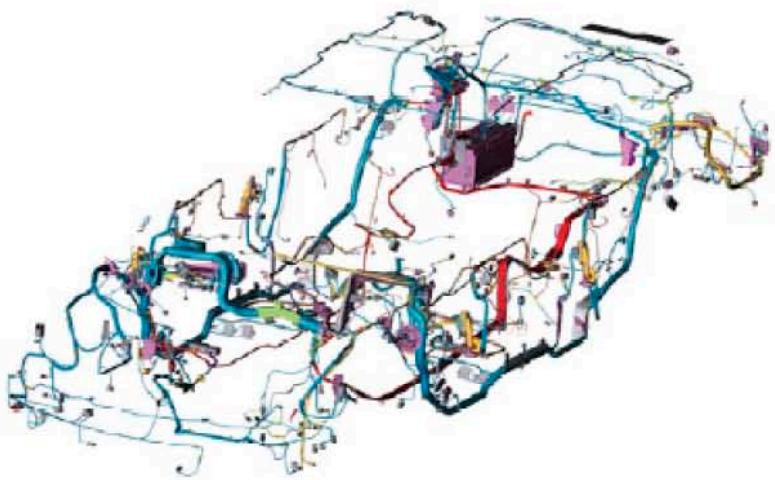
To improve software quality by automatically identifying and resolving critical defects and security vulnerabilities in your source code

1. Exploding complexity

coverity

TODAY'S PREMIUM AUTOMOBILE
10 MILLION LINES OF CODE

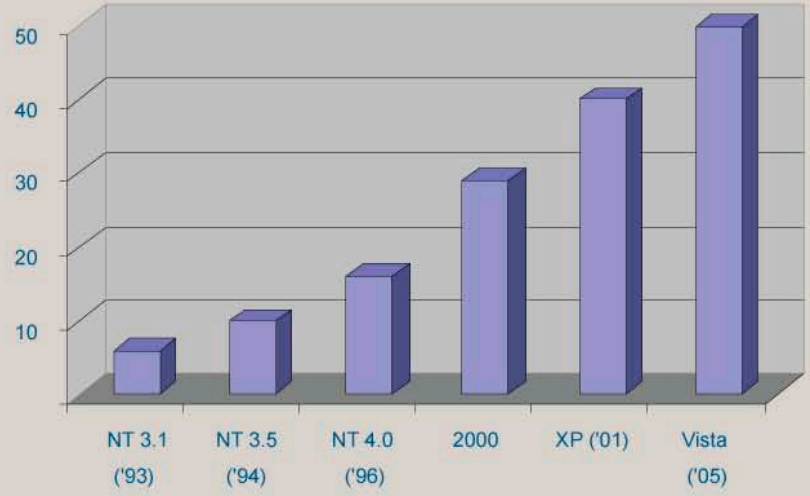
Diagram of Onboard Network of a Premium Car
(Source: Manfred Boy, ICSE'06)



Windows Vista

WINDOWS VISTA
50 MILLION LINES OF CODE

Windows OS Code Growth - MLOC
(Source: Wikipedia)



2. Multiple Origins of Source Code

coverity

Outsourced Code

- Offshore
- Onshore

3rd Party Code

- Components and Objects
- Libraries

Infrastructure Frameworks

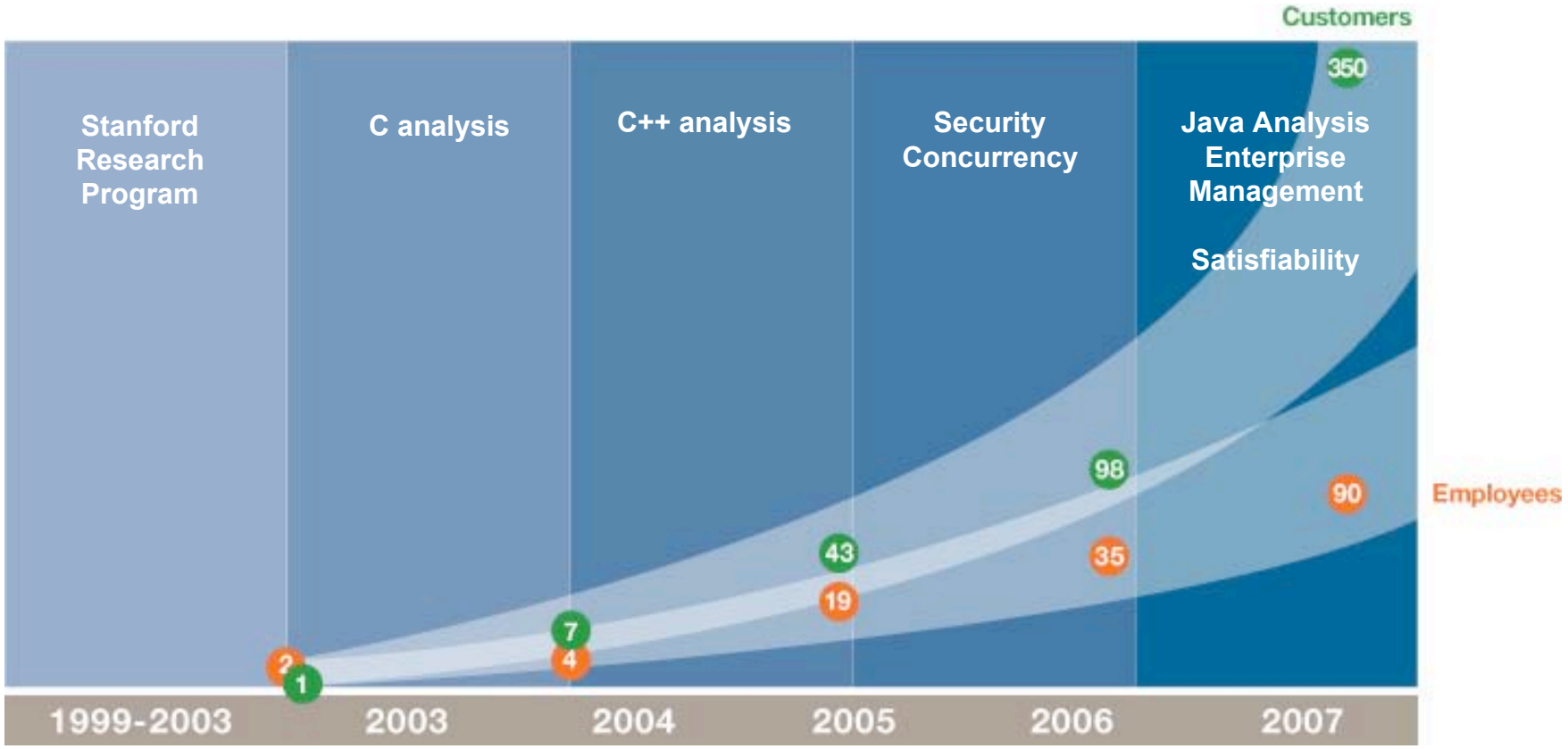
- Java EE
- Service Oriented Architecture (SOA)

Legacy Code Bases

- Code through Acquisitions
- Code Created by Past Employees

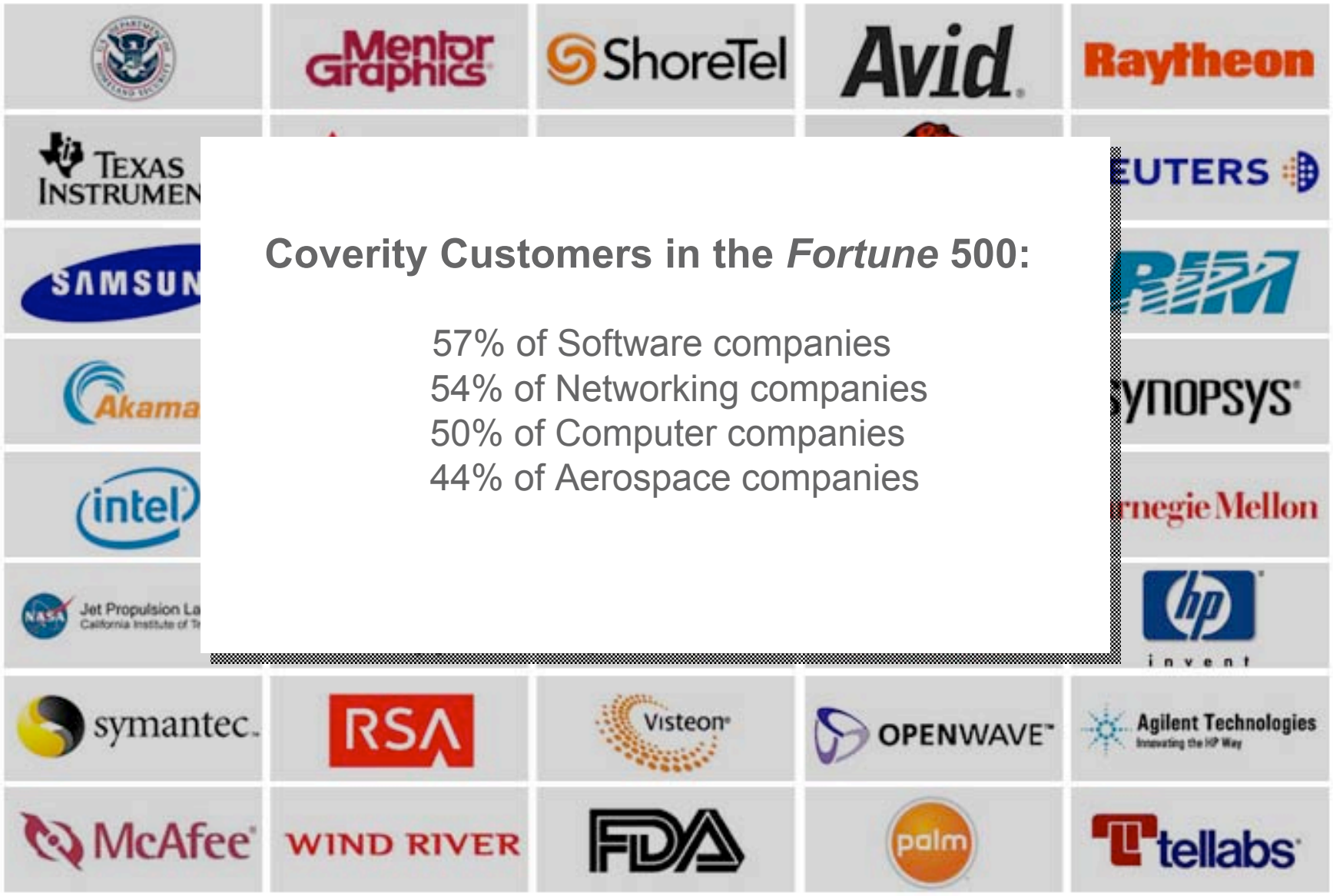
Technological Leadership [2007:dated]

coverity



Over 1 Billion Lines of Code

coverity



Coverity Customers in the *Fortune* 500:

57% of Software companies
54% of Networking companies
50% of Computer companies
44% of Aerospace companies

Over 1 Billion Lines of Code

coverity

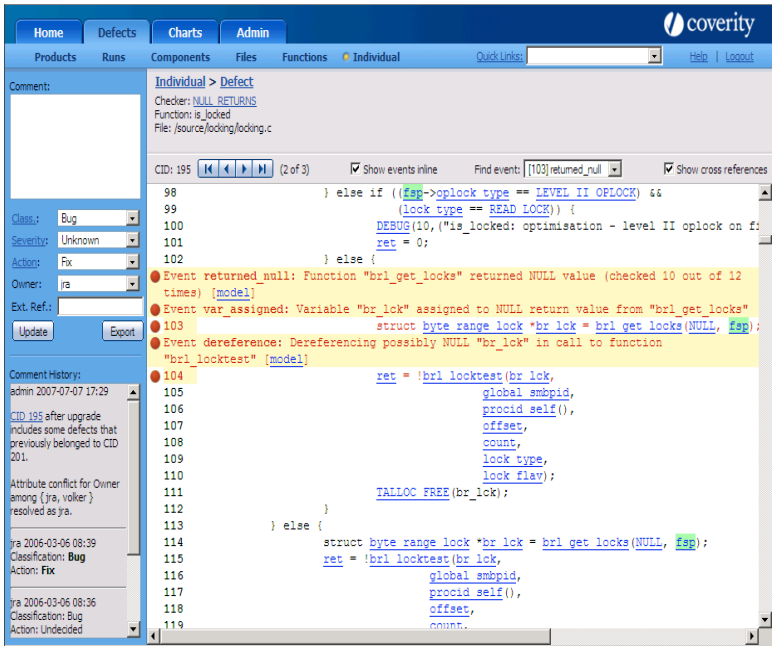
Coverity Trial Process

Test your code quality

- Analyze your largest code base
- One day set up, two hours for results presentation
- Test drive the product at your facility

Benefit to your team

- Post trial report describing summary of findings
- Sample defects from your code base
- Fully functional defect resolution dashboard



Trial = main verb of company.

- Can't do trial right, won't have anything.
- First order dynamics:
  - **Setup, run, present "live" w/ little time.**
  - **Error reports must be good, setup easy since won't understand code**
  - **Can't have many false positives since can't prune**
  - **Must have good bugs since can't cherry pick**
- Some features:
  - **$0. means anyone can get you in. Cuts days of negotiation. Sales guy goes farther.**
  - **Straight-technology sale. Often buyer=user.**
  - **Filters high support costs: if customer buys, had a setup where we could configure and get good errors.**
  - **Con: trial requires shipping SE + sales guy.**

Overview

- Context
- Now:
  - **A crucial myth.**
  - **Some laws of static analysis**
  - **And how much both matter.**
- Then: The rest of the talk.

A naïve view

- Initial market analysis:
  - **“We handle Linux, BSD, we just need a pretty box!”**
  - **Obviously naïve.**
  - **But not for the obvious reasons.**

- Academia vs reality difference #1:
  - **In lab: check one or two things. Even if big = monoculturish.**
  - **In reality: you check many things. Independently built.**
  - **Many independent things = normal distribution.**
  - **Normal dists have points several std dev from mean (“weird”)**
  - **Weird is not good.**

- First law of checking: no check = no bug.
  - **Two even more basic laws we’d never have guessed mattered.**

Law of static analysis: cannot check code you do not see.

How to find all code?

- `find . –name “*.c” ’ ?

  Lots of random things. Don’t know command line or includes.

- Replace compiler?

  “No.”

- Better: intercept and rewrite build commands.

  make –w > & out
  replay.pl out # replace ‘gcc’ w/ ‘prevent’

  In theory: see all compilation calls and all options etc.

- Worked fine for a few customers.

  Then: “make?”

  Then: “Why do you only check 10K lines of our 3MLOC system?”

  Kept plowing ahead.

  “Why do I have to re-install my OS from CD after I run your tool?”

  Good question…

The right solution

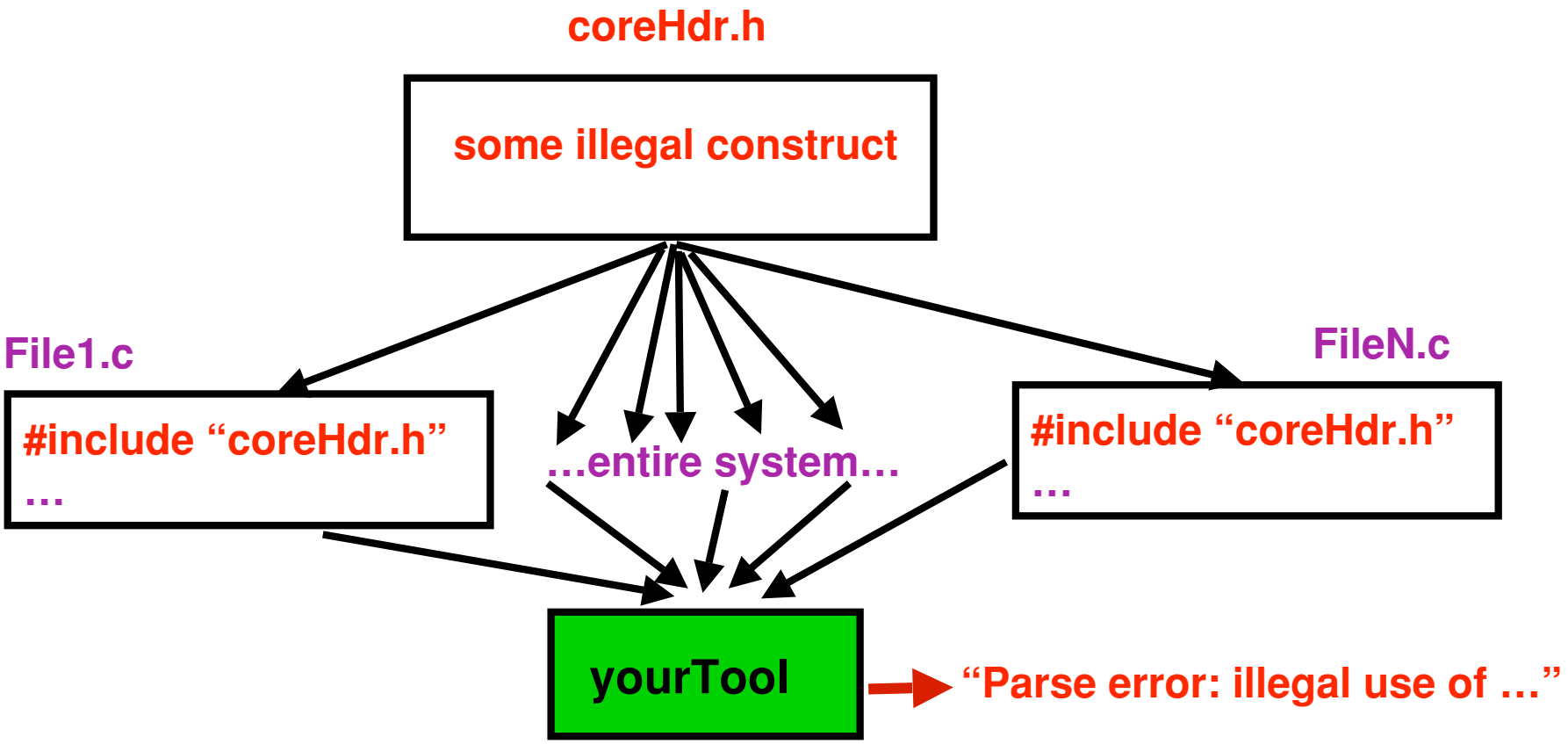
- Kick off build and intercept all system calls
  - **"cov_build <your build command>" → grab chdir, execve, …**
  - **Know exact location of compile, version, options, environ.**
- Probably *the* crucial technical feature for initial success.
  - **Go into company cold, touch nothing, kick off, see all code.**
  - **In early 2000s more important than quality of analysis?**
  - **Not bulletproof. Little known law: "Can't find code if you can't get command prompt"**
- A only-in-company-land sad story:
  - **On windows: intercept means we must run compiler in debugger.**
  - **Widely used version of msoft compiler has a use-after-free bug hit if source contains "#using"**
  - **Works fine normally. Until run w/ debugger!**
  - **Solution?**
- Lesson learned?
  - **Well, no: Java.**

(Another) Law of static analysis: cannot check code you cannot parse

Myth: the C language exists.

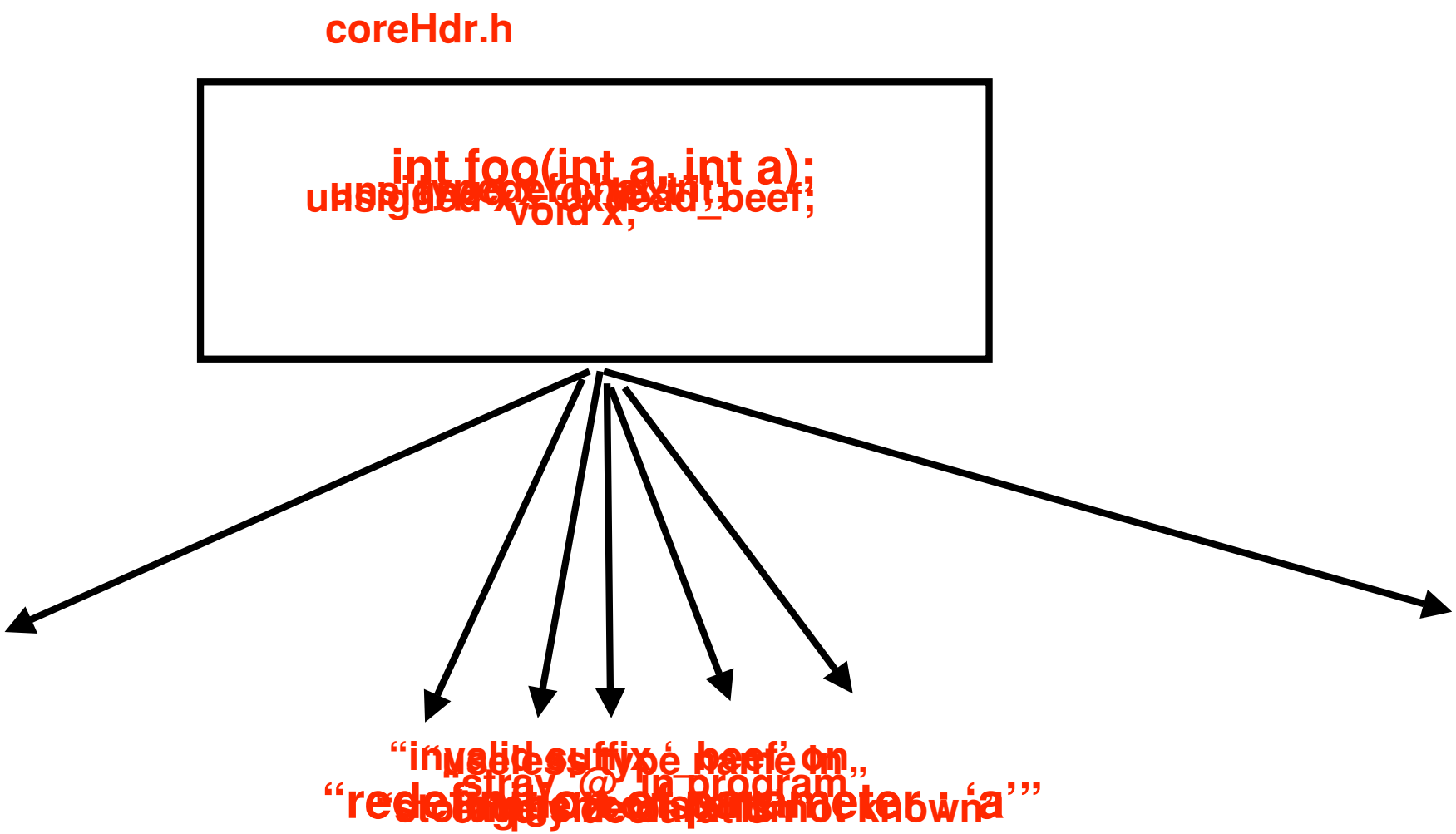
- Well, not really. The standard is not a compiler.
  - **The language people code in?**
  - **Whatever their compiler eats.**
  - **Fed illegal code, your frontend will reject it.**
  - **It's *your* problem. Their compiler "certified" it.**
- Amplifiers:
  - **Embedded = weird.**
  - **Msoft: standard conformance = competitive disadvantage.**
  - **C++ = language standard measured in kilos.**
- Basic LALR law:
  - **What can be parsed will be written. Promptly.**
  - **The inverse of "the strong Whorfian hypothesis" is a empirical fact, given enough monkeys..**

A sad storyline that will gross exactly $0.

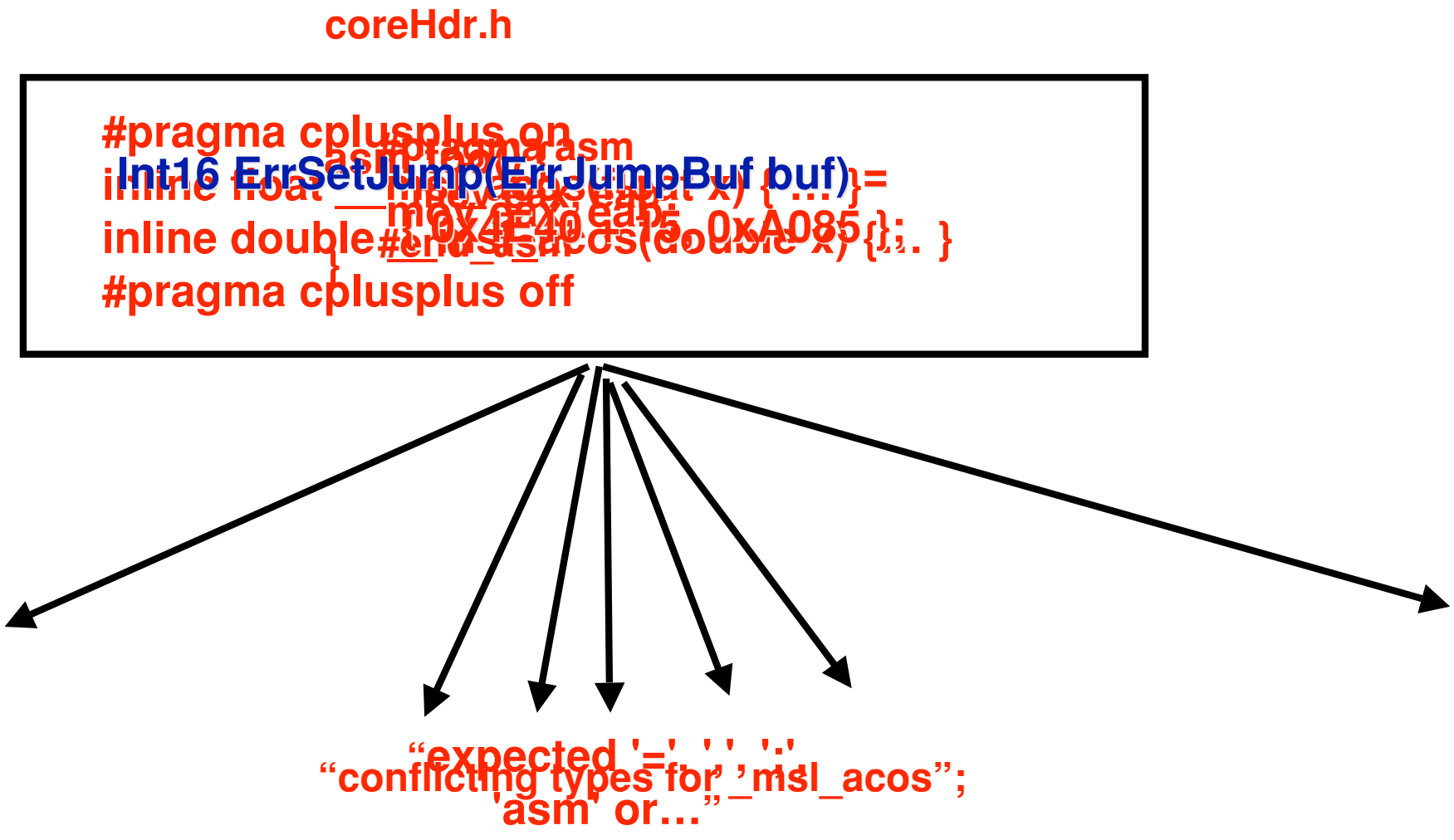


- “Deep analysis?! Your tool is so weak it can’t even parse C!”

Some specific example stories.



Some specific example stories.



Great moments in unsound hacks

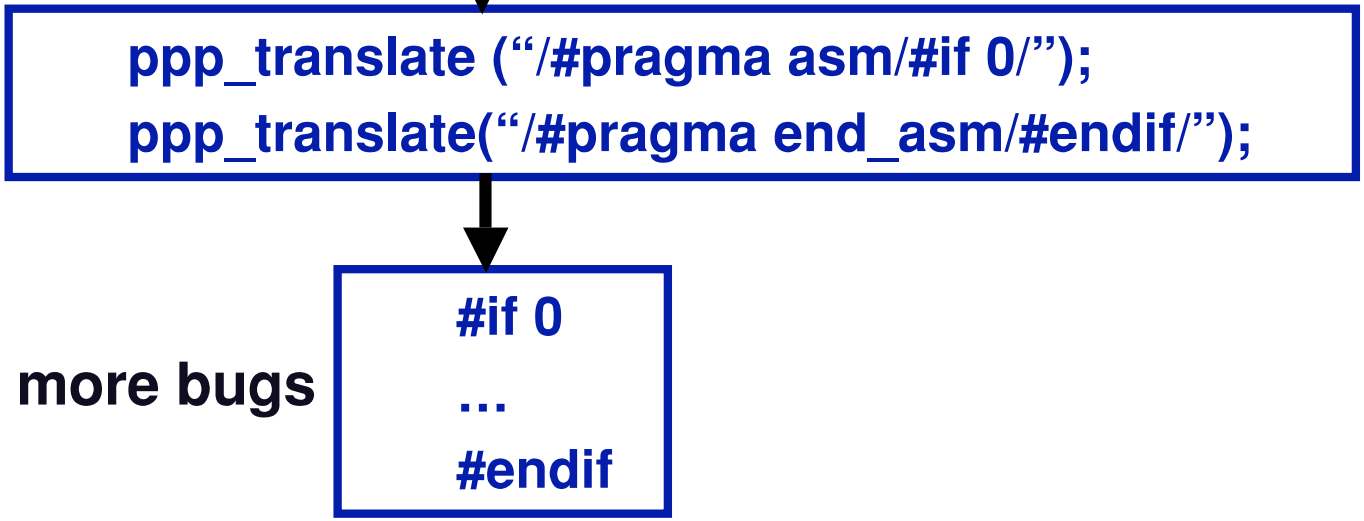
- Tool doesn't handle (illegal) construct?

Have reg-ex that runs before preprocessor to rip it out.

Amazingly gross.

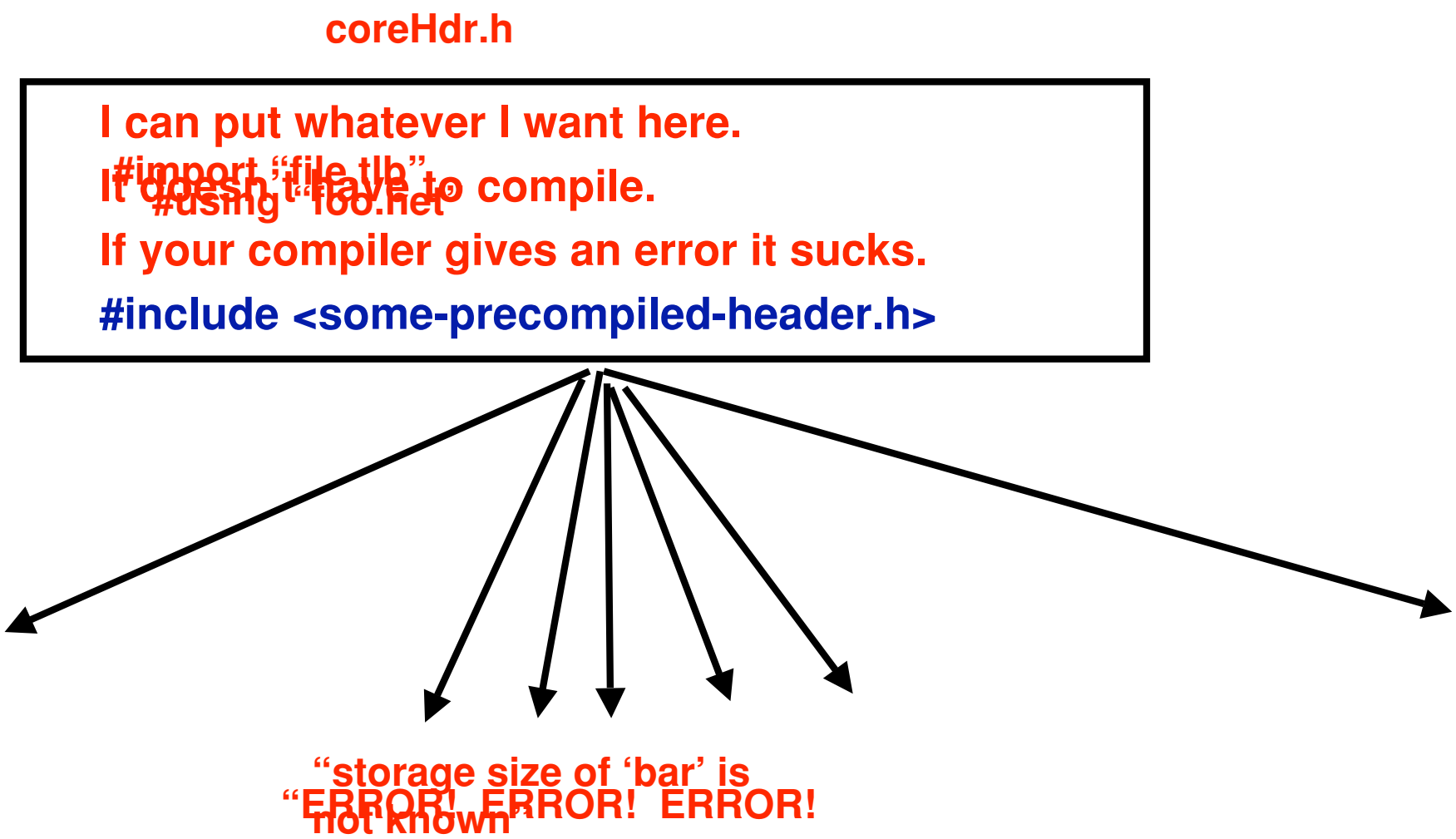
Actually works.

```
#pragma asm
…
#pragma end_asm
```



Unsound = more bugs

Msoft story: ubiquitous, gross.



Making the depressing more precise

- Goal: you must approximate:

  Lang(Compiler) = { f | ∃ o,e,c Accepts(Compiler,f,o,e,c) }

- Where:

  Compiler = a specific version of a "native" compiler

  f = file.c, plus headers

  o = ordered list of command line options and response files

  e = environment

  c = compiler-specific configuration files

- To work well:

  Lang(Tool) ~ Lang(Compiler$_1$) ∪ Lang(Compiler$_2$) ∪ …

  Where | Lang(Compiler$_1$) ∪ Lang(Compiler$_2$) ∪ … | = large.

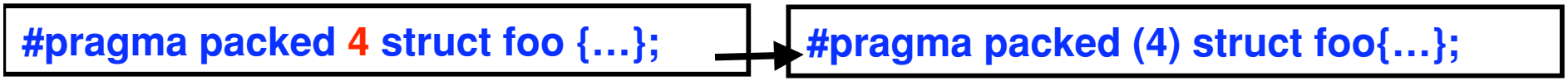
OK: so just how much does C not exist?

- We use Edison Design Group (EDG) frontend
  - **Pretty much everyone uses. Been around since 1989.**
  - **Aggressive support for gcc, microsoft, etc. (bug compat!)**
- Still: coverity by far the largest source of EDG bugs:
  - **406 places where frontend hacked ("#ifdef COVERITY")**
  - **266 "add compiler flag" calls**
  - **still need custom rewriter for many supported compilers:**

912 arm.cpp
629 bcc.cpp
334 cosmic.cpp
1848 cw.cpp
673 diab.cpp
914 gnu.cpp
1656 metrowerks.cpp
1294 microsoft.cpp
285 picc.cpp
160 qnx.cpp
1861 renesa.cpp
384 st.cpp
457 sun.cpp
294 sun_java_.cpp
756 xlc.cpp
280 hpux.cpp
603 iccmsa.cpp
421 intel.cpp
1425 keil.cpp

Completely unbelievable!

- Incredibly banal. But if not done, can't play.
  - **Takes more effort than can imagine.**
  - **Full time team.**
  - **This is their mission. Never finished.**
  - **Certainly not in only 5 years.**
- Two examples from trial reports from within *72 hours* of making this slide:



- *Never* would have guessed this is the first-order bound on how much bugs you find.

Annoying amplifier: Can we get source?

- *NO*!
  - **Despite NDAs**
  - **Even for parse errors**
  - **Even for preprocessed**
  - **Might just be because coverity too small to sue?**
- Sales engineer has to type in from memory.
  - **And this works as well as you'd expect.**
  - **Even worse for performance problems.**
  - **Oh, and you get about 3 tries to fix a problem.**
- Bonus: add a TLA and things get worse.
  - **NSA = Can we see source? NO!**
  - **FDA, FAA = frozen toolchain. Theirs. Yours. Banal, crucial: Where to get license for a 20+yr/old compiler?**

The end result

- Heuristic: If you've heard of it, will wind up supporting it:

- Forced support for many things haven't heard of (or read obituary for).

Tasking, Microtec, Metaware, Microchip C-18, Code Vision,

A compiler development company / Photography company: "Specializing in Anime and SF/Fantasy Convention photography and other costuming photography. We can also do on-location photoshoots."

Overview

- Context

- The banal hand of reality:
  - **Law: Cannot check code you can't find**
  - **Law: Cannot check code you can't parse**
  - **Myth: C exists**
- Next:
  - **Do bugs matter?**
  - **Do false positives matter?**
  - **Do false negatives matter?**
- The best bugs
- Academics meet reality. Reality wins.
  - **You fix all bugs, right?**
  - **The evils of non-determinism**

Do bugs matter? ("Huh?")

- Shockingly common: clear, ugly crash error.
  - **"So?"**
  - **"Isn't that bad? What happens?"**
  - **"Oh, will crash. We will get a call." Shrug.**
- If developers don't feel pain, they often don't care.
  - **Technical: clustered applications that reboot quick.**
  - **Non-technical: if QA cannot reproduce, then no blame.**
- But bugs matter right?
  - **Not if: Too many. Too hard. [More later]**
- The next step down: "That's not a bug"
  - **Recognition requires understanding.**
  - **Cubicles are plentiful. Understanding, not so much.**

“No, your tool is broken: that’s not a bug”

- “No, it’s *loop*.”

```
for(i=1; i < 0; i++)
    …deadcode…
```

- “No, I meant to do that: they are next to each other”

```
int a[2], b;
memset(a, 0, 12);
```

- “No, that’s ok: there is no malloc() between”

```
free(foo);
foo->bar = …;
```

- “No, ANSI lets you write 1 past end of the array!”

(“We’ll have to agree to disagree.” !!!!)

```
unsigned p[4];   p[4] = 1;
```

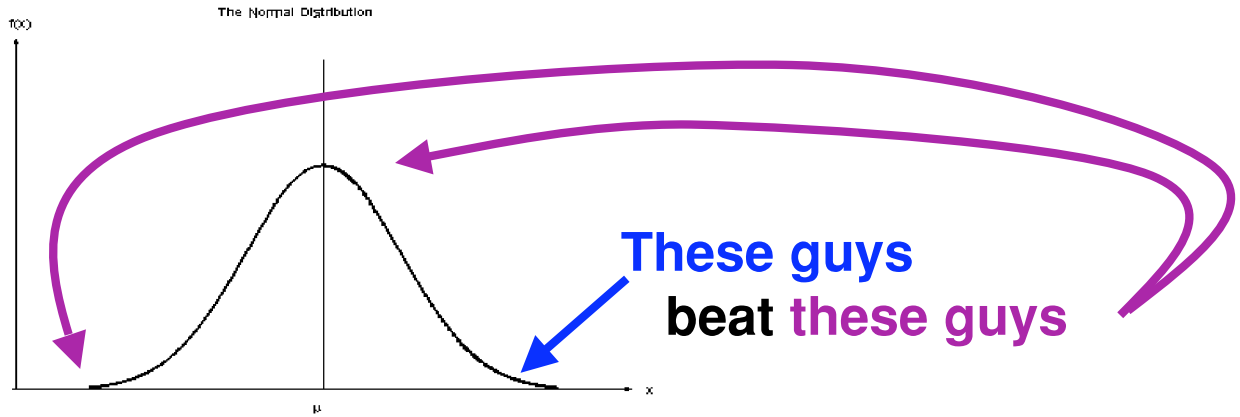
(Often) People don't understand much.

- Our initial naïve expectation: People who write code for money understand it. Instead:
  - **"To build, I just press this button…"**
  - **"I'm just the security guy"**
  - **"That bug is in 3rd party code"**
  - **"Is it a leak? Author left years ago…"**
- People don't understand compilers.
  - **"Static" analysis? What is the performance overhead?**
  - **Business card at customer site: "Static analyzer" (?!)**
  - **"We use purify, why do we need your tool?"**

  - **Anything that finds bugs = testing.**
  - **"Think of it as super compiler warnings"**

How to handle cluelessness?

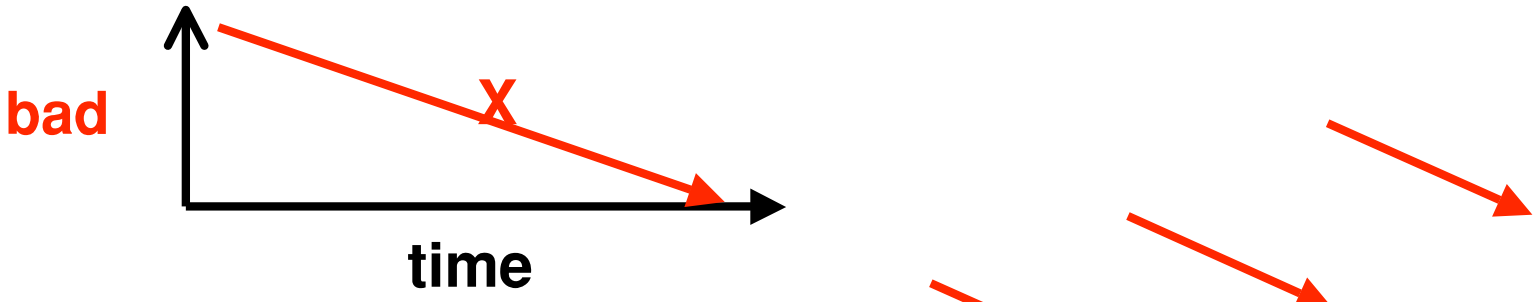
- Can't argue
  Stupidity works with modular & emotional arithmetic.
- Instead: use normal distributions.
  Try to get a large meeting. (Schedule before lunch?)



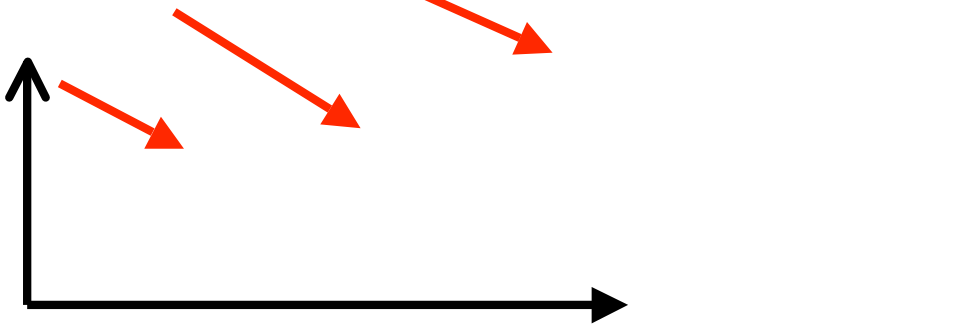
- More people in room = more likely someone in room that:
  Cares; is very smart; can diagnose error; has been burned by similar error; loses bonus for errors; …
  Is in another group!
  If layoffs happen: will be fired(!)

What happens when can't fix all the bugs?

- Rough heuristic:
  - **< 1000 bugs? Fix them all.**
  - **>= 1000? "Baseline"**
- Tool improvement viewed as "bad"
  - **You are manager. Forall metrics X of badness you want:**



No manager gets a bonus for:



How to upgrade when more bugs != good?

- Upgrade cycles:
  - **Never. Guaranteed "improvement."**
  - **Never before release (when could be most crucial)**
  - **Never before a meeting (at least is rational).**
  - **Upgrade. Roll back. (~ once per company.)**
  - **Renew, but don't upgrade. (Not cheap.)**
  - **Once a year (most large customers). "Rebaseline"**
  - **Upgrade only checkers where you fix all/most errors.**

- People really will complain when your tool gets better.
  - **V2.4: 2,400 initial errors. Fixed to get to 1,200**
  - **Upgrade to V3.5 = 2,600 errors.**
  - ***MAD* For both reasons.**

Do false positives matter?

- > 30% false rate = big problem.
  - **Ignore tool. Miss true errors amidst false.**
  - **Low trust = Complex bugs called false positives. Vicious cycle.**
  - **Caveat: unless you wrap person around checker?**
  - **Caveat: some users accept 70% (or more: security guys).**
  - **Current deployment threshold = ~20%.**
  - **Unfortunately: many cases "high FP" rate not analysis problem**
- Not all false positives equal:
  - **Initial N reports false? "Tool sucks" (N ~ 3)**

  - ***Crucial*: no embarrassing FPs.**
  - **Stupid FP? Implies tool stupid. Not good for credibility.**
  - **Social: don't want to embarrass tool champions internally**

  - **Important: no failed merges.**
  - **Mark FP once? Fine. Reappears & mark again? email support.**

A false positive pop quiz

- Remove false positives: good or bad?
  - **Initial trial: 700 reports**
  - **Fixed some problems. Remove 300 false positives. Yea!**
  - **What's the problem if they want to rerun before buy?**

- Tool X flags more errors than your tool.
  - **However: Tool X sucks and these are almost all FPs!**
  - **Do you get sale or not?**
  - **What's a bad evaluation method for your company?**
- Your checker X does tricky thing
  - **It finds *many* *many* good bugs.**
  - **Developer X does not understand your checker**
  - **What happens?**

Do false negatives matter?

- Of course not! Invisible! Oops:

  Trial: intentionally put in bugs. "Why didn't you find it?"

  Easiest sale: horribly burned by specific bug last week. You find it. If you don't?

  Upgrade checker: set of defects shifts slightly = "Dude, where is my bug?" (Goal: 5% "jitter")

  Run A and B. Even if A >> B, often A's bugs not a superset

- A very nasty dynamic (static, testing, formal)

  Tool has bugs. Some lead to FPs some to FNs.

  FPs visible = fixable. But each fix has chance of adding FN.

  FNs invisible.

- Currently: favor analysis hacks to remove FPs at cost of FNs

Overview

- Context
- The banal, vicious laws of reality and its cruel myths.

- Practical questions:
  - **Do bugs matter?**
  - **Do false positives matter?**
  - **Do false negatives matter?**

- Academics meet reality. Reality wins.
  - **The evils of non-determinism**

- The best bugs
- Commerce factoids

Non-determinism = very bad.

- Major difference from academia
  - **People really want the same result from run to run.**
  - **Even if they changed code base**
  - **Even if they upgraded tool.**
  - **Their model = compiler warnings.**

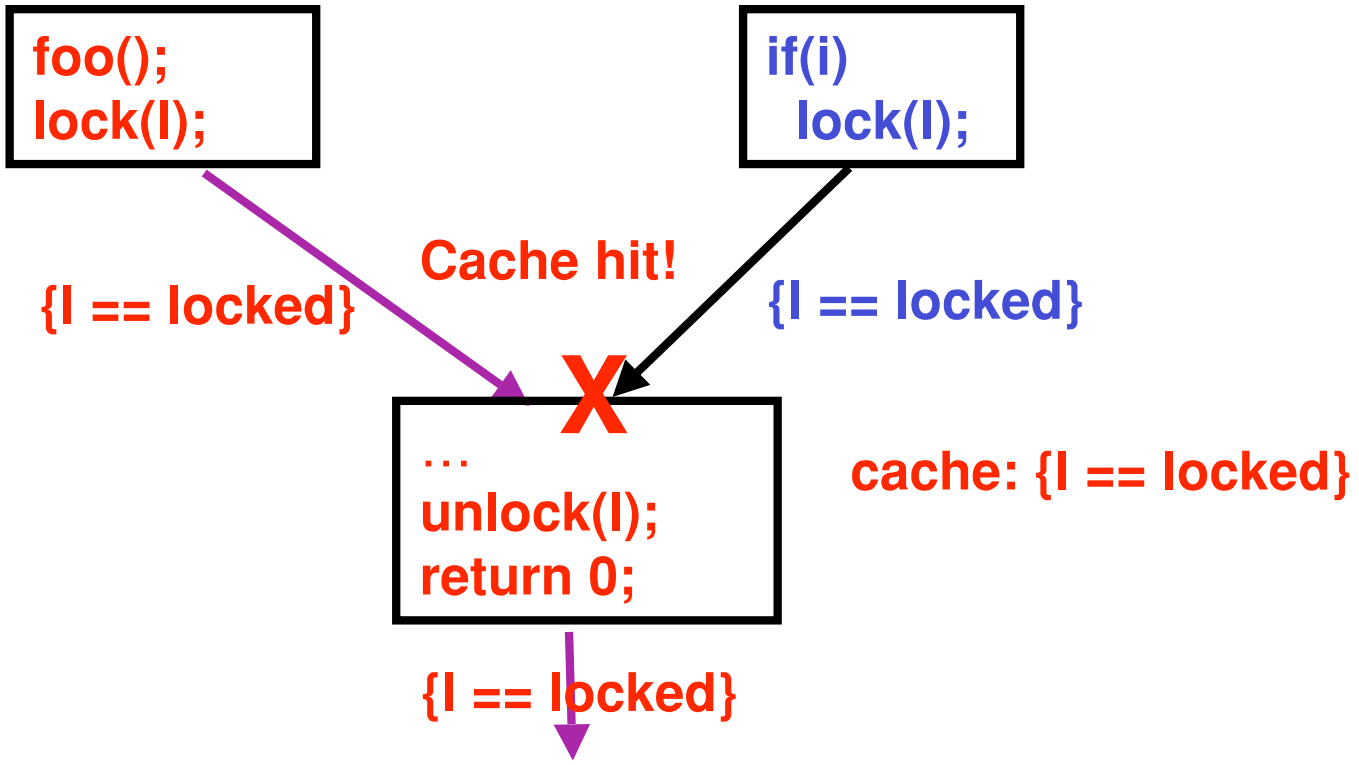
  - **Classic determism: same input + same function = same result.**
  - **Customer determinism: different input (modified code base) + different function (tool version) = same result.**

  - **They know in theory "not a verifier". Different when they actually see you lose known errors. Rule: 5% jitter.**
- Determinism requirement really sucks.
  - **Often tool changes have very unclear implications. [Next.]**
  - **Often randomization = elegant solution to scalability. Can't do.**

An explosion of non-determ unfun: caching.

- Code has exponential paths.

  At join points and in same state, prune.

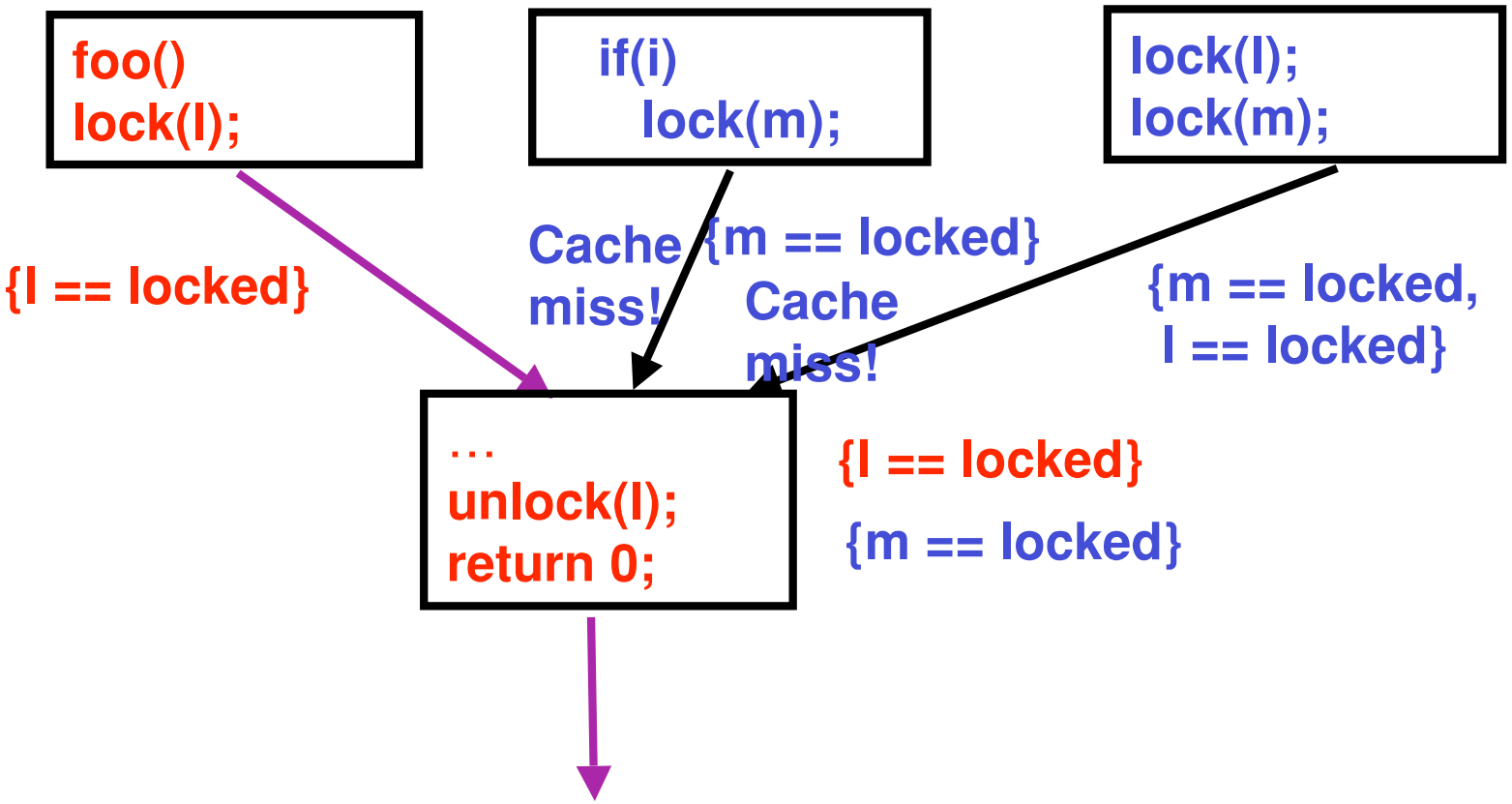


So far so good. What about multiple pieces of state?

Problem: more code = less cache hits

- Analyze more code?

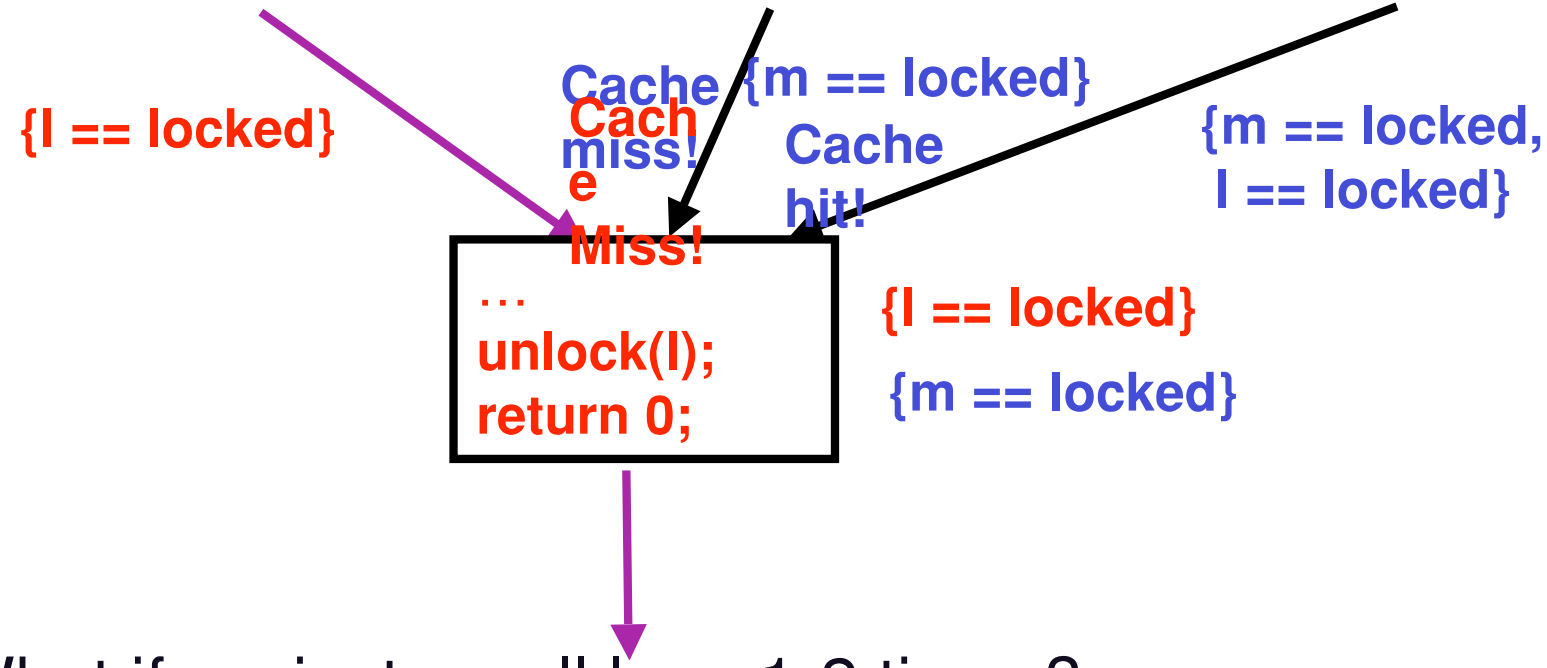
  Often don't get cache hits b/c independent state



Subset caching

- Hack:

  Cache = union of prior states. Hit = subset of that



- What if we just unroll loop 1-2 times?

  Not enough: 1MLOC + if-statements = not terminate.

  Misses bugs: want fixed point based on checker value

So?

- Basically a deal with the devil [that we would do all over again]
  - **Works well for finding many bugs on large code bases.**
  - **Not so well at finding the *same* bugs**
- Bad: Minor code change = different cache hit or miss.
  - **Effect is enormous.**
- True story:
  - **Version2.0: follows true path, then false.**
  - **Version3.0: follows false path, then true.**
  - **People went *insane*. 20% fluctuation in errors. Soln?**
- Bad: don't analyze an interesting path b/c cache hit.
  - **The occasional *very* stupid false positive or negative**
  - **Hurts trust in tool.**
  - **Lost *huge* sale:**
  - **found lots of bugs just not this one:**

```
x = 0;
for(…)
  switch(…)
    …
w = y / x;
```

Just how bad is non-determism?

- Users pick determinism over bugs, over manual labor
  - **CPrevent builds model of each analyzed function.**
  - **1st scan: missing models for functions hasn't analyzed yet**
  - **2nd scan: has these models. If use them = less FPs + more bugs.**
  - **Common: people turn off so *discards* the prior results!**
- Thwarts natural solutions to large code problems
  - **10+MLOC can be more than 24hrs. Lose sales.**
  - **Natural sol'n for exponential paths: Random search, timeout.**
  - **Both are complete non-starters.**
- No inference, no ranking.
  - **I think this is *literally* 10x dropped right on the floor.**
- Even worse in java:
  - **Represent function models (summaries) as bytecode**
  - **Elegant! Clean! Yea!.**
  - **Uh oh: must be < 32K. Larger? Discard.**
  - **Small change = different discards. Ugh…**

Overview

- Context
- The banal, vicious laws of reality and its cruel myths
- What actually matters?

- Academics meet reality, good and hard.

  The evils of non-determinism

- Bugs: often best come from analyzing programmer beliefs.

- Business factoids an academic finds amusing.

Myth: more analysis is always better

- Does not always improve results, and can make worse
- The best error:
  - **Easy to diagnose**
  - **True error**
- More analysis used, the worse it is for both
  - **More analysis = the harder error is to reason about, since user has to manually emulate each analysis step.**
  - **Number of steps increase, so does the chance that one went wrong. No analysis = no mistake.**
- In practice:
  - **Demote errors based on how much analysis required**
  - **Revert to weaker analysis to cherry pick easy bugs**
  - **Give up on error classes that are too hard to diagnose.**

More general: A too-hard bug didn't happen.

- In fact, can be worse.
  - **People don't want to look stupid.**
  - **If they don't understand error, what will they do?**
- Social has *major* big impact on technical.
  - **User not same as tool builder.**
  - **Uninformed. Inattentive. Cruel.**
  - **HUGE problem. Prevents getting many things out in world.**
- Give up on error classes that need too much sophistication.
  - **statistical inference,**
  - **race conditions,**
  - **heap tracking**
  - **globals.**
  - **In some ways, checkers lag much behind our research ones.**

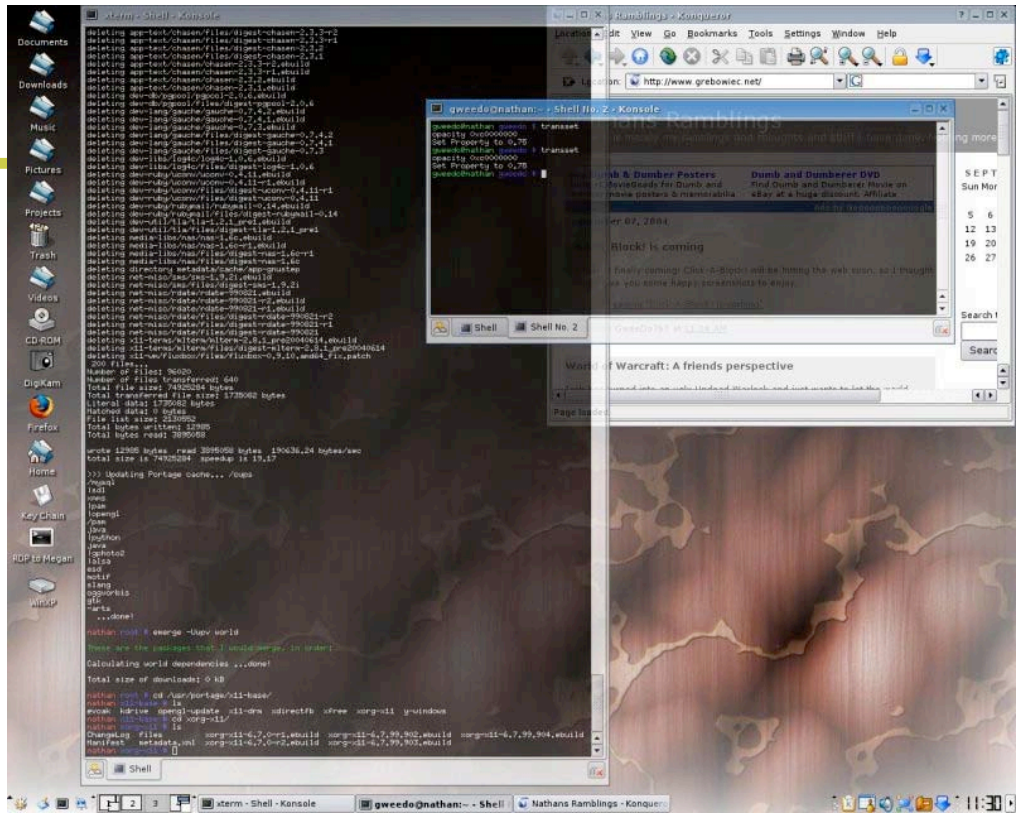
No bug is too stupid to check for.

- Someone, somewhere will do anything you can think of.
- Best recent example:

  From security patch for bug found by Coverity in X windows that lets almost any local user get root.

  Got on foxnews (website, not O'Riley)

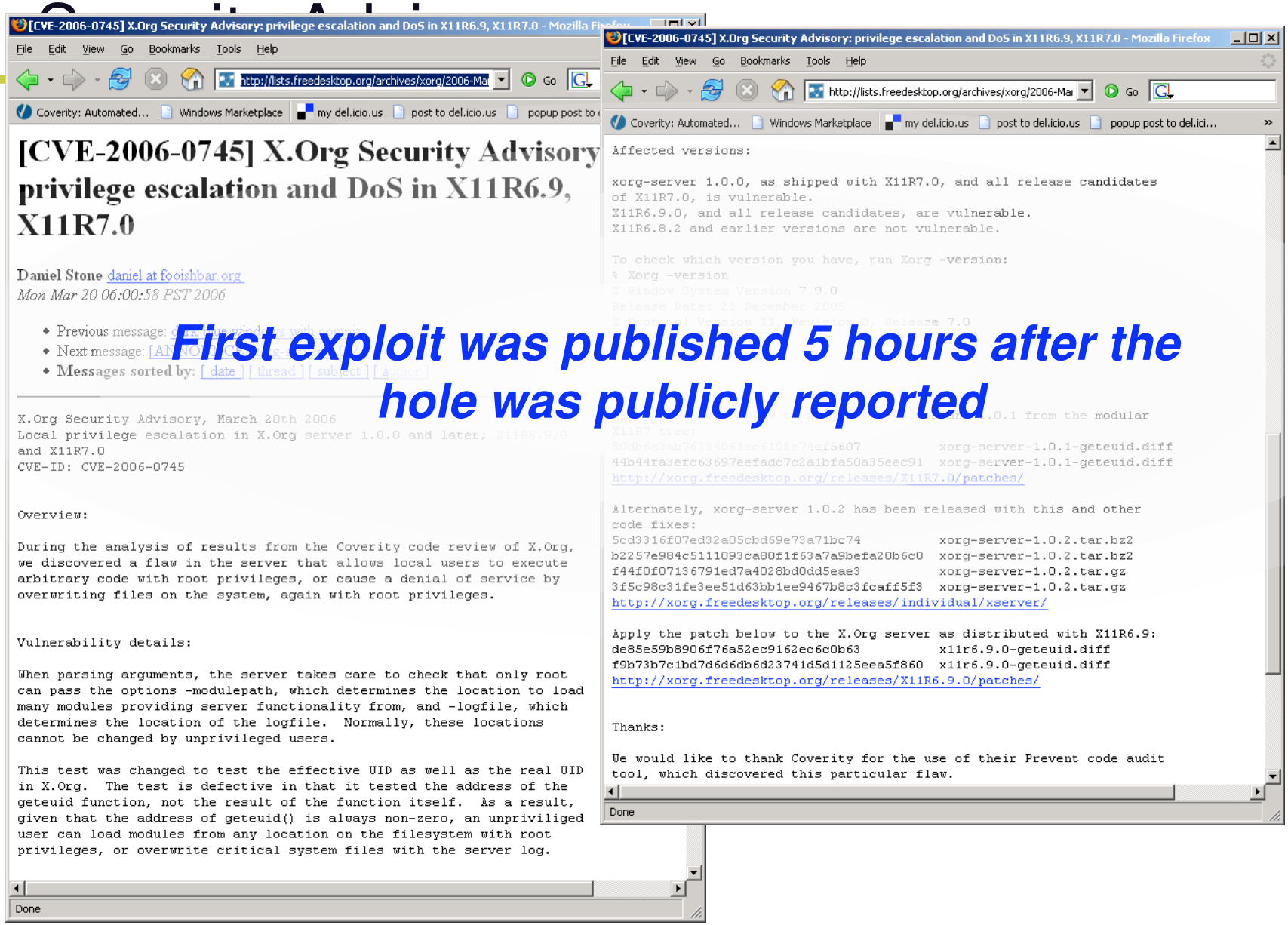
  So important marketing went to town:



```
if (getuid() != 0 &&
    geteuid == 0) {
     ErrorF(“    only root”);
   exit(1);
    }
```

Since without the parentheses, the code is simply checking to see if the geteuid function in libc was loaded somewhere other than address 0 (which is pretty much guaranteed to be true), it was reporting it was safe to allow risky options for all users, and thus a security hole was born.

- Alan Coopersmith, Sun Developer



First exploit was published 5 hours after the hole was publicly reported

One of the best stupid checks: Deadcode

- Programmer generally intends to do useful work

  Flag code where all paths to it are impossible or it makes no sense. Often serious logic bug.

- From UU aodv

  After send, take packet off queue. Bug: if any packet on list before the one we want will discard them!

```
// packet_queue.c:packet_queue_send
prev = null;
while(curr) {
    if(curr->dst_addr == dst_addr) {
        if(prev == NULL)
            PQ.head = curr->next;
        else
            …DEADCODE [prev never updated]…
```

Deadcode: Most serious error ever(?)

- Trial at chemotherapy machine company.

- During results meeting:

  Literally ran out to fix

  Note: heavily sanitized & simplified code.

```
enum Tube { TUBE0, TUBE1 };
void PickAndMix(int i) {
  enum Tube tfirst, tlast;

  if (TUBE0 == i) {
    tfirst=TUBE0;
    tlast=TUBE1;
  } else if (TUBE0 == i)  {
    tfirst=TUBE1;
    tlast=TUBE0;
  }
  MixDrugs(tfirst,tlast);
}
```

Best bugs: Cross-check code 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();   B();   B();  B();   // MAY: A() and B()
                           // must be paired
```

```
B(); // MUST: B() need not
     // be preceded by A()
```

  Check as MUST beliefs; rank errors by belief confidence.

Internal null: trivial, probably best checker.

- "*p" implies programmer believes p is not null
- A check (p == NULL) implies two beliefs:
  - **POST: p is null on true path, not null on false path**
  - **PRE: p was unknown before check**
- Cross-check beliefs: contradiction = error.

- 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;
```

- Nice thing about belief analysis: perspective.
  - **Natural to reason about: "Does this code make any sense?"**
  - **And once you do that, some very interesting errors…**
  - **X bug: Must know B is true but check**
  - **Chemo: Must know B is false, but check**
- If only read one of my papers, read this one:
  "Bugs as deviant behavior…" [sosp'01]

Overview

- Context
- The banal, vicious laws of reality and its cruel myths
- What actually matters?

- Academics meet reality, good and hard.

  The evils of non-determinism

- Bugs: often best come from analyzing programmer beliefs.

- Business factoids an academic finds amusing.

Technical can help social

- Tool has simple message: “No touch, low false positives, good bugs”
  - **Can explain it to mom? Then can explain to almost all sales guys & customers**
  - **Complicated? Population that understands much smaller.**
  - **This effect is not trivial.**
- Relationship therapy through tool “objectivity”
  - **UK company B outsources to India company A**
  - **B complains about A’s code quality. They fight.**
  - **Decide to use Coverity as arbitor. Happy. (I still can’t believe this.)**
- Wide tool use = seismic change in the last ~4 years.
  - **People get it. “Static” no longer = “huh?” or “lint” (i.e., suck)**
  - **Networking effects.**
  - **Result: Much much much easier to sell tools now.**

Some commercial experiences

- Surprise: Sales guys are great
  - **Easy to evaluate. Modular.**
- Careful what you wish for: bad competitor tools
  - **Time to sale ~ max(time for all competitors to do trial).**
  - **Worst case: tool sounds "great" but requires lot of hacking on build system**
  - **Take existing customer from really bad tool. Great? Well…**
  - **Culture = disdain rather than curiosity.**
  - **Social: Often have ugly processes in place in attempt to make tool usable.**
  - **Poetic justice: bad process left at your early-adopting customers!**
- But sometimes bad is good:
  - **Huge company: early on did 15+ trials across company, in end lost seven figure perpetual license deal. Sad faces.**
  - **Have since made *2-3x* off of them!**
  - **Company X bought license, next week fired 110 people. Bad?**
- VCs... Some are good, interesting people. Some are evil, and in dumb ways.

Some useful numbers

- Already seen:
  - **1000: number of bugs after which they baseline.**
  - **1.0: probability error labeled as FP if they don't understand**
  - **-m: slope of bug trend line for manager to get bonus.**
- Code numbers:
  - **12hr, 24hr: common upper bounds for analysis time**
  - **700 lines / second: ~speed of analysis to meet these times**
  - **10M: "large" code base**
- Bugs:
  - **3: number of attempts you can make to fix a bug in your tool**
  - **10: reduction in fix time if you assign blame for bugs**
- People:
  - **5 minutes before asymptotic decay in programmer interest**
  - **40: upper bound on active opportunities sales guy can manage**
  - **0: price of initial trial.**
  - **20K: not even worth it to charge per trial**

Academics don't understand money.

- "We'll just charge per seat like everyone else"
  - **Finish the story: "Company X buys three Purify seats, one for Asia, one for Europe and one for the US…"**
- Try #2: "we'll charge per lines of code"
  - **"That is a really stupid idea: (1) …, (2) … , … (n) …"**
  - **Actually works. I'm still in shock. Would recommend it.**
- Good feature for you the seller:
  - **No seat games. Revenue grows with code size. Run on another code base = new sale.**
- Good feature for buyer: No seat-model problems
  - **Buy once for project, then done. No per-seat or per-usage cost; no node lock problems; no problems adding, removing or renaming developers (or machines)**
  - **People actually seem to like this pitch.**

Laws of static bug finding

- Vacuous tautologies that imply trouble
  - **Can't find code, can't check.**
  - **Can't compile code, can't check.**
- A nice, balancing empirical tautology
  - **If can find code**
  - **AND checked system is big**
  - **AND can compile (enough) of it**
  - **THEN: will *always* find serious errors.**
- A nice special case:
  - **Check rule never checked? Always find bugs. Otherwise immediate kneejerk: what wrong with checker???**

Outline

- Context
- Experience. Assertions.

  Big problem 1: normal distributions. Not like the lab.
  Big problem 2: 10x reduction in knowledge in user base.

- Next: One of most consistently powerful tricks: belief analysis.

  Find errors where you don't know what the truth is.

  Infer rule.
  Infer the state of the system

  Old trick: but have used in every checker written since '01.
  Haven't seen any checker that wouldn't be improved.

- Summary

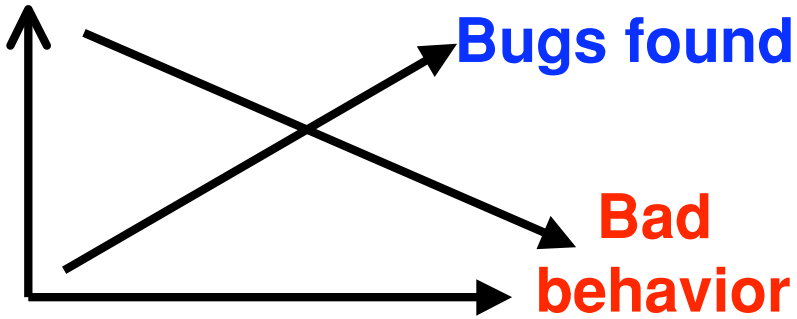
Static vs dynamic bug finding

- Static: precondition = compile (some) code.
  - **All paths + don't need to run + easy diagnosis.**
  - **Low incremental cost per line of code**
  - **Can get results in an afternoon.**
  - **10-100x more bugs.**
- Dynamic: precondition = compile all code + run
  - **What does code do? How to build? How to run?**
  - **Pros: on executed paths:**
    - **Runs code, so can check implications.**
    - **End-to-end check: all ways to cause crash.**
    - **Reasonable coverage: surprised when crash.**
- Result:
  - **Static better at checking properties visible in source, dynamic better at properties implied by source.**

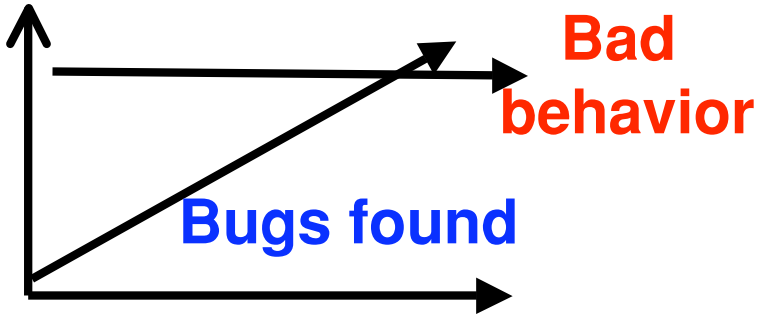
Assertion: Soundness is often a distraction

- Soundness: Find all bugs of type X.
  - **Not a bad thing. More bugs good.**
  - **BUT: can only do if you check weak properties.**
- What soundness really wants to be when it grows up:
  - **Total correctness: Find all bugs.**
  - **Most direct approximation: find as many bugs as possible.**
- Opportunity cost:
  - **Diminishing returns: Initial analysis finds most bugs**
  - **Spend time on what gets the next biggest set of bugs**
  - **Easy experiment: bug counts for sound vs unsound tools.**
- Soundness violates end-to-end argument:
  - **"It generally does not make much sense to reduce the residual error rate of one system component (property) much below that of the others."**

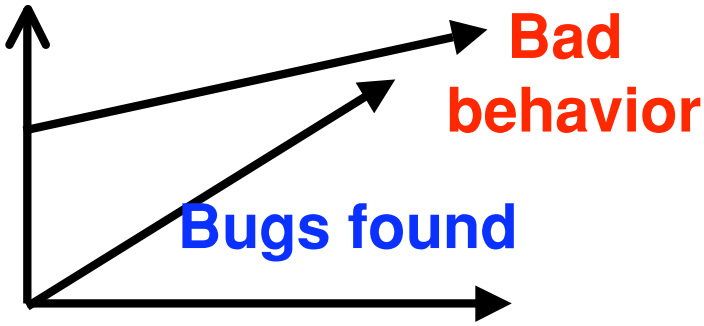
Open Q: Do static tools really help?



The optimistic hope



The null hypothesis



An Ugly Possibility

Danger: Opportunity cost.

Danger: Deterministic canary bugs to non-deterministic.

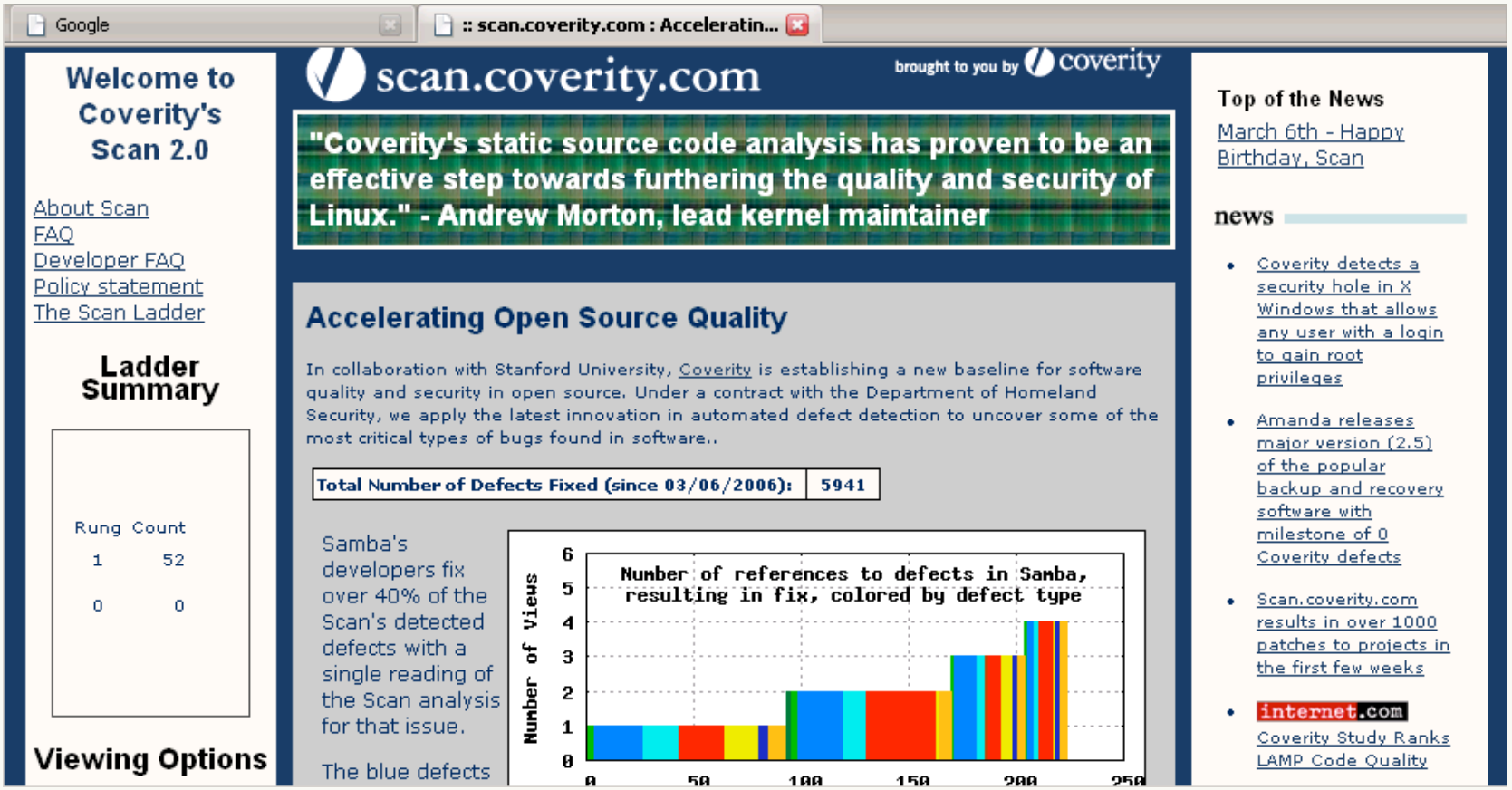
Open Q: how to get the bugs that matter?

- Myth: all bugs matter and all will be fixed
  - ***FALSE***
  - **Find 10 bugs, all get fixed. Find 10,000…**
- Reality
  - **Sites have many open bugs (observed by us & PREfix)**
  - **Myth lives because state-of-art is so bad at bug finding**
  - **What users really want: The 5-10 that "really matter"**
- General belief: bugs follow 90/10 distribution
  - **Out of 1000, 100 (10? or 1?) account for most pain.**
  - **Fixing 900+ waste of resources & may make things worse**
- How to find worst? No one has a good answer to this.
  - **Possibilities: promote bugs on executed paths or in code people care about, …**

Scan's One Year Anniversary

coverity

Website Relaunch on March 6th, 2007

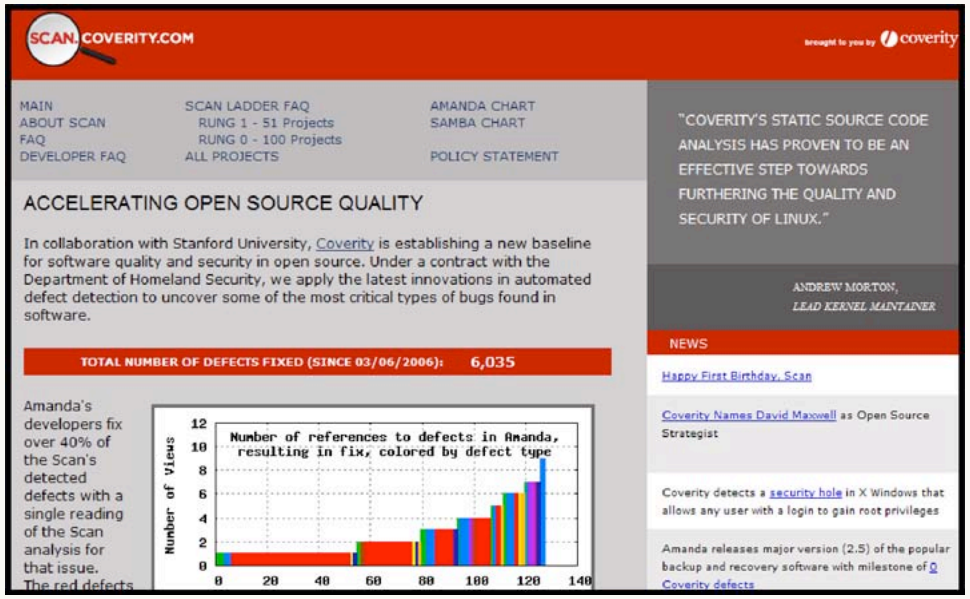


Authority on Open Source Code

coverity

Chosen by DHS to Harden Open Source

- Over 250 commonly used open source packages
- Over 55 Million LOC analyzed nightly on standard hardware
- Maintainers fixed over 7000 bugs and security violations to date



coverity

symantec.

History of Research&Growth of Coverity [2007:outdated]