Fault Localization and Profiling

The Story So Far ...

- Quality assurance is critical to software engineering.
  - Static and dynamic QA approaches are common
- Defect reports are tracked from creation to resolution
- Some are assigned to developers for resolution
- How do we know which part of a program to change to repair a bug or improve a program?

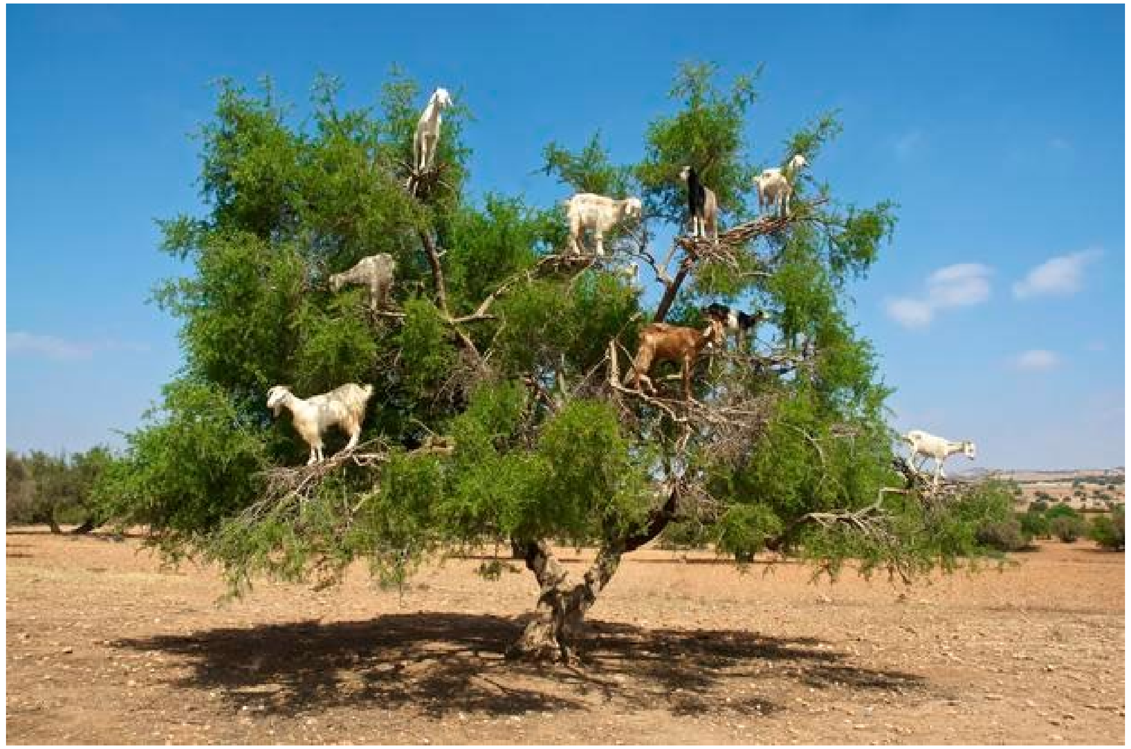
One-Slide Summary

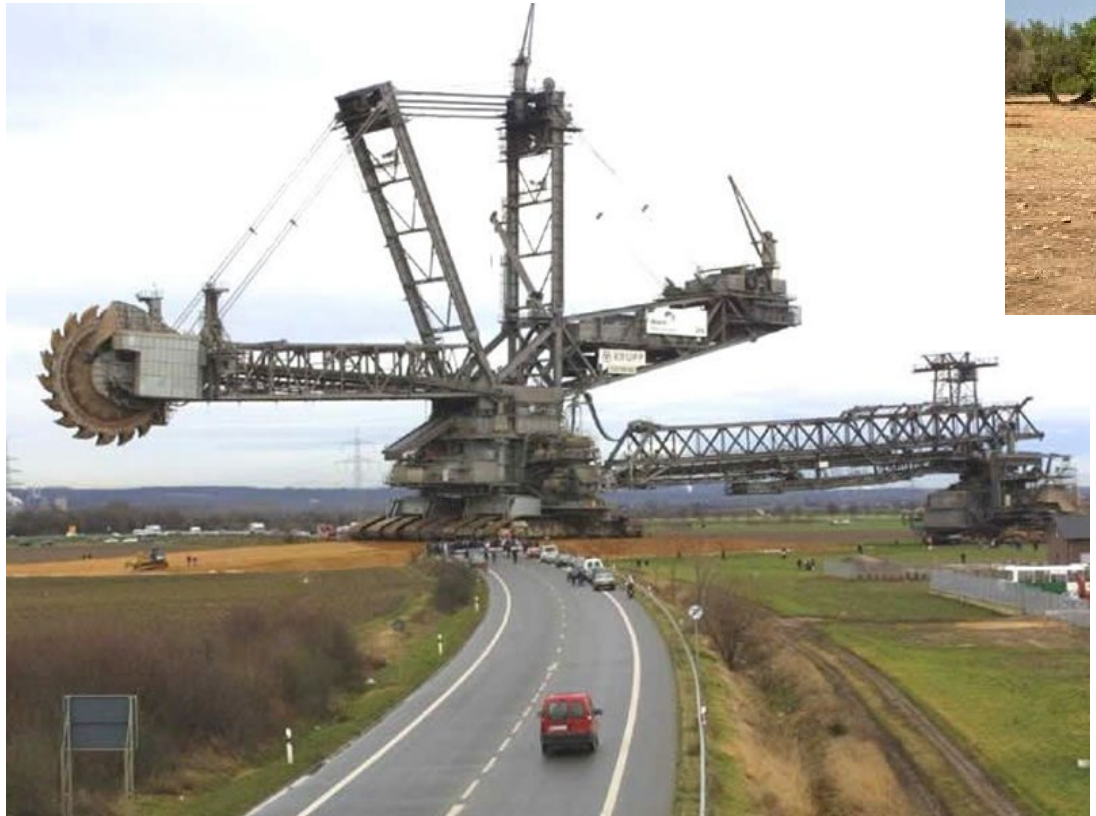
- A **debugger** helps to detect the source of a program error by single-stepping through the program and inspecting variable values.
- **Fault localization** is the task of identifying lines implicated in a bug. Humans are better at localizing some types of bugs than others.
- Automatic **tools** can help with the dynamic analyses of fault localization and profiling.
- Care must be taken when evaluating such tools (and their assumptions) for real-world use.

Outline

- Software Scales
- Manual Debuggers
- Human Study Results
- Automatic Tools
- Profilers
- Are Tools Helping?

Quick Quiz: Which of these is photoshopped?





Bucket-Wheel Excavators

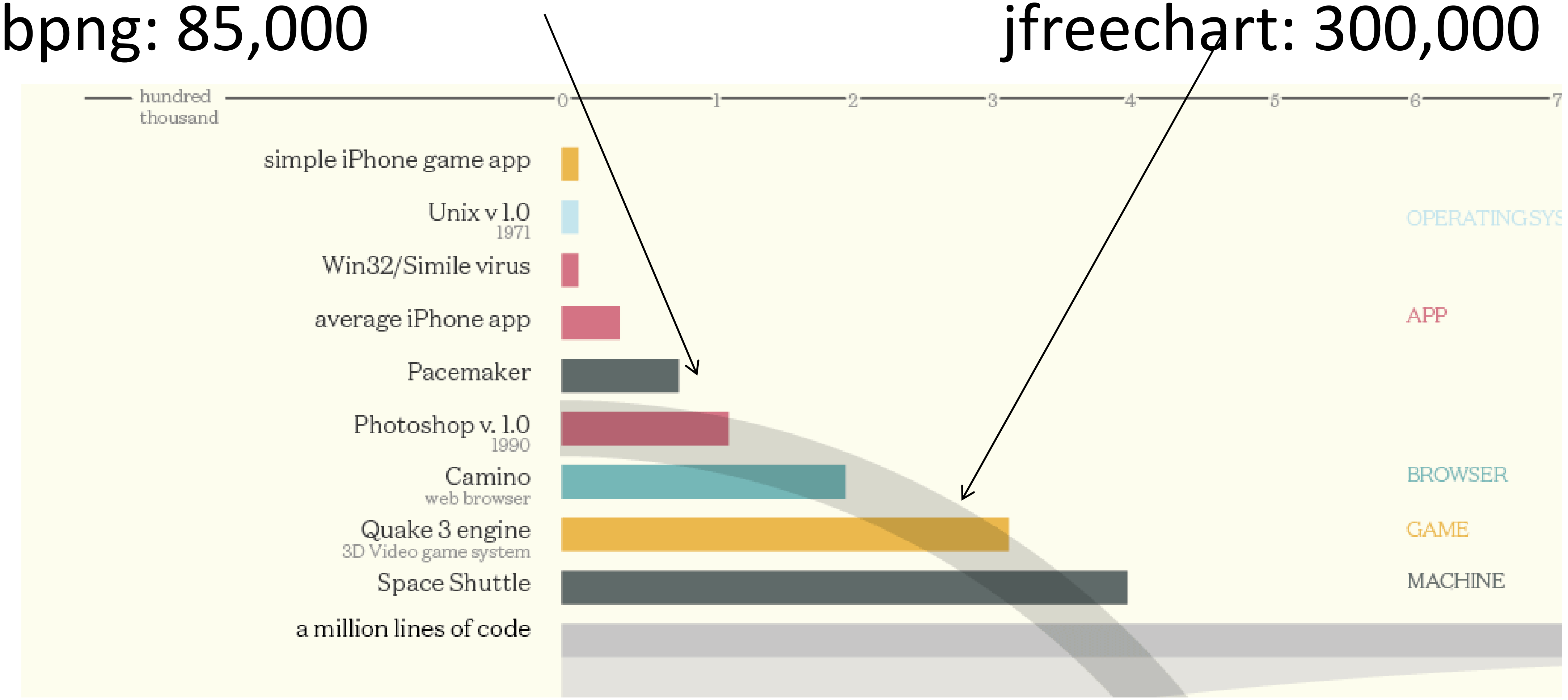
- Heaviest land vehicles
  - ~14,000 tons
    - (avg USA car: 2 tons)
  - Mobile strip-mining

Modern Software Is Huge?

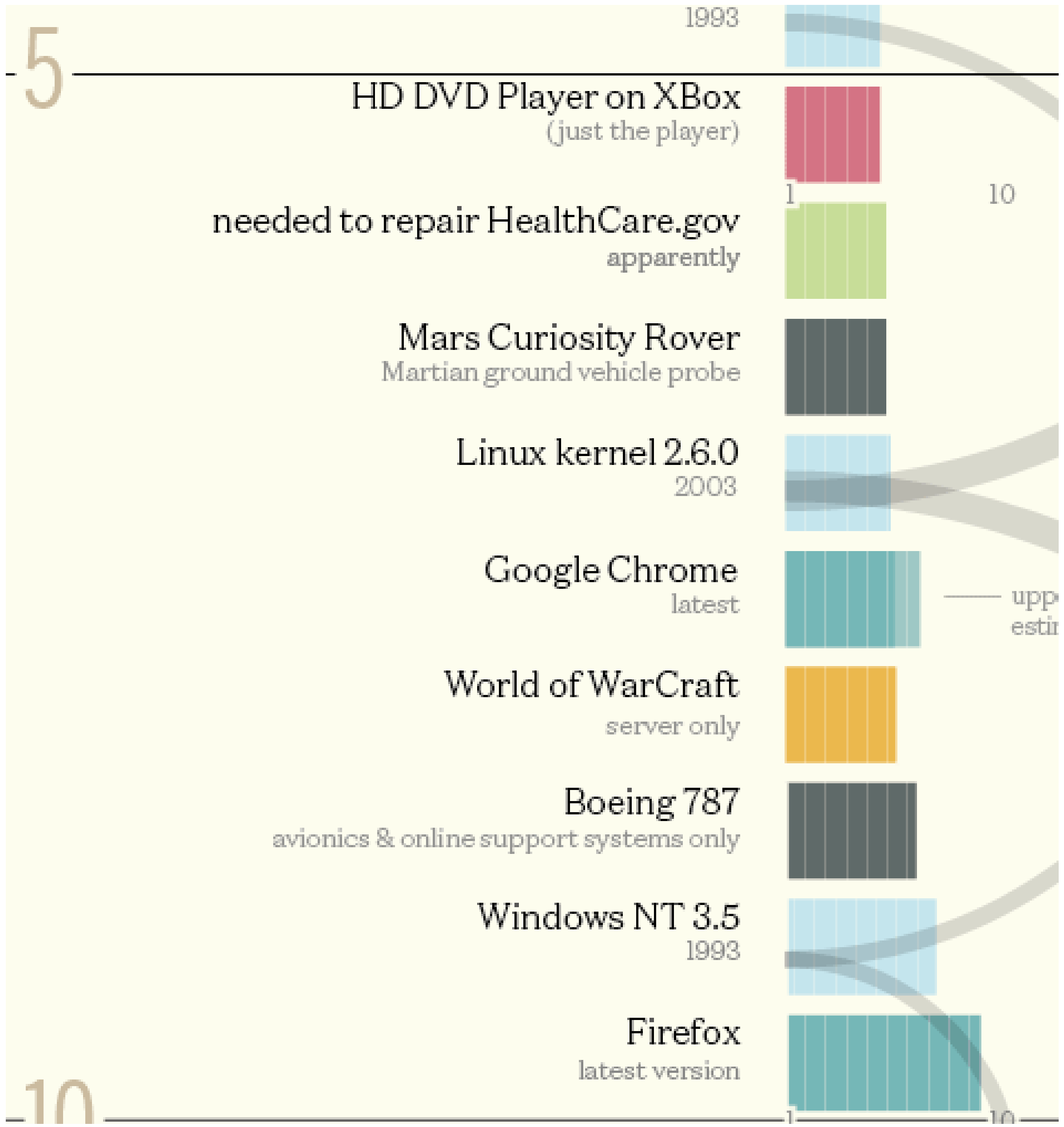
Example Programs: < 1MLOC

- libpng: 85,000

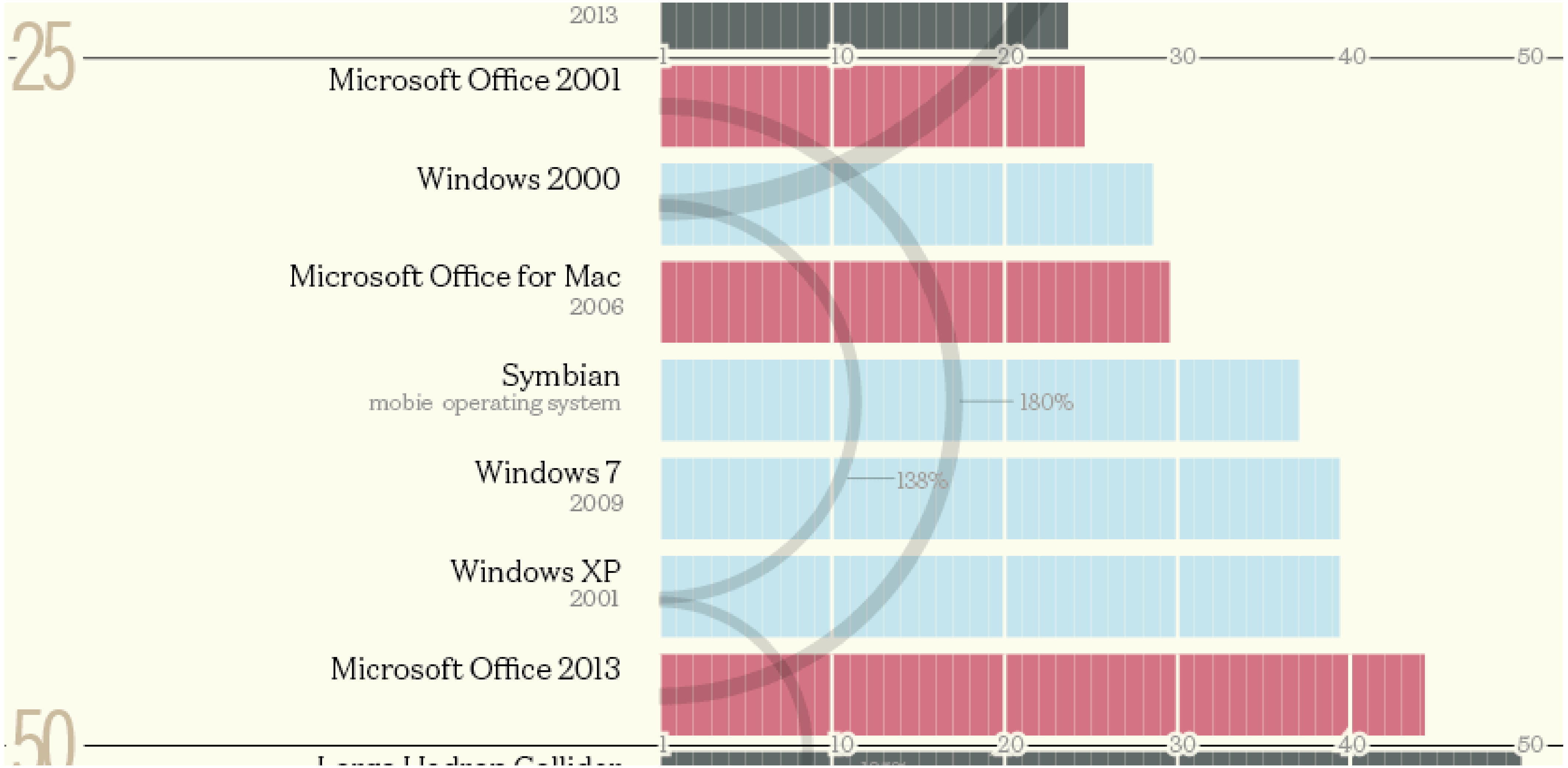
jfreechart: 300,000



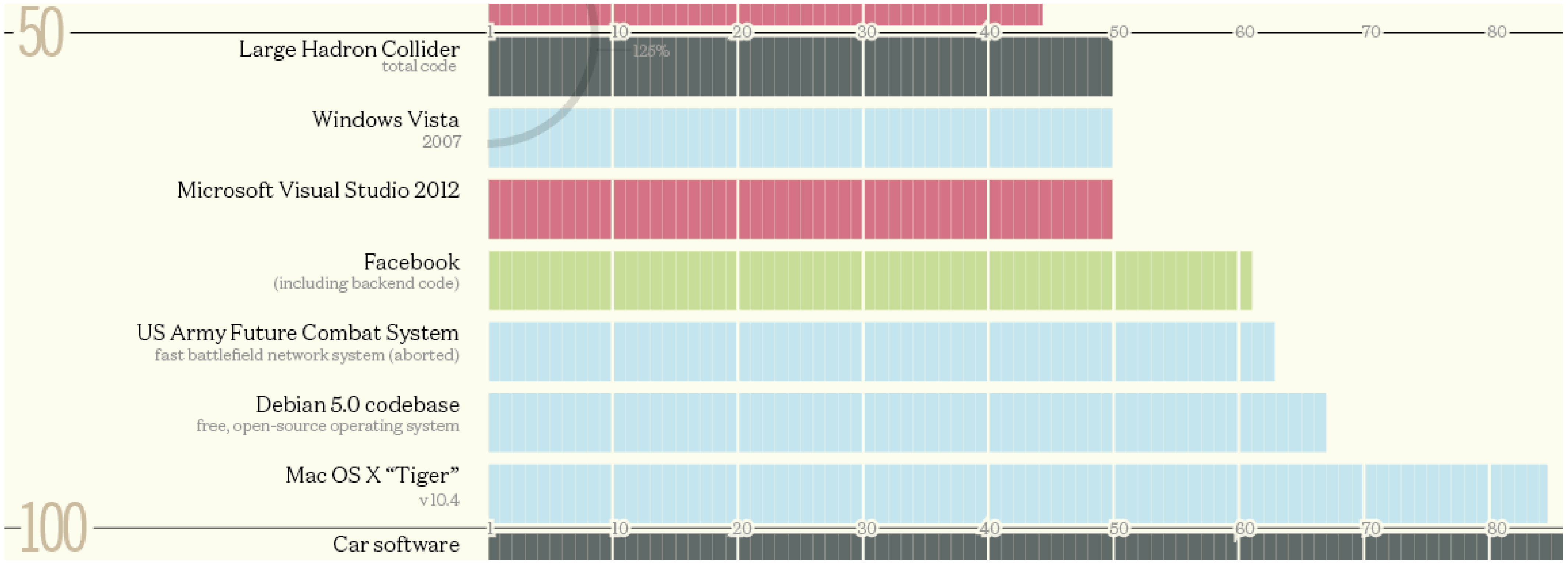
Example Programs: 5-10 MLOC



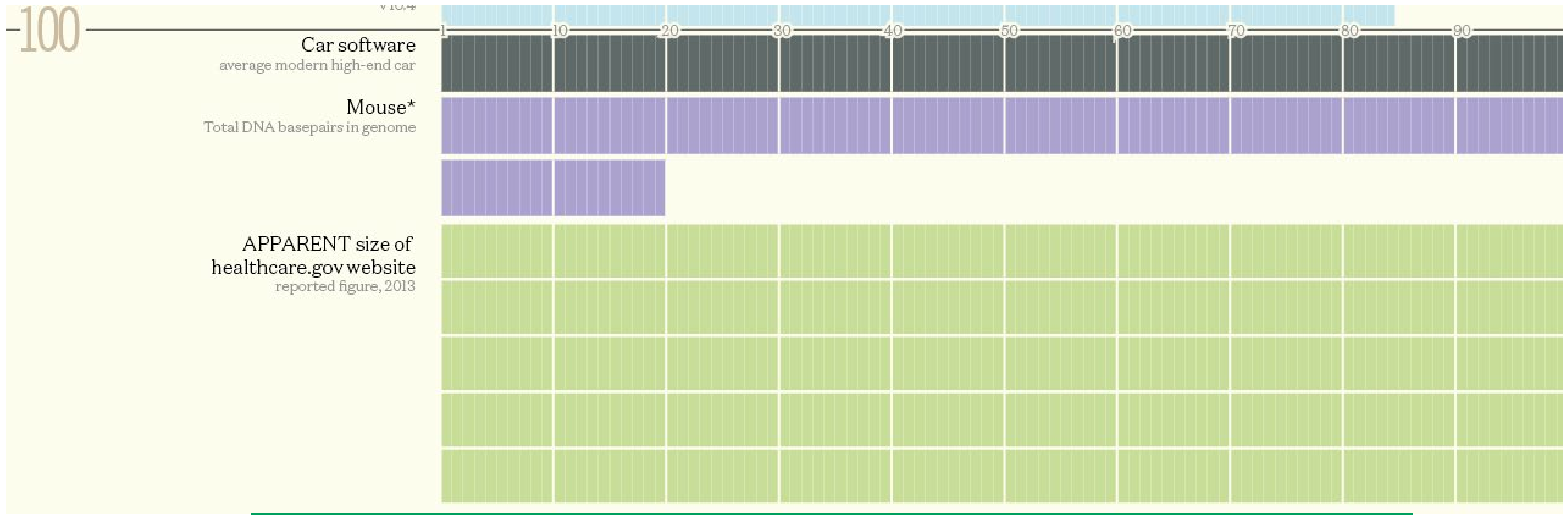
Example Programs: 25 – 50 MLOC

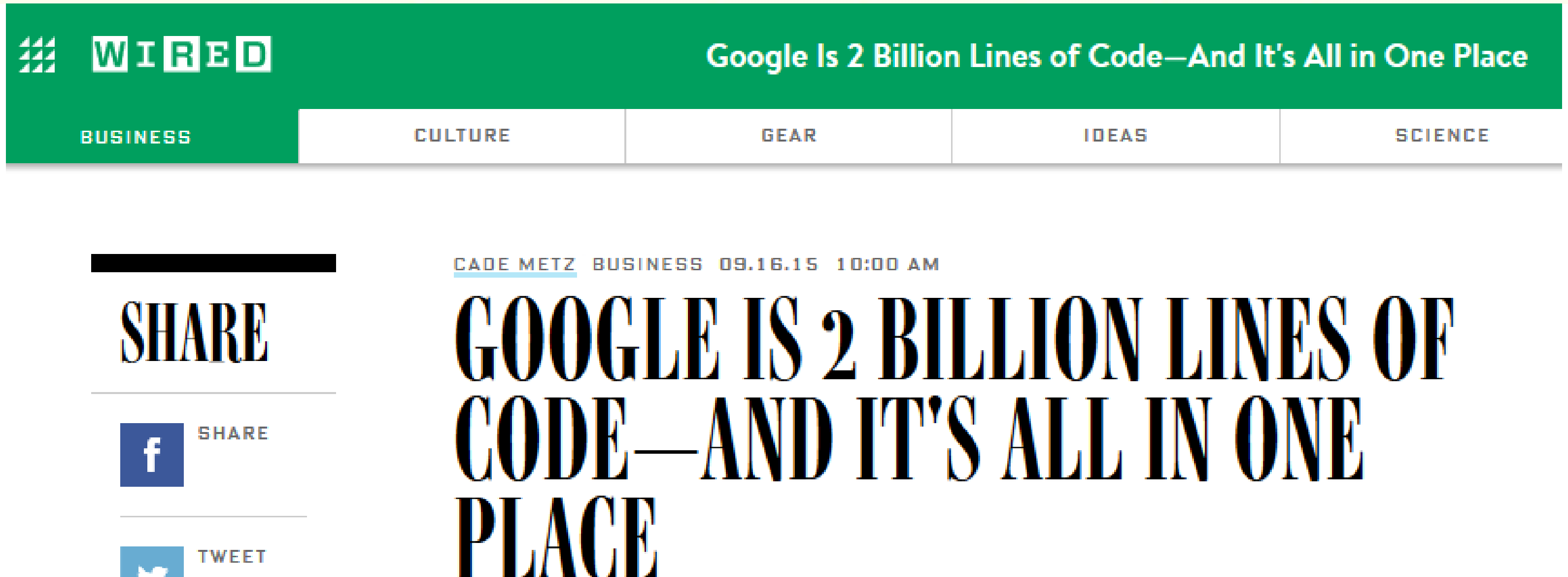


Example Programs: 50 – 100 MLOC



Example Programs: 0.1 – 2.0BLOC





Google Is 2 Billion Lines of Code—And It's All in One Place

CADE METZ BUSINESS 09.16.15 10:00 AM

GOOGLE IS 2 BILLION LINES OF CODE—AND IT'S ALL IN ONE PLACE

Modern Software Is Huge!

- Who cares?
  - Techniques developed based on smaller code bases simply do *not* apply or scale to larger code bases
    - Techniques from the 1980s or your habits from classes

Humans Are Poor At Comprehending Large Scales

- libpng 85 000
- google 2 000 000 000
- Imagine that there is a bug somewhere, anywhere, in libpng
- You can find it in a minute (assume!)!
- At that same rate, it will take you *more than two weeks* to find it in all of google
  - A one-hour bug on libpng is three years on google
  - Unless we do things differently …

Fault Localization

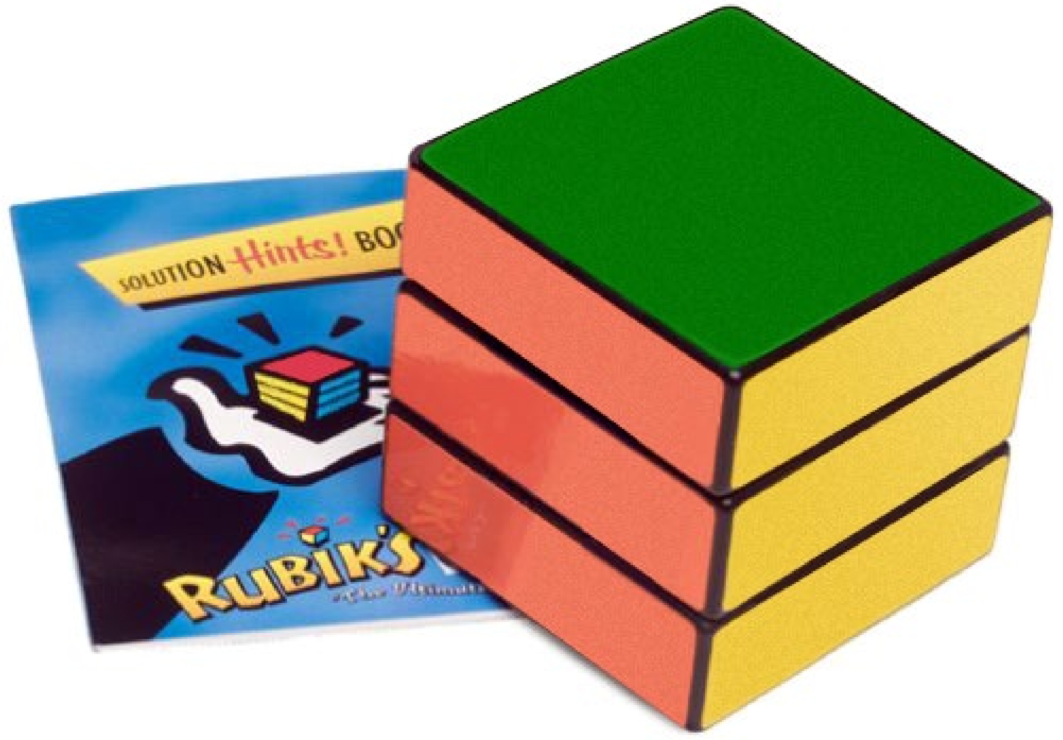
- **Fault localization** is the task of identifying source code regions implicated in a bug
    - "This regression test is failing. Which lines should we change to fix things?"
- Answer is not unique: there are often many places to fix a big
    - Example: check for null at caller or callee?
- Debugging includes fault localization
- Answer may take the form of a list (e.g., of lines) ranked by suspiciousness

What is a Debugger?

- “A software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.”
  - - Microsoft Developer Network

Debuggers

- Can operate on source code or assembly code
- Inspect the values of registers, memory
- Key Features (we'll explain all of them)
  - Attach to process
  - Single-stepping
  - Breakpoints
  - Conditional Breakpoints
  - Watchpoints



How to design a basic debugger?

Signals

- A **signal** is an asynchronous notification sent to a process about an event; A **software interrupt delivered to a process by OS**.
  - User pressed Ctrl-C (or did **kill %pid**)
    - Or asked the Windows Task Manager to terminate it
  - Exceptions (divide by zero, null pointer)
  - From the OS

I Am Devloper
@iamdevloper

Roses are red
And so are you
Violets are blue
Asynchronous operations
are great

15:53 · 14 Feb 19 · Twitter Web App

1,993 Retweets 6,207 Likes

- You can install a **signal handler** – a procedure that will be executed when the signal occurs.
  - Signal handlers are vulnerable to race conditions. Why?

Signal Example

- What does this program print?
  - SIGSEGV
    - Signal for segmentation fault
  - Signal()
    - Specifies a function to be executed when the program receives a given signal.

```
#include <stdio.h>
#include <signal.h>

int global = 11;

int my_handler() {
  printf("In signal handler, global = %d\n",global);
  exit(1);
}

void main() {
  int * pointer = NULL;

  signal(SIGSEGV, my_handler) ;

  global = 33;

  * pointer = 0;

  global = 55;

  printf("Outside, global = %d\n", global);
}
```

Attaching A Debugger

```
(gdb) attach 13769
Attaching to process 13769
```

- Requires operating system support
- There is a special **system call** that allows one process to act as a debugger for a target
  - What are the security concerns?
- Once this is done, the debugger can basically “catch signals” delivered to the target
  - This isn’t exactly what happens, but it’s a good explanation …

Building a Debugger

- We can then get breakpoints and interactive debugging
    - Attach to target
    - Set up signal handler
    - Add in exception-causing instructions
    - Inspect globals, etc.
    - #define BREAKPOINT *(0)=0:
        - Macro to replace BREAKPOINT with *(0) = 0

```
#include <stdio.h>
#include <signal.h>

#define BREAKPOINT *(0)=0

int global = 11;

int debugger_signal_handler() {
  printf(“debugger prompt: \n”);
  // debugger code goes here!
}

void main() {
  signal(SIGSEGV, debugger_signal_handler) ;

  global = 33;

  BREAKPOINT;

  global = 55;

  printf("Outside, global = %d\n", global);
}
```

Advanced Breakpoints

- Optimization: **hardware breakpoints (HBP)**
  - Special register: if PC value = HBP register value, signal an exception
  - Faster than software, works on ROMs, only limited number of breakpoints, etc.
- Feature: **conditional breakpoint**: "break at instruction X if some_variable = some_value"
  - As before, but signal handler checks to see if some_variable = some_value
  - If so, present interactive debugging prompt
  - If not, return to program immediately
  - Is this fast or slow?

Single-Stepping

- Debuggers also allow you to advance through code one instruction at a time
- To implement this, put a breakpoint at the first instruction (= at program start)
- The "**single step**" or "next" interactive command is equal to:
  - Put a breakpoint at the next instruction
  - Resume execution
  - (No, really.)

Watchpoints

- You want to know when a variable changes
- A **watchpoint** is like a breakpoint, but it stops execution after any instruction changes the value at location L
- How could we implement this?

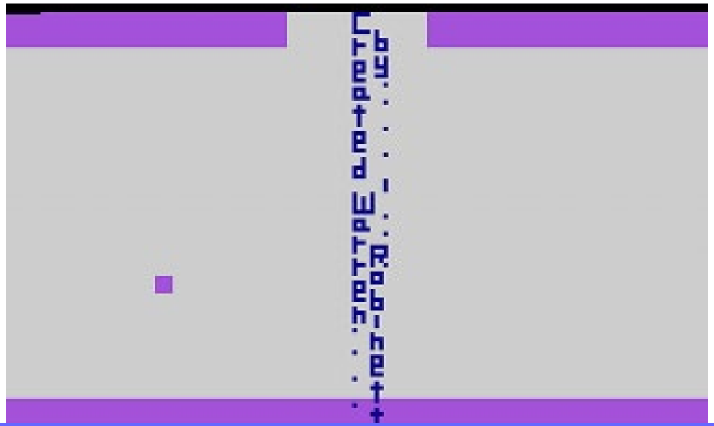
DID YOU WATCH THE MOVIE ON TV LAST NIGHT?
NOPE.
DID YOU WATCH THE GAME THEN?
NOPE.
DID YOU WATCH ANY TV LAST NIGHT?
NOPE.
THEN WHAT DID YOU WATCH?
WATTERSON

Watchpoint Implementation

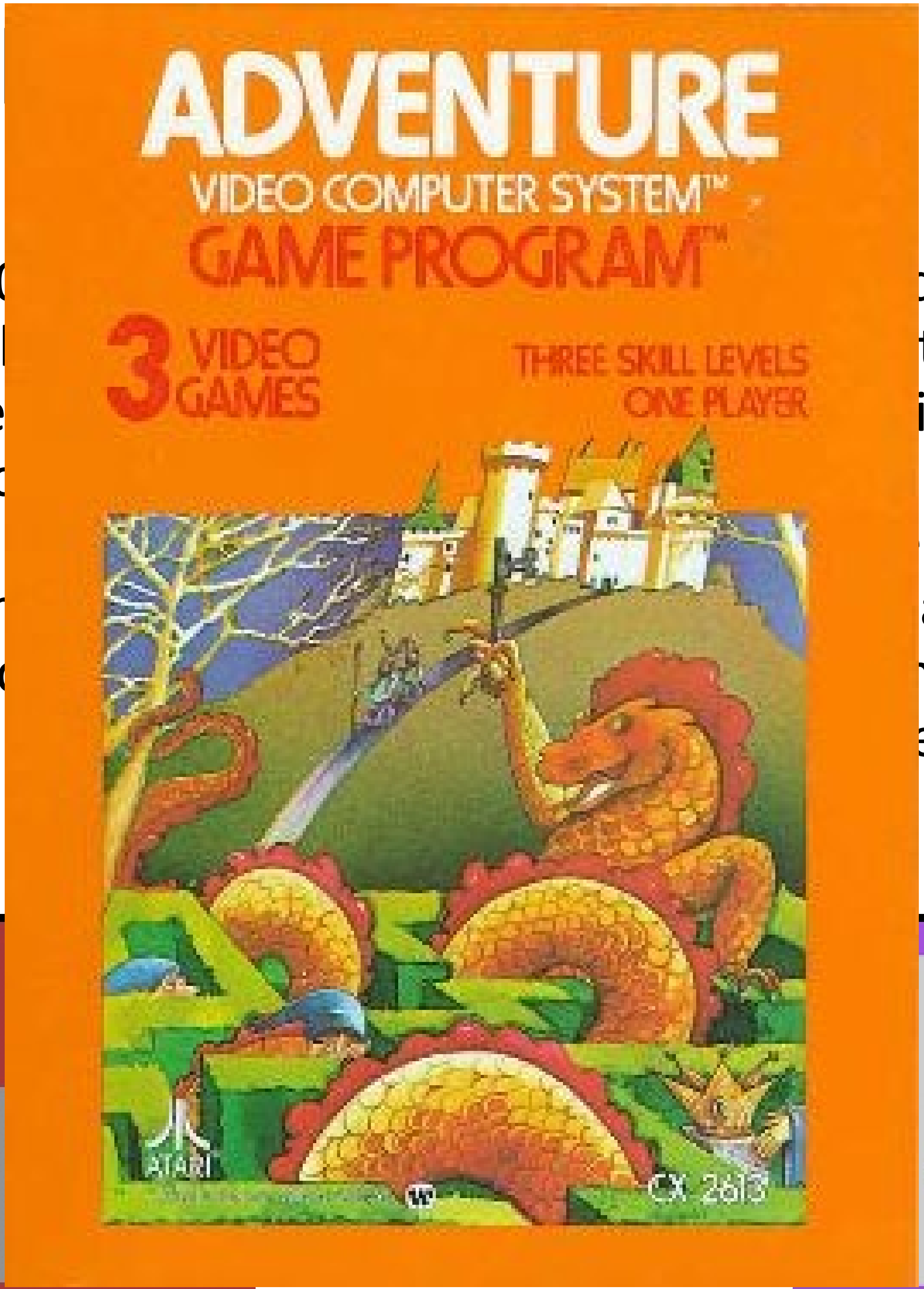
- Software Watchpoints
  - Put a breakpoint at *every instruction* (ouch!)
  - Check the current value of L against a stored value
  - If different, give interactive debugging prompt
  - If not, set next breakpoint and continue (single-step)
- Hardware Watchpoints
  - Special register holds L: if the value at address L ever changes, the CPU raises an exception

Video Game History

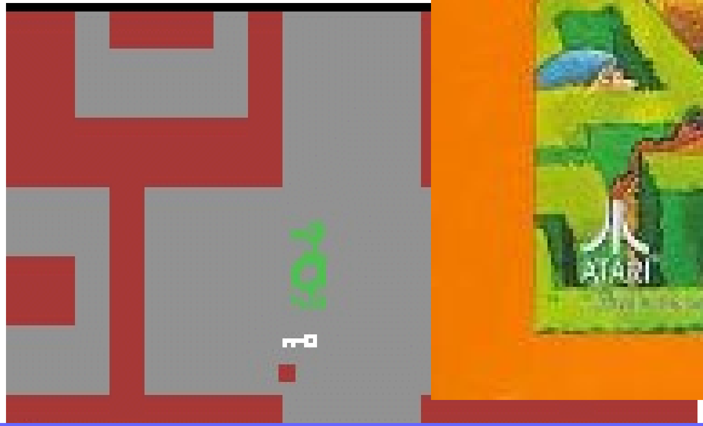
- This 1979-1980 Atari 2600 video game introduced the first widely-known Easter egg. At the time, Atari did not allow game designers or programmers to credit themselves in any way (games were marketed and branded as produced by Atari overall). Warren Robinett included a secret room crediting himself as the designer. When a 15-year-old from Utah discovered it and wrote to Atari for an explanation, they tasked Brad Stewart to fix it, but he said he would only change it to “Fixed by Brad Stewart”. Atari decided to leave it in game, dubbing such hidden features *Easter eggs* and saying they would include more in the future. The game itself involves carrying items around three castles to defeat three dragons.

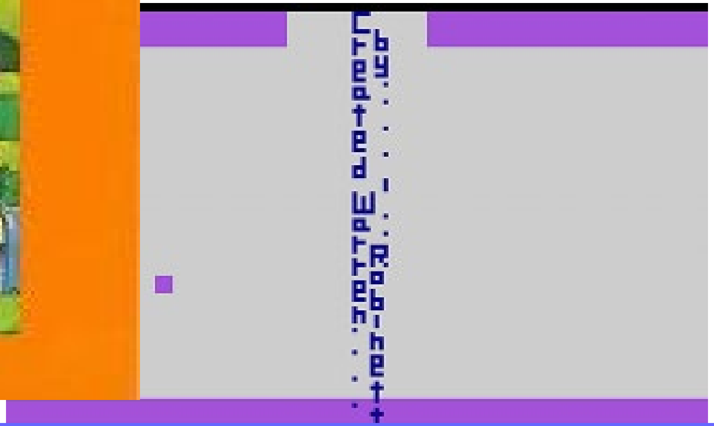


Video Game



- This 1979-1980 Atari 26 dely-known Easter egg. At
  the time, Atari did not al to credit themselves in any
  way (games were marke i overall). Warren Robinett
  included a secret room c n a 15-year-old from Utah
  discovered it and wrote d Brad Stewart to fix it, but
  he said he would only ch ari decided to leave it in
  game, dubbing such hidc hey would include more in
  the future. The game its ee castles to defeat three
  dragons.



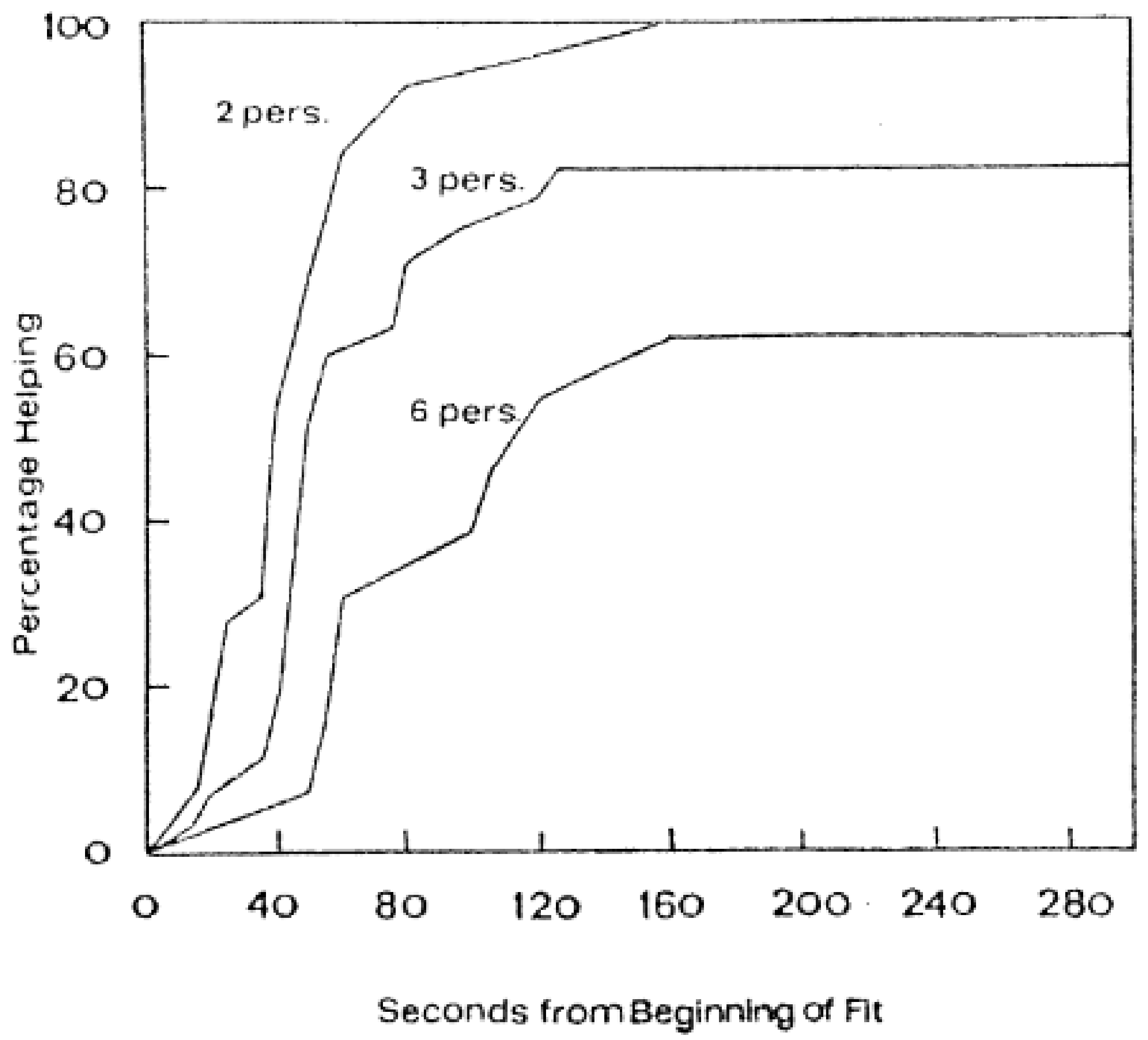


Psychology: Reactions

- You are invited to participate in a group discussion of “personal problems”. Because of the sensitive nature of the discussion, it takes place over an intercom. During the discussion, you hear:
  - “I-er-um-I think I-I need-er-if-if could-er-er-somebody er-er-er-er-er-er-er give me a little-er-give me a little help here because-er-I-er-I’m-er-erh-h-having a-a-a real problem-er-right now and I-er-if somebody could help me out it would-it would-er-er s-s-sure be-sure be good . . . because-there-er-er-a cause I-er-I-uh-I’ve got a-a one of the-er-sei er-er-things coming on and-and-and I could really-er-use some help so if somebody would-er-give me a little h-help-uh-er-er-er-er-er c-could somebody-er-er-help-er-uh-uh-uh (choking sounds). . . . I’m gonna die-er-er-I’m . . . gonna die-er-help-er-er-seizure-er-[chokes, then quiet].”

Psychology: Reactions

- The more people in the discussion, the longer it takes anyone to take action
- Gender (of you or others) had no effect

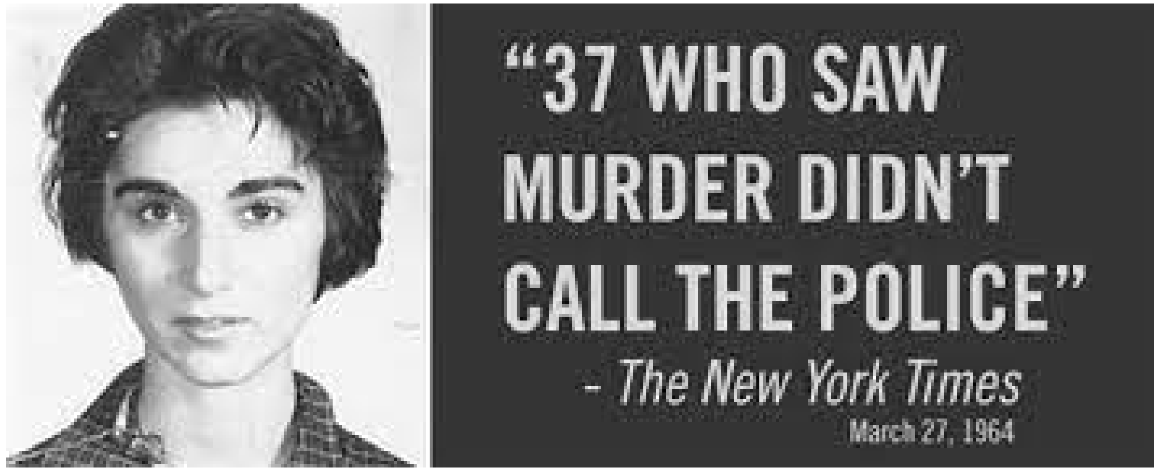


Bystander Effect

- "It is our impression that non-intervening subjects not decided *not* to respond. Rather they were still in a state of indecision and conflict concerning whether to respond or not. The emotional behavior of these nonresponding subjects was a sign of their continuing conflict ..."
- Motivated by 1964 attack on Kitty Genovese in residential New York: rape and murder took 30+ minutes and had 37(?) witnesses → no one came out to help

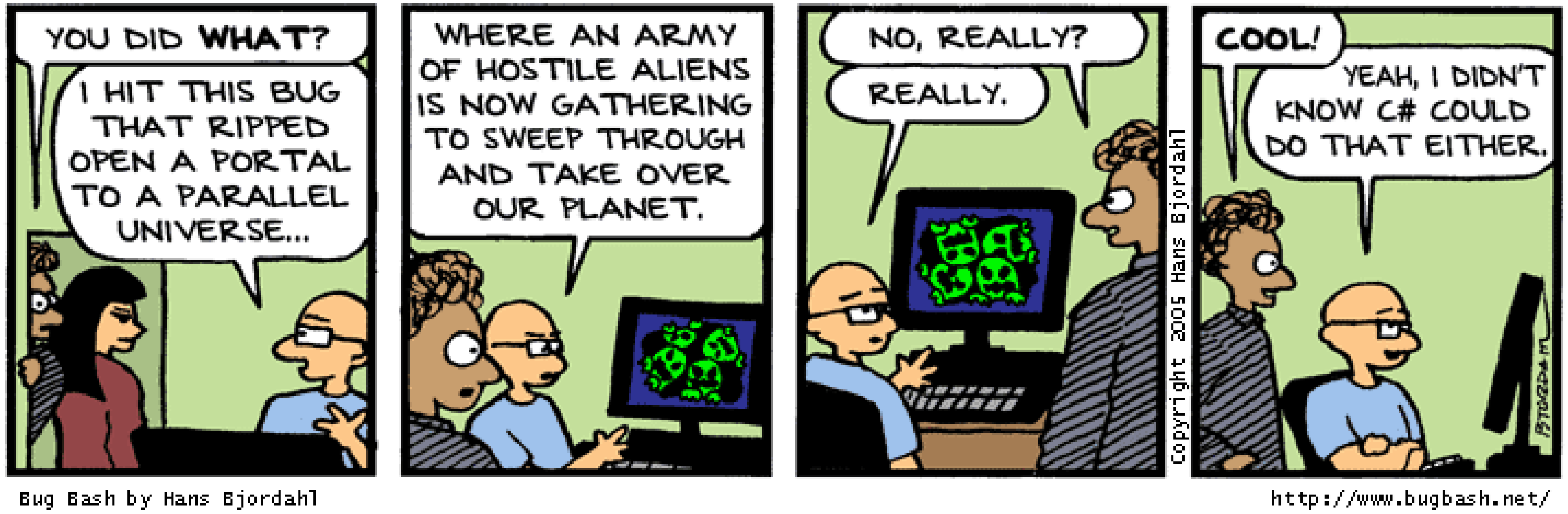
Bystander Effect

- [Darley and Latane. Bystander Intervention in Emergencies: Diffusion of Responsibility. J. Personality and Social Psych. 8(4) 1968.]
- Implications for SE: Team sizing considerations. Who will volunteer to be assigned this bug?



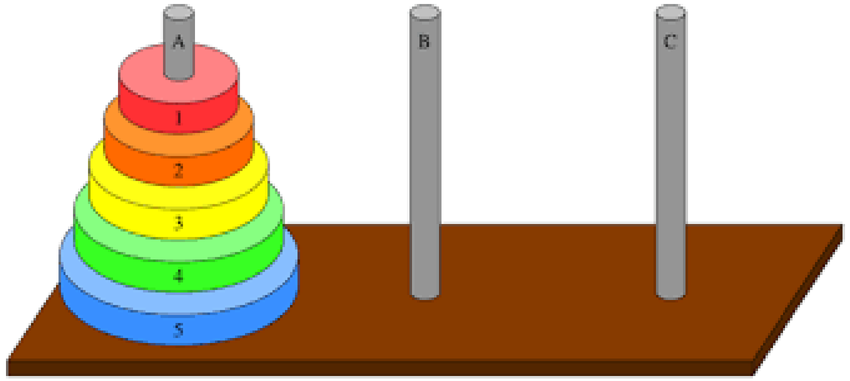
Human Fault Localization

- OK, so humans have debugg*ers*
- Are humans any good at debugg*ing*?
- Not all bugs are equally easy to find
- Not all programs are equally easy to debug


YOU DID WHAT?
I HIT THIS BUG THAT RIPPED OPEN A PORTAL TO A PARALLEL UNIVERSE...
WHERE AN ARMY OF HOSTILE ALIENS IS NOW GATHERING TO SWEEP THROUGH AND TAKE OVER OUR PLANET.
NO, REALLY?
REALLY.
COOL!
YEAH, I DIDN'T KNOW C# COULD DO THAT EITHER.
Copyright 2005 Hans Bjordahl
Bug Bash by Hans Bjordahl
http://www.bugbash.net/

Find The Bug: Tower of Hanoi

- Over 53% of participants (seniors) could find the bug in about 3 minutes
- Note: conditional branches, recursive calls, rich comments, variable names



```
/*******************************************
  Performs the initial call to moveTower
  to solve the puzzle.  Moves the disks
  from tower 1 to tower 3 using tower 2.
*******************************************/
public void solve () {
  moveTower (totalDisks, 1, 3, 2);
}

/*******************************************
  Moves the specified number of disks
  from one tower to another by moving a
  subtower of n-1 disks out of the way,
  moving one disk, then moving the
  subtower back. Base case of 1 disk.
*******************************************/
private void moveTower (int numDisks,
                int start, int end, int temp) {
if (numDisks == 1)
    moveTower(numDisks-1, temp, end, start);
  else {
    moveTower (numDisks-1, start, temp, end);
    moveOneDisk (start, end);
    moveTower (numDisks-1, temp, end, start);
  }
}
/*******************************************
  Prints instructions to move one disk
  from the specified start tower to the
  specified end tower.
*******************************************/
private void moveOneDisk (int start, int end) {
  System.out.println ("Move one disk from "
      + start + " to " + end);
}
```

Find The Bug 2

- Only 33% could locate the bug
- Note: shorter, simpler identifiers, simpler control flow, not as abstract

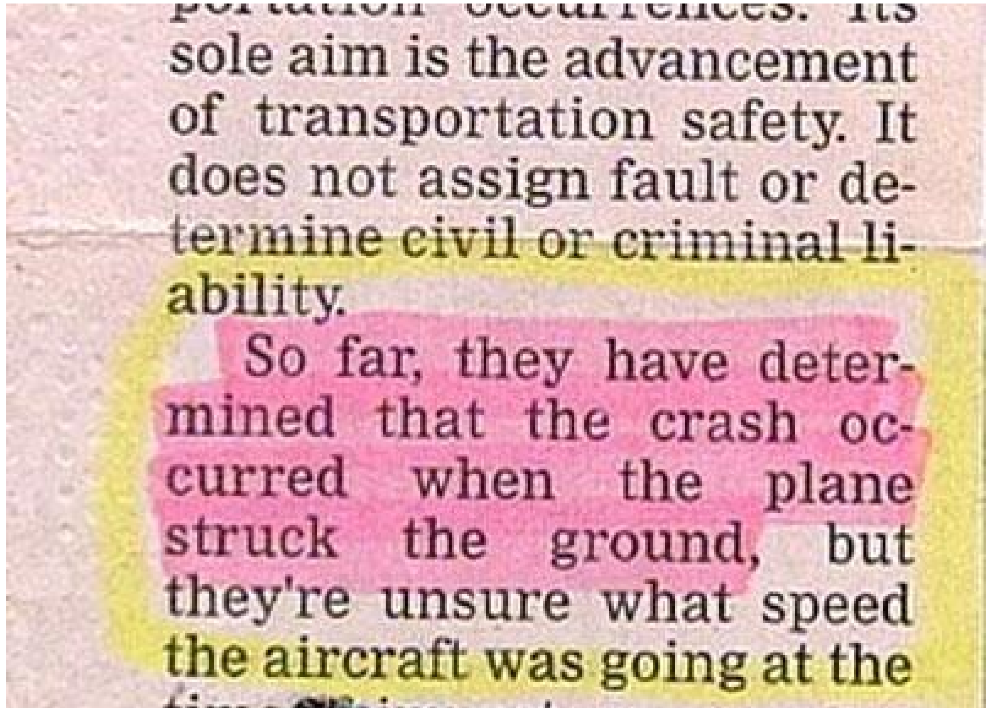
```java
/** Move a single disk from src to dest. */
public static void hanoi1(int src, int dest){
  System.out.println(src + " => " + dest);
}
/** Move two disks from src to dest,
    making use of a spare peg. */
public static void hanoi2(int src,
                    int dest, int spare) {
  hanoi1(src, dest);
  System.out.println(src + " => " + dest);
  hanoi1(spare, dest);
}
/** Move three disks from src to dest,
    making use of a spare peg. */
public static void hanoi3(int src,
                    int dest, int spare) {
  hanoi2(src, spare, dest);
  System.out.println(src + " => " + dest);
  hanoi2(spare, dest, src);
}
```

Human Study

- Participants (n=65, half with >4 years of experience) were shown snippets of textbook
  - Defects seeded based on 100 consecutive bug fixes from the Mozilla bug repository
- Double experimental control
  - Quicksort in Textbook A vs. Textbook B has the same complexity (differs only in style)
  - Bubblesort in Textbook A vs. AVL Tree in Textbook A differ in complexity (have same presentation style)
  - [Z. Fry et al.: A Human Study of Fault Localization Accuracy. International Conference on Software Maintenance (ICSM) 2010]

What Do You Think?

- Rank these: which of these bugs is easiest for humans to find?
  - Extra Assignment
  - Missing Statement
  - Extra Conditional
  - Calling Wrong Method
  - Extra Statement


sole aim is the advancement of transportation safety. It does not assign fault or determine civil or criminal liability.

So far, they have determined that the crash occurred when the plane struck the ground, but they're unsure what speed the aircraft was going at the

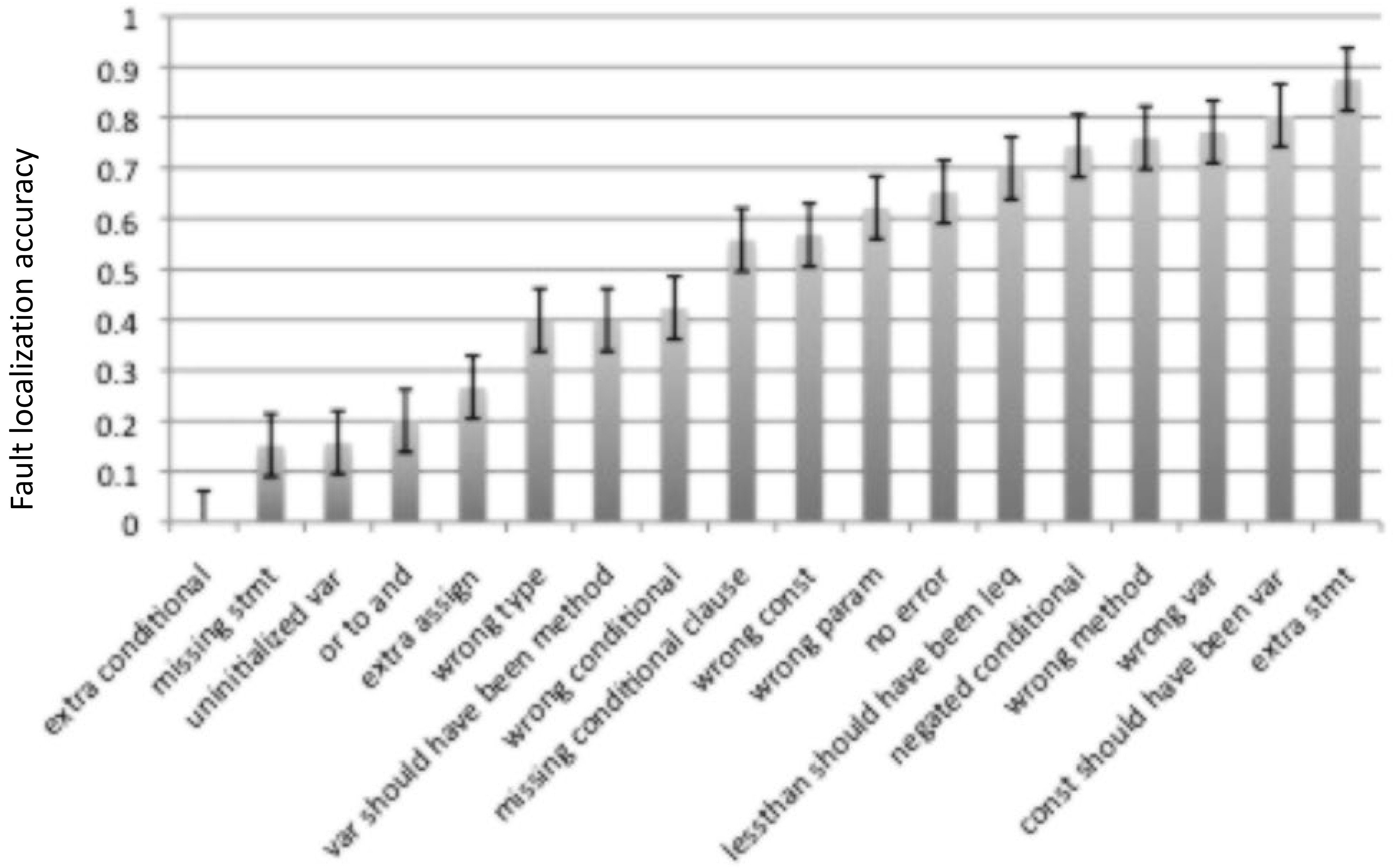


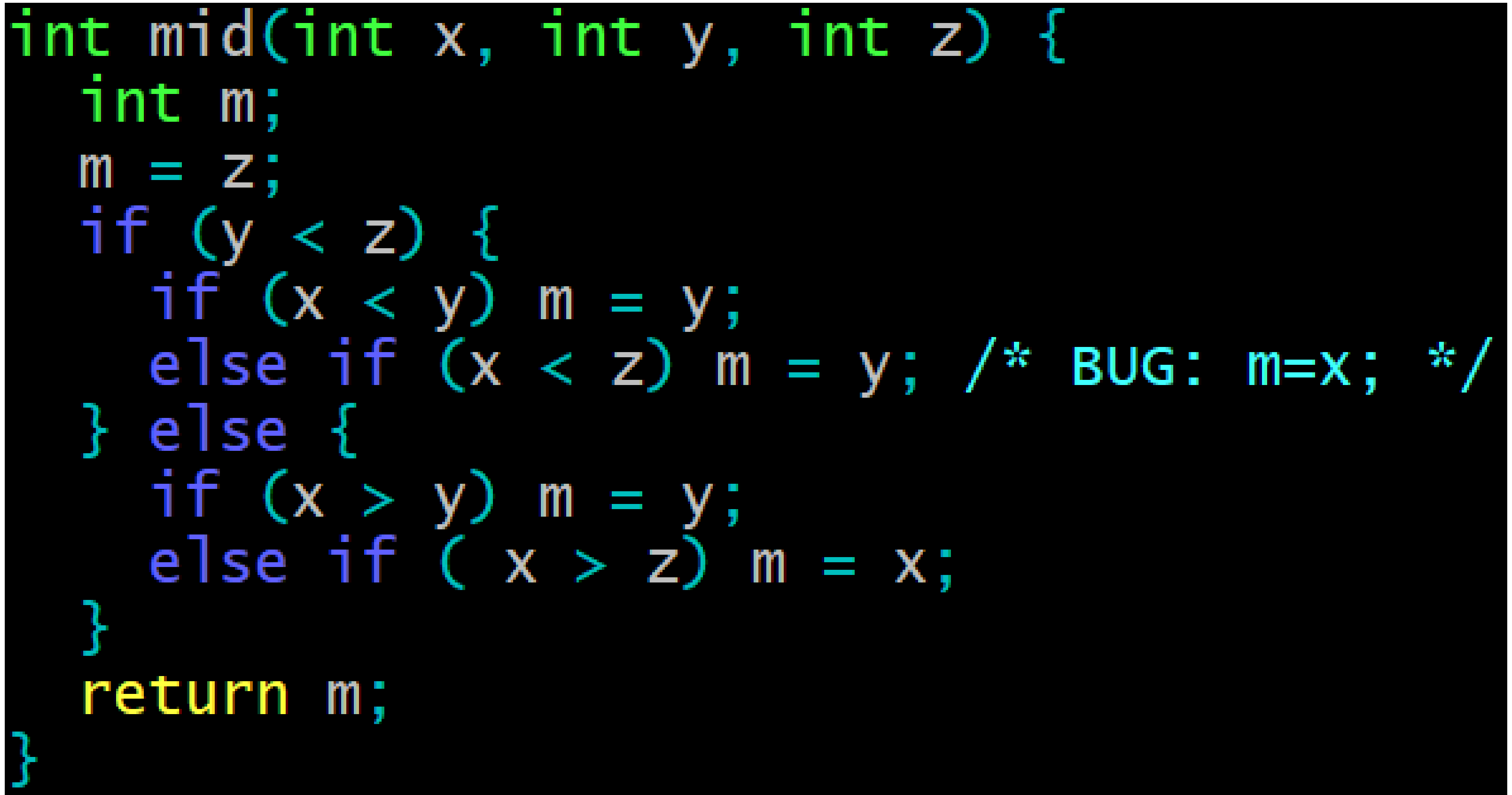
Fig. 3. Human fault localization accuracy as a function of defect type.

Tool Support for Fault Localization

- A **spectrum-based fault localization** tool uses a dynamic analysis to rank suspicious statements implicated in a fault by comparing the statements covered on failing tests to the statements covered on passing tests
- Basic idea:
  - Instrument the program for coverage (put print statements everywhere)
  - Run separately on normal inputs and bug-inducing inputs
  - Compute the set difference on coverage!

Fault Localization Example

- Consider this simple buggy program:



```
int mid(int x, int y, int z) {
  int m;
  m = z;
  if (y < z) {
    if (x < y) m = y;
    else if (x < z) m = y; /* BUG: m=x; */
  } else {
    if (x > y) m = y;
    else if ( x > z) m = x;
  }
  return m;
}
```

Coverage-Based Fault Localization

Statement	**3,3,5**	**1,2,3**	**3,2,1**	**3,2,1**	**5,5,5**	**2,1,3**
int m;	🟥	🟥	🟥	🟥	🟥	🟥
m = z;	🟥	🟥	🟥	🟥	🟥	🟥
if (y < z)	🟥	🟥	🟥	🟥	🟥	🟥
if (x < y)	🟥	🟥	🟥		🟥	🟥
m = y;			🟥			
else if (x<z)	🟥				🟥	🟥
m = y; // bug	🟥					🟥
else			🟥	🟥		
if (x > y)			🟥	🟥		
m = y;			🟥			
else if (x>z)				🟥		
m = x;						
return m;	🟥	🟥	🟥	🟥	🟥	🟥
	Pass	**Pass**	**Pass**	**Pass**	**Pass**	**Fail**

Insight: Print-Statement Debugging

- If you do not execute X but you do observe the bug, X cannot be related to that bug
- If Y is primarily executed when you observe the bug, it is more likely to be implicated that Y is more related to the bug than X
- **Suspiciousness Ranking** from **Tarantula**

$$Suspicious(s) = \frac{fail(s) / totalfail}{fail(s) / totalfail + pass(s) / totalpass}$$

Fault Localization Ranking

Statement	**3,3,5**	**1,2,3**	**3,2,1**	**3,2,1**	**5,5,5**	**2,1,3**	**susp(s)**
int m;	🟥	🟥	🟥	🟥	🟥	🟥	**0.5**
m = z;	🟥	🟥	🟥	🟥	🟥	🟥	**0.5**
if (y < z)	🟥	🟥	🟥	🟥	🟥	🟥	**0.5**
if (x < y)	🟥	🟥	🟥		🟥	🟥	**0.63**
m = y;			🟥				**0**
else if (x<z)	🟥				🟥	🟥	**0.71**
m = y; // bug	🟥					🟥	**0.83**
else			🟥	🟥			**0**
if (x > y)			🟥	🟥			**0**
m = y;			🟥				**0**
else if (x>z)				🟥			**0**
m = x;							**0**
return m;	🟥	🟥	🟥	🟥	🟥	🟥	**0.5**
	Pass	**Pass**	**Pass**	**Pass**	**Pass**	**Fail**	

Popular SBFL Formula

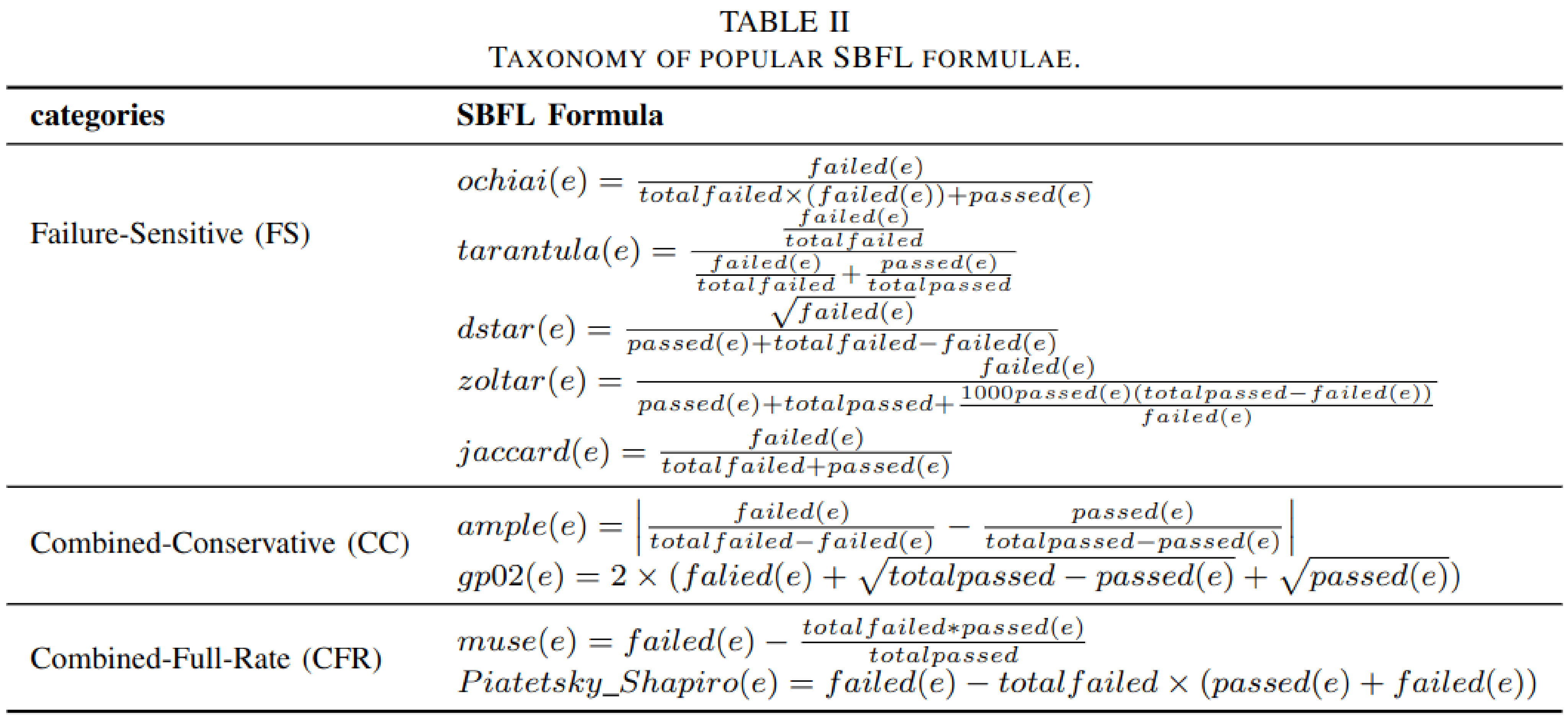
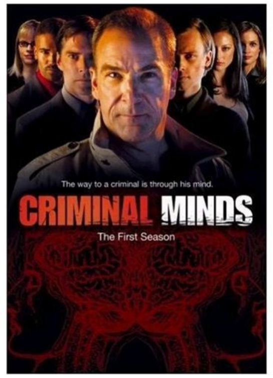


TABLE II
TAXONOMY OF POPULAR SBFL FORMULAE.

categories	SBFL Formula
Failure-Sensitive (FS)	$ochiai(e) = \frac{failed(e)}{totalfailed \times (failed(e)) + passed(e)}$ $tarantula(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{failed(e)}{totalfailed} + \frac{passed(e)}{totalpassed}}$ $dstar(e) = \frac{\sqrt{failed(e)}}{passed(e) + totalfailed - failed(e)}$ $zoltar(e) = \frac{failed(e)}{passed(e) + totalpassed + \frac{1000passed(e)(totalpassed - failed(e))}{failed(e)}}$ $jaccard(e) = \frac{failed(e)}{totalfailed + passed(e)}$
Combined-Conservative (CC)	$ample(e) = \left\| \frac{failed(e)}{totalfailed - failed(e)} - \frac{passed(e)}{totalpassed - passed(e)} \right\|$ $gp02(e) = 2 \times (falied(e) + \sqrt{totalpassed - passed(e)} + \sqrt{passed(e)})$
Combined-Full-Rate (CFR)	$muse(e) = failed(e) - \frac{totalfailed * passed(e)}{totalpassed}$ $Piatetsky\_Shapiro(e) = failed(e) - totalfailed \times (passed(e) + failed(e))$

Zhang, Yueke, Kevin Leach, and Yu Huang. "Leveraging Evidence Theory to Improve Fault Localization: An Exploratory Study." In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1-12. IEEE, 2023.

Profiling



- A **profiler** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.
- A **flat profile** computes the average call times for functions but does not break times down based on context
- A **call-graph profile** computes call times for functions and also the call-chains involved

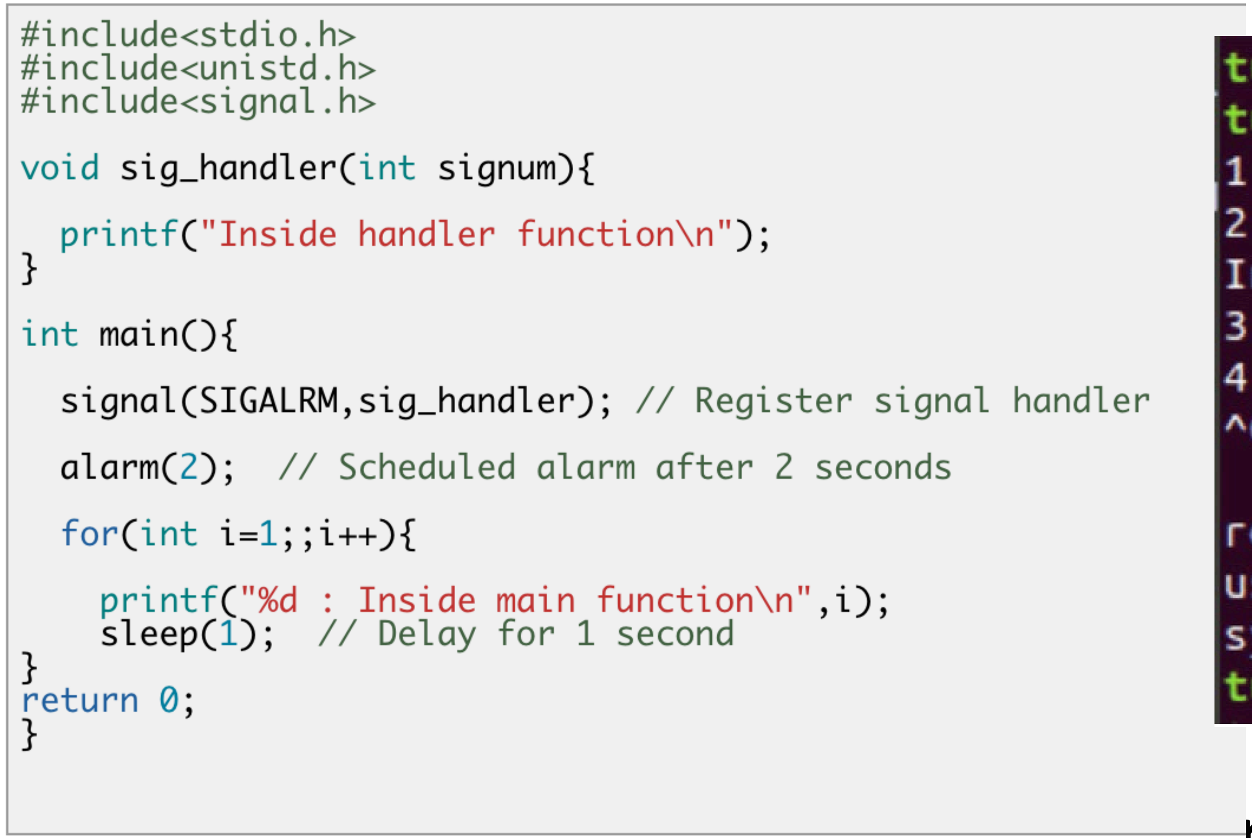
Event-Based Profiling

- Interpreted languages provide special hooks for profiling
  - Java: JVM-Profile Interface, JVM API
  - Python: sys.set_profile() module
  - Ruby: profile.rb, etc.
- You register a function that will get called whenever the target program calls a method, loads a class, allocates an object, etc.
  - cf. "signal handler"

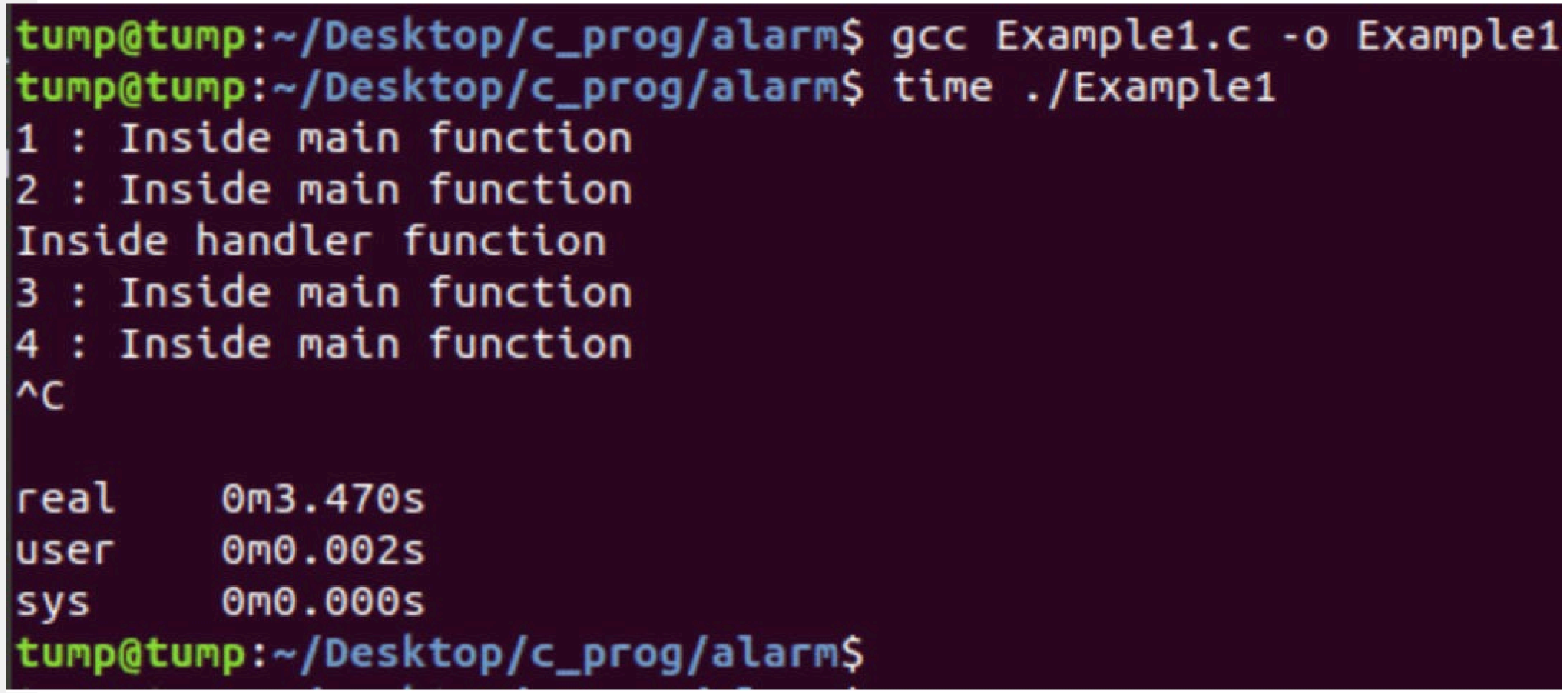
Statistical Profiling

- You can arrange for the operating system to send you a **signal** (just like before) every X seconds (see **alarm(2)**)
- In the **signal handler** you determine the value of the target program counter
  - And append it to a growing list file
  - This is **sampling**
- Later, you use debug information from the compiler to map the PC values to procedure names
  - Sum up to get amount of time in each procedure

Statistical Profiling



```c
#include<stdio.h>
#include<unistd.h>
#include<signal.h>

void sig_handler(int signum){

  printf("Inside handler function\n");
}

int main(){

  signal(SIGALRM,sig_handler); // Register signal handler

  alarm(2);  // Scheduled alarm after 2 seconds

  for(int i=1;;i++){

    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
}
return 0;
}
```



```
tump@tump:~/Desktop/c_prog/alarm$ gcc Example1.c -o Example1
tump@tump:~/Desktop/c_prog/alarm$ time ./Example1
1 : Inside main function
2 : Inside main function
Inside handler function
3 : Inside main function
4 : Inside main function
^C

real	0m3.470s
user	0m0.002s
sys	0m0.000s
tump@tump:~/Desktop/c_prog/alarm$
```

mation from the compiler to map the PC values to procedure names

- Sum up to get amount of time in each procedure

Sampling Analysis

- Advantages
  - Simple and cheap – the **instrumentation** is unlikely to disturb the program
  - No big slowdown
- Disadvantages

Sampling Analysis

- Advantages
  - Simple and cheap – the **instrumentation** is unlikely to disturb the program
  - No big slowdown
- Disadvantages
  - Can completely miss periodic behavior (e.g., you sample every *k* seconds but do a network send at times 0.5 + n*k* seconds)
  - High error rate
- Read the **gprof** paper

Sampling Analysis

- Accuracy depends on sampling rate
- Higher sampling rate incurs higher overhead

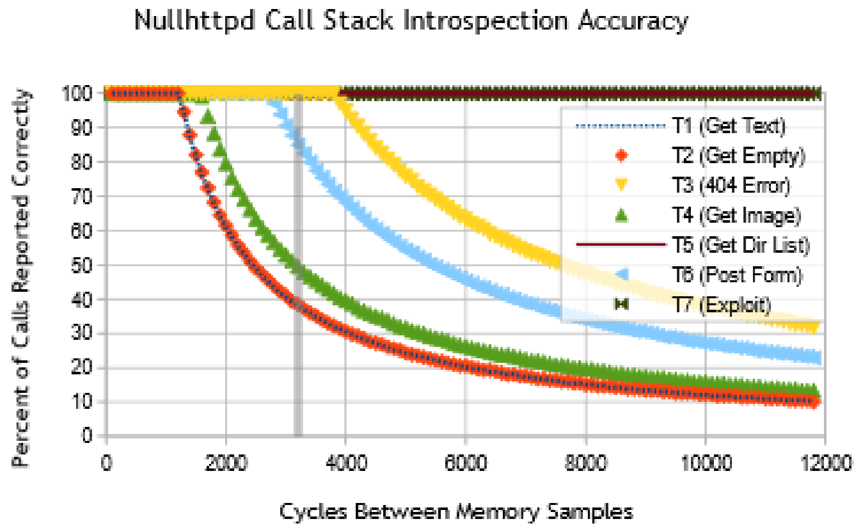


Fig. 5. Call stack introspection accuracy for Nullhttpd as a function of the number of machine cycles between memory samples. The reference line at 3200 corresponds to current hardware. On all tests sampling every 1200 cycles yields perfect accuracy.

[Leach et al. Towards Transparent Introspection. SANER 2016]

Real-World Tool Utility

- Human study of 34 graduate students
- Given Tarantula (as a friendly plugin for Eclipse) and asked to complete two debugging tasks
  - Tetris: square block rotation bug
  - NanoXML: parsing library exception
- Hypotheses:
  - Tools will help us debug faster
  - Tools help more on harder problems

[Parnin and Orso. Are Automated Debugging Techniques Actually Helping Programmers? ISSTA '11.]

Results

- Experts Are Faster When Using Tools
  - Over all participants, tools did not help
  - Top-third of participants went from 14:28 to 8:51 with tool support (for Tetris, $p < 0.05$)
- Tools Did Not Help With Harder Tasks
- Changes In Rank Did Not Matter
  - 7 → 35 in Tetris, 83 → 16 in NanoXML
  - Why is this so crucial here?

Explanations

- "Based on this data, we have determined that programmers do not visit each statement in a linear fashion."
- "If the *faulty nature* of a statement were apparent to the developers by *just looking at it*, tool usage *should stop* as soon as they get to *that statement* in the list."
  - "participants, on average, spent another ten minutes using the tool after they first examined the faulty statement. That is, participants spent (or wasted) on average 61% of their time continuing to inspect statements with the tool after they had already encountered the fault."

Implications

- You are a Software Engineering manager
- Making a process decision: do we purchase, train on, and deploy Tarantula?
- Tarantula claims: this tool will correctly rank buggy statements near the top of the list
  - This is almost a red herring!
  - You must examine the "end-to-end" performance
- So fault localization tools are worthless?

Nuanced Example

- Suppose you have three devs: A, B and C
  - Expert, Medium, Novice
- Tarantula makes A, the expert, 39% faster
  - But makes everything 13% slower (training, overhead, whatever)
- If everything is equal, net gain = 0 (as in study)
- But suppose A is 25x faster than C (*productivity* later)
  - A=25, B=13, C=1 → in this world your team, overall, is 8.7% faster with Tarantula

Questions?

- HW4
  - Please read the homework website first:

    Recommended Work Flow
  - Due this Sunday
  - Can use GP