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 (triage)
- 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?

A lot of code. A lot of defects.

Microsoft: 70 percent of all security bugs are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

By Catalin Cimpanu for Zero Day | February 11, 2019 -- 15:48 GMT (07:48 PST) | Topic: Security

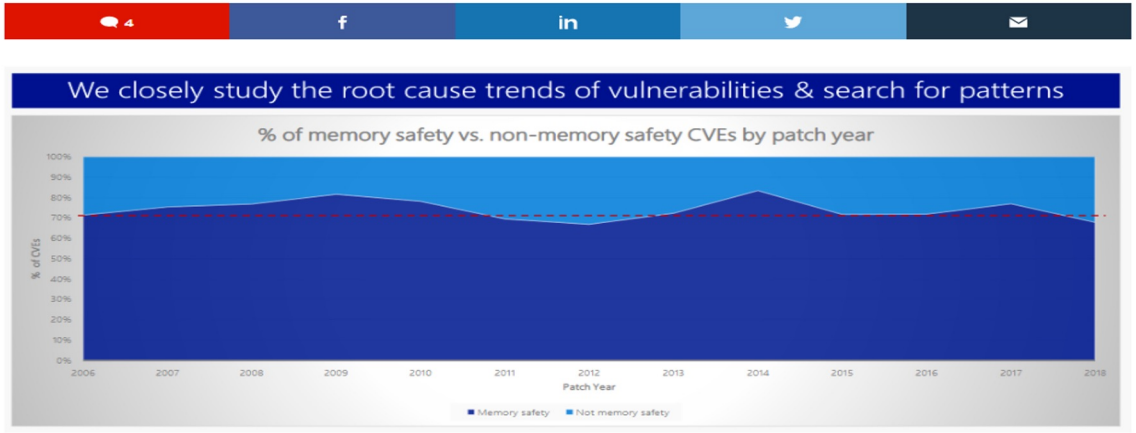


Image: Matt Miller

https://www.cve.org/

Around 70 percent of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues; a Microsoft engineer revealed last week at a security conference.

https://edu.chainguard.dev/software-security/cves/cve-intro/

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

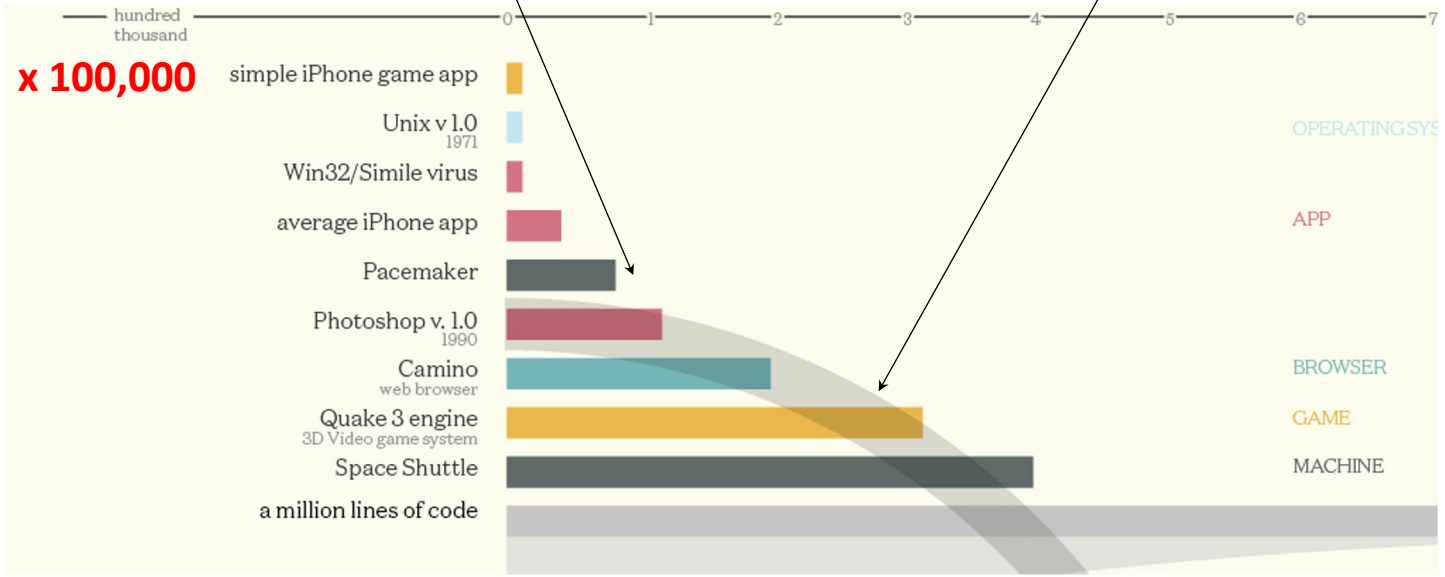
- "Space is big. Really big. You just won't believe how vastly, hugely, mind-bogglingly big it is. I mean, you may think it's a long way down the road to the chemist, but that's just peanuts to space." – Douglas Adams
- Who cares?
  - Techniques developed based on smaller code bases simply do *not* apply or scale to larger code bases
    - Techniques from the 1980s or your habits from classes

- How many lines of code? Guess??
  - iPhone app
  - Facebook
  - Chrome/Firefox
  - Microsoft Office
  - Car Software
  - Space Shuttle

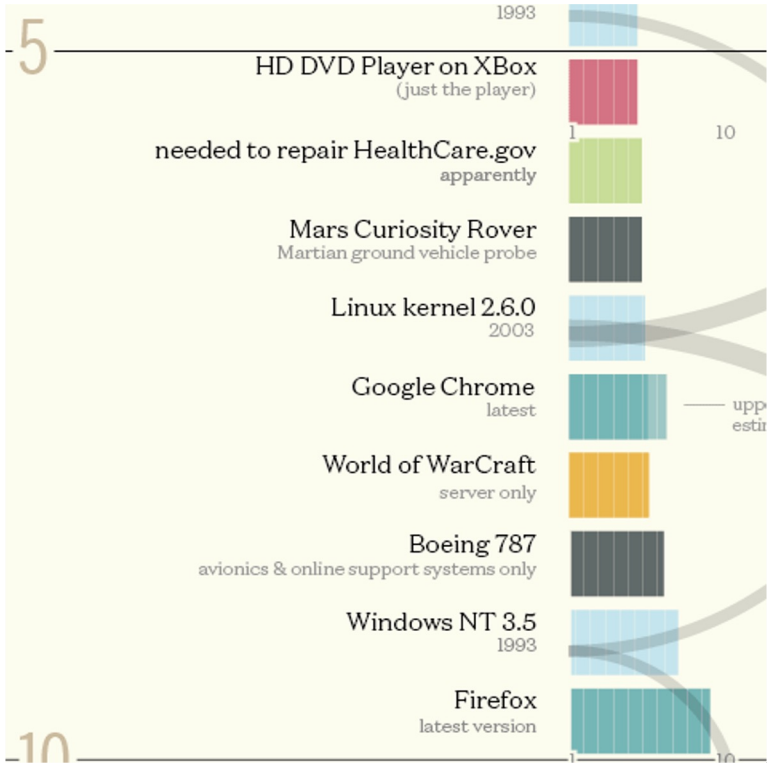
Example Programs: < 1MLOC

- libpng: 85,000

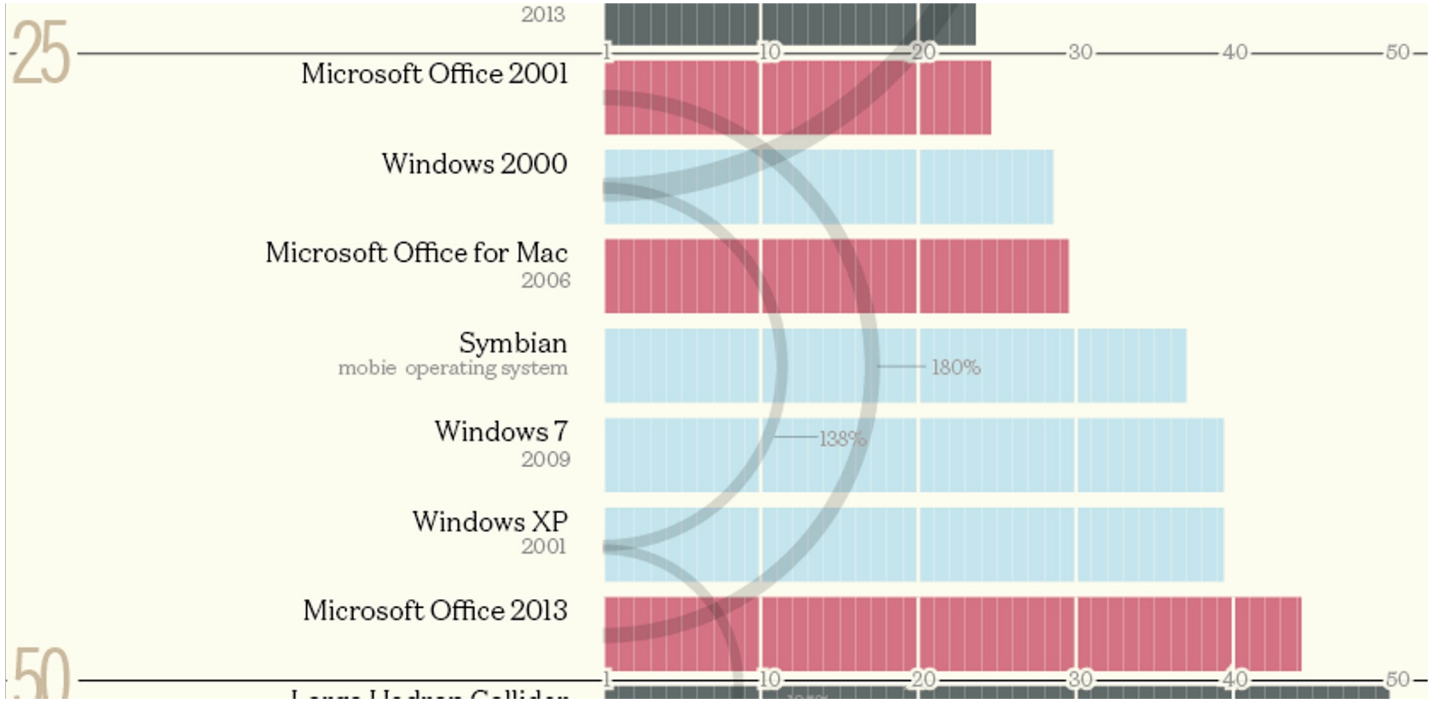
jfreechart: 300,000



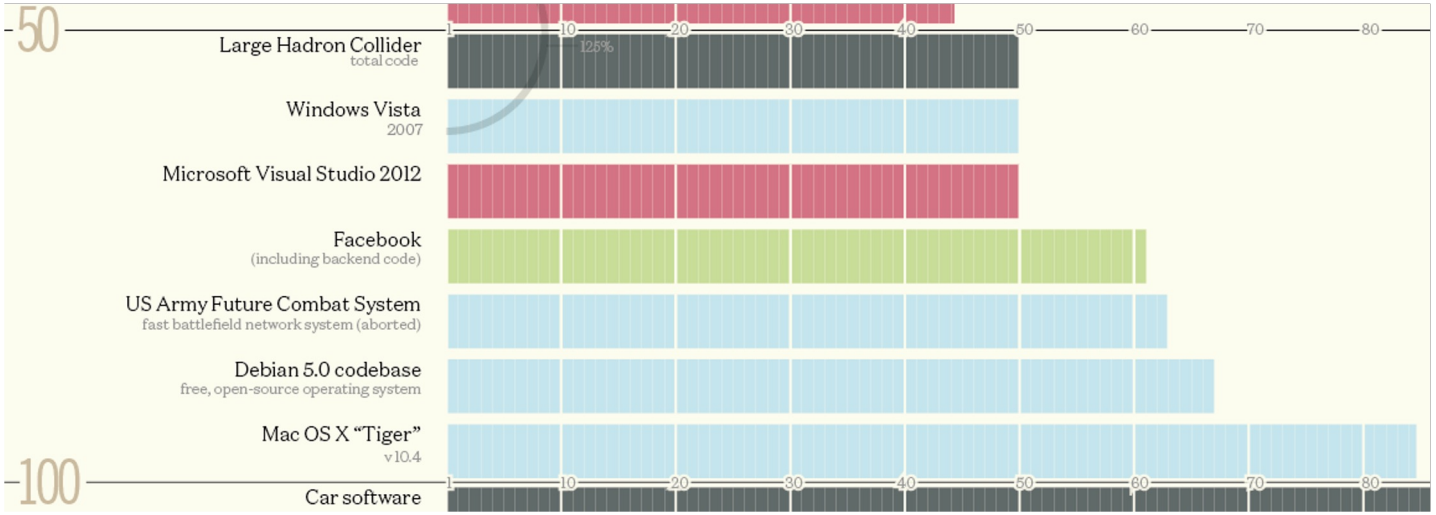
Example Programs: 1-10 MLOC



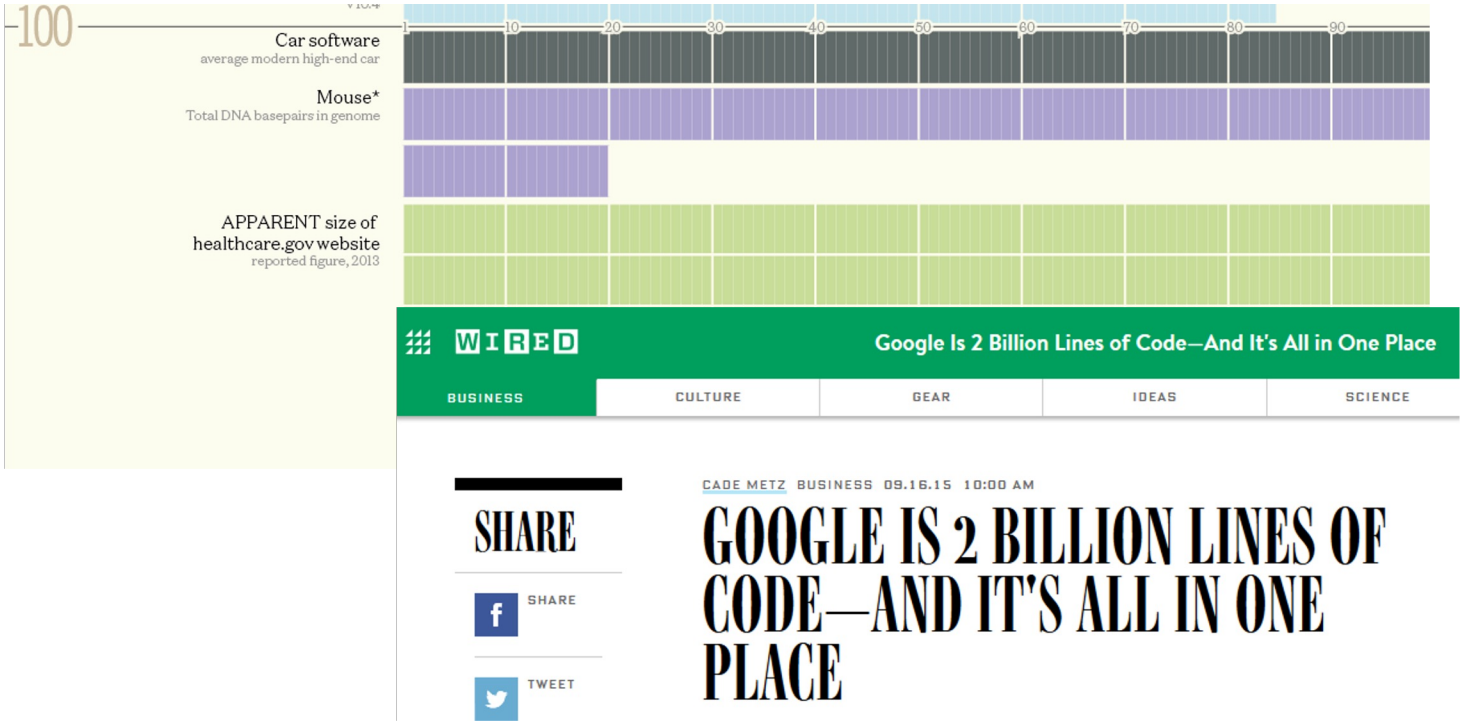
Example Programs: 25 – 50 MLOC



Example Programs: 50 – 100 MLOC



Example Programs: 0.1 – 2.0BLOC



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!
- 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 …

Program debugging

- Program debugging is the process of finding and fixing errors or bugs in a software program.
- Debugging can help improve the quality, performance, and reliability of the software.
- Debugging can be done manually or with the help of tools and techniques.
- Debugging is a dynamic analysis technique that involves examining and modifying the state of a program during its execution and finding and fixing errors or bugs.

Program debugging (Cont'd)

- Debugging can be done manually, using tools such as print statements, breakpoints, or watchpoints, or automatically, using tools such as debuggers, profilers, or monitors.
- Debugging can also be done at different levels of abstraction, such as source code, assembly code, or machine code.
- Debugging can help developers understand the logic and flow of their program, identify the causes and effects of errors, and validate the correctness and performance of their program.

Steps of debugging

- Identifying the problem: This involves determining what the expected behavior of the program is and what the actual behavior is. This can be done by running test cases, checking error messages, or reproducing the problem.
- Locating the source of the problem: This involves finding where in the code the problem occurs and what causes it. This can be done by using breakpoints, tracing, logging, or inspecting variables.
- Correcting the problem: This involves modifying the code to eliminate the error and ensure it does not happen again. This can be done by editing, refactoring, or testing the code.
- Validating the solution: This involves checking if the problem is solved and if there are any side effects or new errors. This can be done by running test cases, reviewing the code, or monitoring the performance.

Some popular debugging tools

- Visual Studio Debugger: A tool for debugging C#, C++, Visual Basic, and other languages in Visual Studio.
- Chrome Debugger: A tool for debugging JavaScript, HTML, CSS, and other web technologies in Chrome DevTools. It allows you to pause execution, inspect elements, modify values, and more.
- ExifTool: A tool for extracting metadata from various types of files, such as images, documents, or archives.
- PE Studio: A tool for statically examining many aspects of a suspicious Windows executable file, such as imported and exported function names, strings, hashes, packers, and suspicious API calls.

Size of a program & its debugging

- The size of a program can affect the number and quality of test cases that are needed to cover the program and reveal the bugs.
- Larger programs may require more test cases to achieve a high code coverage and expose faulty behaviors. However, generating and executing more test cases can also be more time-consuming and resource-intensive.
- Moreover, the quality of the test cases can also influence the debugging process, as test cases that are more effective in distinguishing between correct and incorrect program behaviors can help narrow down the search space for the bugs.

Size of a program & its debugging (Cont'd)

- The size of a program can affect the complexity and diversity of bugs that may occur in the program.
- Larger programs may have more complex and diverse bugs that are harder to locate and fix. For example, larger programs may have more dependencies, interactions, and concurrency issues that can cause bugs that are difficult to reproduce or isolate.
- Moreover, larger programs may have more types of bugs, such as syntactic, semantic, logical, or design bugs, that may require different debugging techniques or tools.

Size of a program & its debugging (Cont'd)

- The size of a program can affect the performance and accuracy of debugging techniques or tools that are used to locate the bugs.
- Larger programs may pose more challenges for debugging techniques or tools, as they may have more statements, variables, branches, loops, or functions that need to be analyzed and ranked according to their suspiciousness of being faulty.
- Moreover, larger programs may have more noise or irrelevant information that can affect the accuracy of debugging techniques or tools. For example, some debugging techniques or tools may rely on statistical models, machine learning algorithms, or information retrieval methods that can be affected by the size of the program and the data.

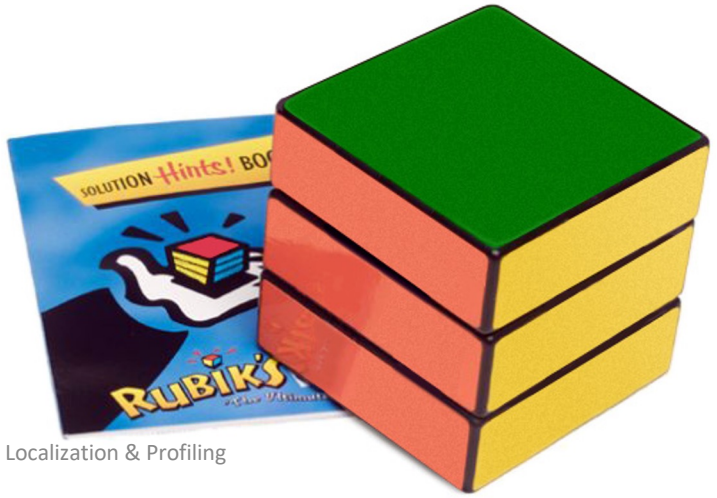
What is a Debugger?

- “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



Signals in Debugging

- A signal in debugging is a way of communicating between a running program and a debugger or another process.
- A signal can indicate that the program has encountered an error, an exception, an interruption, or a termination request.
- A signal can also be used to control the execution of the program, such as pausing, resuming, or stopping it.

Signals & Debugging (Cont'd)

- Signals are used in debugging to control the execution of a program and to inspect its state.
- Signals are messages that are sent to a process by the operating system, another process, or itself.
- Some signals indicate errors, such as segmentation faults or illegal instructions, while others indicate events, such as interrupts or alarms.

How does a debugger use signals?

- A debugger can use signals to stop, resume, or modify the behavior of a program. For example, a debugger can send a SIGINT signal to interrupt a program, a SIGCONT signal to continue a program, or a SIGTRAP signal to set a breakpoint in a program.
- A debugger can also register a signal handler for a program, which is a function that is executed when a signal is received.
- A signal handler can perform some actions, such as printing the values of variables, modifying the memory or registers, or terminating the program.

Signals

- A **signal** is a notification sent to a process about an event:
  - 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 (**SIGPIPE**)

- You can install a **signal handler** – a procedure that will be executed when the signal occurs.

Signal Example

- What does this program print?


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

- 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.

```
#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);
}
```

Breakpoints

- A breakpoint is a point in a software program where the execution of the program is paused or stopped for debugging purposes.
- A breakpoint can help examine the state of the program, such as the values of variables, the call stack, or the memory usage, at a specific moment.
- A breakpoint can also help control the flow of the program, such as stepping through the code, resuming the execution, or terminating the program.

Breakpoints (Cont'd)

- A debugger can use special registers, such as the program counter (PC) and the hardware breakpoint (HBP), to implement breakpoints.
- The PC register holds the address of the next instruction to be executed, and the HBP register holds the address of a breakpoint location.
- If the PC value equals the HBP register value, it means that the program has reached the breakpoint, and the debugger can signal an exception and invoke the signal handler.

Advanced Breakpoints

- Optimization: **hardware breakpoints**
  - Special registers (a few of them): if PC value = HBP register value, signal an exception
  - Faster than software, works on ROMs, only a 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

- Single-stepping is a debugging technique that allows a programmer or tester to execute a program one instruction at a time and inspect its state after each step.
- Single-stepping can help to find and fix errors, understand complex behaviors, or test specific scenarios.
- Single-stepping can be done using tools such as debuggers, which provide features such as step-over and step-into to control the execution flow of the program.

Single-Stepping (Cont'd)

- 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

- A watchpoint is a type of debugging tool that allows monitoring of the value or the memory location of a variable or an expression in a software program.
- A watchpoint can trigger a breakpoint or an action when the variable or the expression is accessed or modified in a certain way.
- Watchpoints can help find and fix errors or bugs in codes.

https://interrupt.memfault.com/blog/cortex-m-watchpoints
https://www.techopedia.com/definition/28718/watchpoint-sap

Watchpoints (Cont'd)

- 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?

Watchpoint Implementation

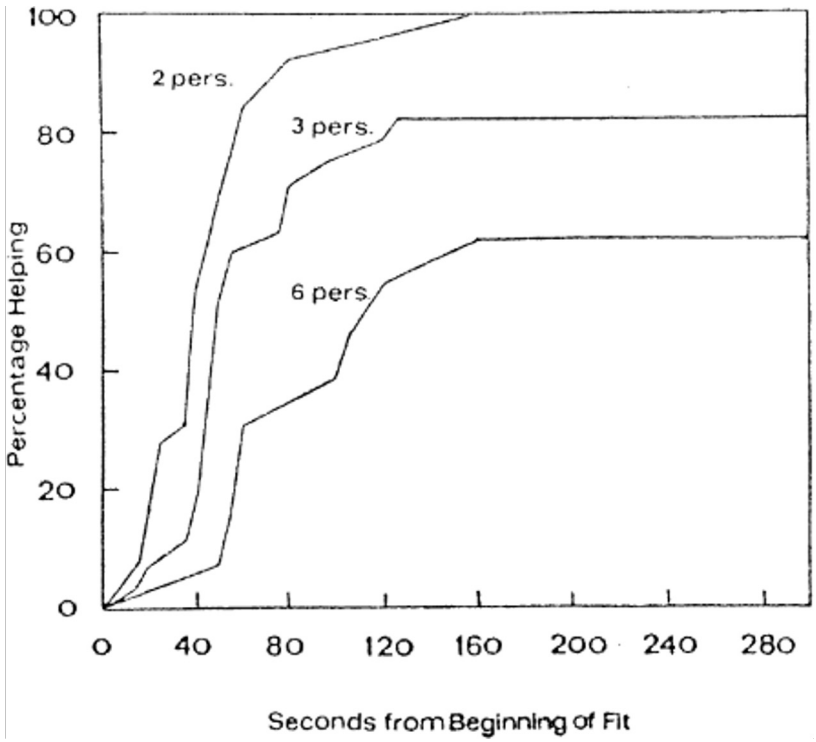
- Software Watchpoints
  - Put a breakpoint at *every instruction* (ouch!)
  - Check the current value of L against a stored value
  - If different, give interactive debugging prompt
  - If not, set next breakpoint and continue (single-step)
- Hardware Watchpoints
  - Special register holds L: if the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

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



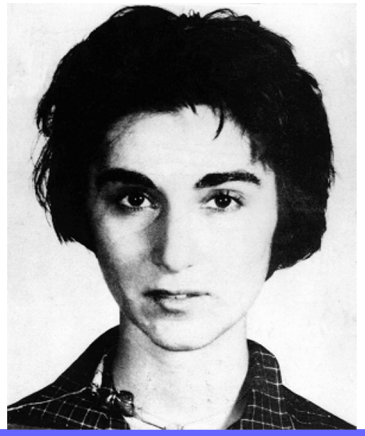
Bystander Effect

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

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?



Fault Localization

- Fault localization is the process of identifying the locations of faults in a program that cause failures or errors.
- Fault localization is an important and challenging task in software debugging, as it can help developers find and fix bugs more efficiently and effectively.

Fault Localization (Cont'd)

- **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?"
- The answer is not unique: there are often many places to fix a bug
  - Example: check for null at caller or callee.
- Debugging includes fault localization
- The answer may take the form of a list (e.g., of lines) ranked by suspiciousness

Program Spectrum-Based Method

- A program spectrum represents the execution behavior of a program or a component.
- It consists of a set of entities, such as statements, branches, or functions, and a set of spectra, which are vectors of binary values that indicate whether each entity was executed or not during a test.

Advanced Fault Localization Methods

Slice-Based Techniques: Program slicing is a technique to abstract a program into a reduced form by deleting irrelevant parts such that the resulting slice will still behave the same as the original program concerning certain specifications.

Statistics-Based Techniques: A statistical debugging technique that can isolate bugs in programs with instrumented predicates at certain points.

Program State-Based Techniques: A program state consists of variables and their values at a particular point during program execution, which can be a good indicator for locating program bugs.

https://www.researchgate.net/publication/291951202_A_Survey_on_Software_Fault_Localization

Advanced Fault Localization Methods (Cont'd)

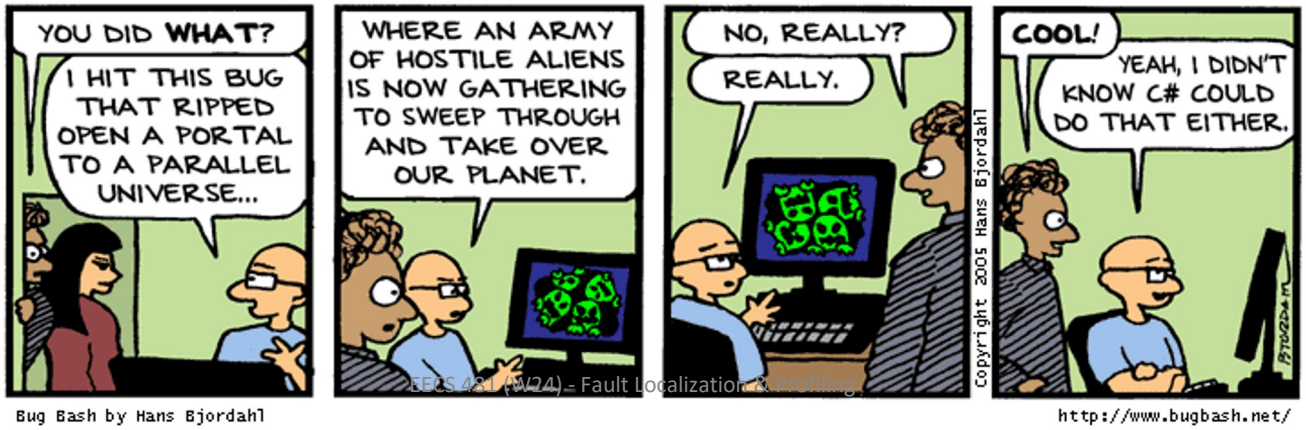
Machine Learning-Based Techniques: In the context of fault localization, the problem at hand can be identified as trying to learn or deduce the location of a fault based on input data such as statement coverage and the execution result (success or failure) of each test case.

Data Mining-Based Techniques: The software fault localization problem can be abstracted to a data mining problem – for example, we wish to identify the pattern of statement execution that leads to failure.

Model-Based Techniques: It is assumed that a correct model of each program being diagnosed is available. That is, these models can serve as the oracles of the corresponding programs. Differences between the behaviors of a model and the actual observed behaviors of the program are used to help find bugs in the program.

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



Find The Bug (Towers of Hanoi)

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

https://web.eecs.umich.edu/~movaghar/A_human_study_of_fault_localization_accuracy.pdf

[Z. Fry et al.: A Human Study of Fault Localization Accuracy. International Conference on Software Maintenance (ICSM) 2010]

```
1  /*****************************************
2    Performs the initial call to moveTower
3    to solve the puzzle.  Moves the disks
4    from tower 1 to tower 3 using tower 2.
5  *****************************************/
6  public void solve () {
7    moveTower (totalDisks, 1, 3, 2);
8  }
9
10 /*****************************************
11   Moves the specified number of disks
12   from one tower to another by moving a
13   subtower of n-1 disks out of the way,
14   moving one disk, then moving the
15   subtower back. Base case of 1 disk.
16 *****************************************/
17 private void moveTower (int numDisks,
18                         int start, int end, int temp) {
19 if (numDisks == 1)
20     moveTower(numDisks-1, temp, end, start);
21   else {
22     moveTower (numDisks-1, start, temp, end);
23     moveOneDisk (start, end);
24     moveTower (numDisks-1, temp, end, start);
25   }
26 }
27 /*****************************************
28   Prints instructions to move one disk
29   from the specified start tower to the
30   specified end tower.
31 *****************************************/
32 private void moveOneDisk (int start, int end) {
33   System.out.println ("Move one disk from "
34       + start + " to " + end);
35 }
```

Find The Bug 2

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

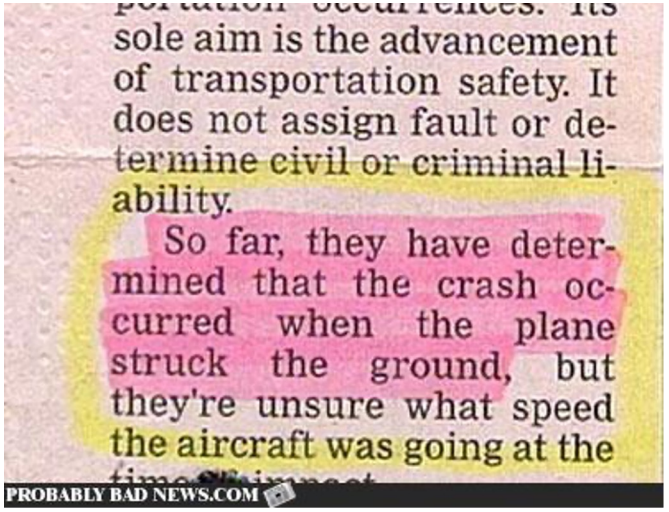
```
1  /** Move a single disk from src to dest. */
2  public static void hanoi1(int src, int dest){
3    System.out.println(src + " => " + dest);
4  }
5  /** Move two disks from src to dest,
6      making use of a spare peg. */
7  public static void hanoi2(int src,
8                            int dest, int spare) {
9    hanoi1(src, dest);
10   System.out.println(src + " => " + dest);
11   hanoi1(spare, dest);
12 }
13 /** Move three disks from src to dest,
14     making use of a spare peg. */
15 public static void hanoi3(int src,
16                            int dest, int spare) {
17   hanoi2(src, spare, dest);
18   System.out.println(src + " => " + dest);
19   hanoi2(spare, dest, src);
20 }
```

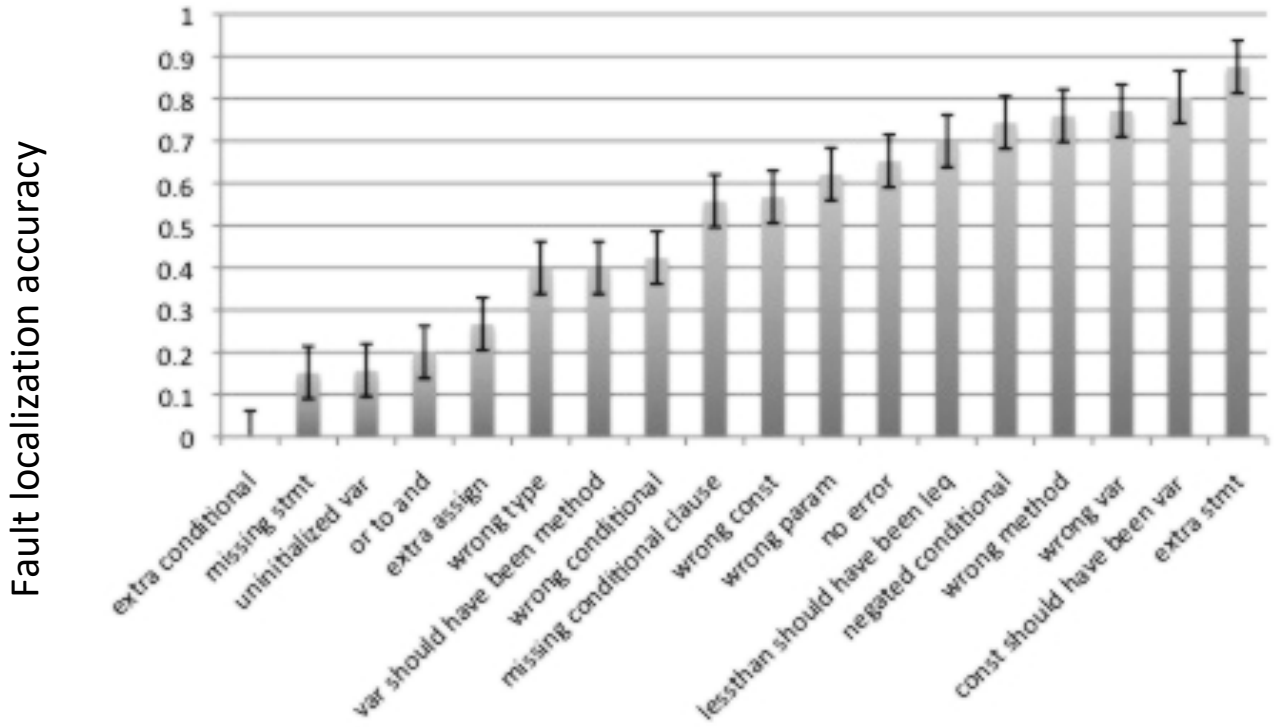
Human Study

- 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)

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





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!

https://people.cs.umass.edu/~rjust/publ/fault_localization_effectiveness_icse_2017.pdf
https://web.eecs.umich.edu/~movaghar/Evaluation Fault Localization 2005.pdf

Tarantula

- Tarantula is a fault localization technique that uses code coverage information and test results to rank the statements in a program by their likelihood of containing a fault.
- Tarantula assigns a suspiciousness score to each statement based on the ratio of passing and failing test cases that execute it. The higher the suspiciousness score, the more likely the statement is to be faulty.
- Tarantula also uses a color scheme to visualize the suspiciousness of statements, ranging from red (most suspicious) to green (least suspicious).

Tarantula

- Tarantula is one of the spectrum-based fault localization techniques, which uses statistical analysis of program spectra to identify fault locations.
- Tarantula was proposed by Jones and Harrold in 2005 and has been compared with other fault localization techniques in several empirical studies.
- Tarantula has been shown to be effective and efficient in locating faults in various types of programs, such as C, Java, and Solidity.

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 than Z which is primarily executed when you do not observe the bug
- **Suspiciousness Ranking**

susp(s) = (fail(s)/total_fail) / (fail(s)/total_fail + pass(s)/total_pass)

[Jones and Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. ASE 2005.]

Fault Localization Ranking

	Test Cases						suspiciousness	rank
mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		
1: read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	0.5	7
2: m = z;	●	●	●	●	●	●	0.5	7
3: if (y<z)	●	●	●	●	●	●	0.5	7
4: if (x<y)	●	●			●	●	0.63	3
5: m = y;		●					0.0	13
6: else if (x<z)	●				●	●	0.71	2
7: m = y; // *** bug ***	●					●	0.83	1
8: else			●	●			0.0	13
9: if (x>y)			●	●			0.0	13
10: m = y;			●				0.0	13
11: else if (x>z)				●			0.0	13
12: m = x;							0.0	13
13: print("Middle number is:",m);	●	●	●	●	●	●	0.5	7
} Pass/Fail Status	P	P	P	P	P	F		

susp(s) = (fail(s)/total_fail) / (fail(s)/total_fail + pass(s)/total_pass)

Software Profiling

- Software profiling is the process of measuring and analyzing the performance of a software program.
- Software profiling can help you identify and fix errors or bugs that affect the speed, memory usage, CPU load, or resource consumption of your program.
- Software profiling can also help you optimize your code and improve the quality and reliability of your software.

Software Profiling (Cont'd)

Profiling is the runtime analysis of metrics such as execution speed and memory usage, which is typically aimed at program optimization. However, it can also be leveraged for debugging activities, such as the following:

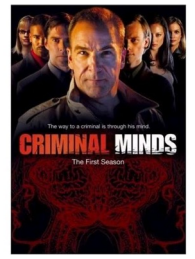
- Detecting unexpected execution frequencies of different functions;
- Identifying memory leaks or code that performs unexpectedly poorly;
- Examining the side effects of lazy evaluation.

Tools that use profiling for program debugging include GNU's gprof and the Eclipse plugin TPTP.

https://sourceware.org/binutils/docs/gprof/index.html

https://www.eclipse.org/articles/Article-TPTP-Profiling-Tool/tptpProfilingArticle.html

Profiler



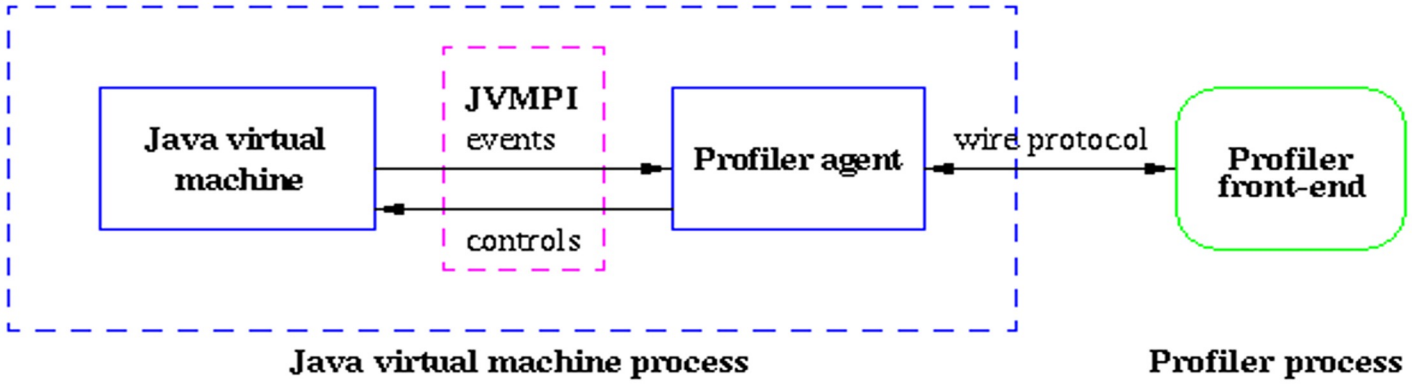
- A **profiler** is a performance analysis tool that measures the frequency and duration of function calls as a program runs.
  - A **flat profile** computes the average call times for functions but does not break times down based on context
  - A **call-graph profile** computes call times for functions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

JVM Profiling Interface

- VM notifies profiler agent of various **events** (heap allocation, thread start, method invocation, etc.)
- Profiler agent issues control commands to the JVM and communicates with a GUI



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

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: if a value is n times the sampling period, the expected error in it is sqrt(n) sampling periods
- Read the **gprof** paper

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 with harder problems

[Parnin and Orso. Are Automated Debugging Techniques Actually Helping Programmers? ISSTA '11.]
https://web.eecs.umich.edu/~movaghar/Automated Debugging helpful 2011.pdf

Results

- Experts Are Faster When Using Tools
  - Over all participants, tools did not help
  - Top-third of participants went from 14m:28s to 8:51 with tool support (for Tetris, $p < 0.05$)
- Tools Did Not Help With Harder Tasks
- Changes In Rank Did Not Matter
  - For the faulty statement, (Rank) 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?

- **HW 3 is due today!**
- **HW 4 is due next Wednesday!**