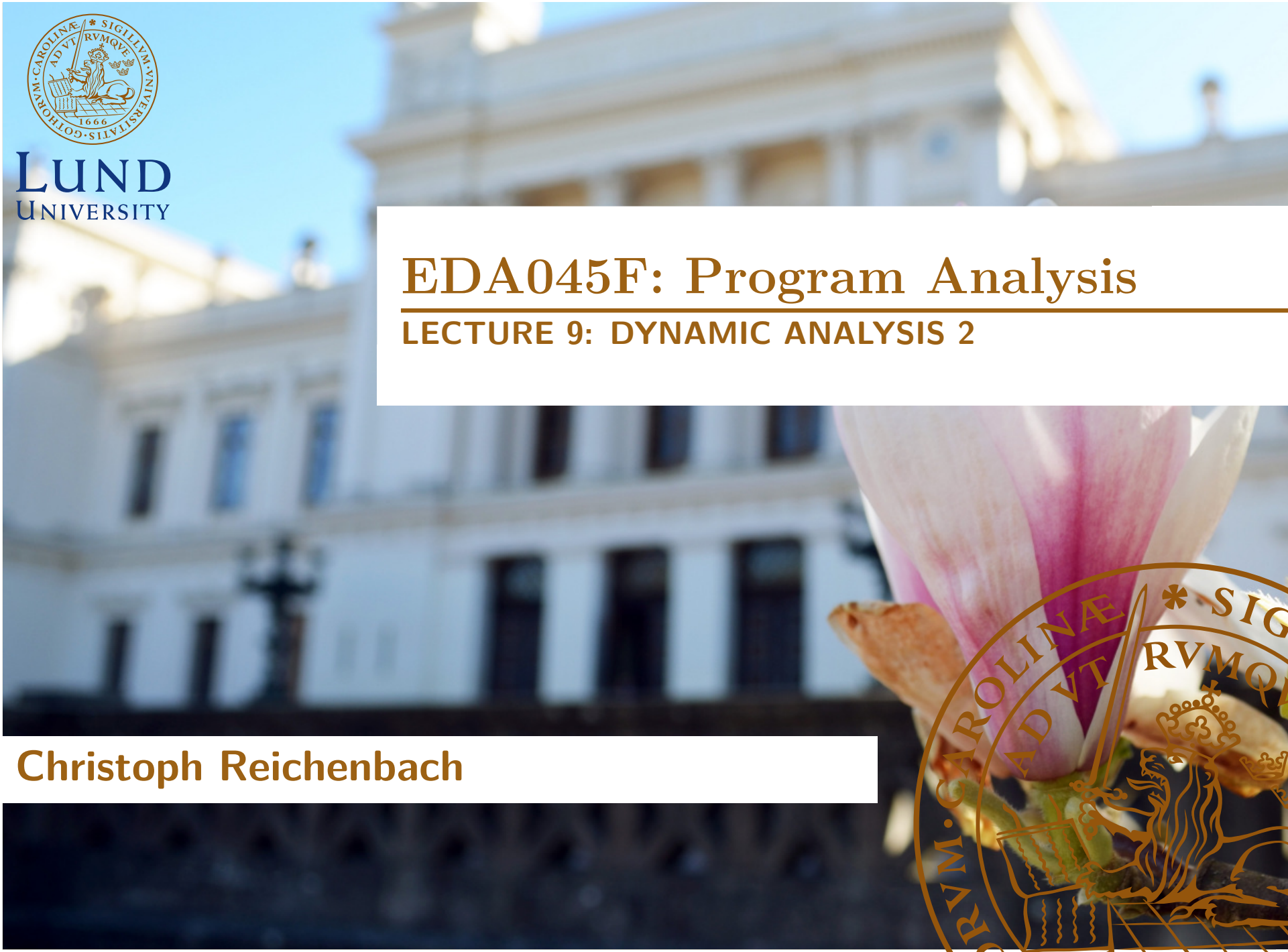


LUND UNIVERSITY

EDA045F: Program Analysis

LECTURE 9: DYNAMIC ANALYSIS 2

Christoph Reichenbach

In the last lecture...

- Dynamic analysis examines behaviour of program during one run
- Can analyse:
  - Output
  - Correctness
  - Safety
  - Security
  - **Performance**
    - Wallclock execution time
    - Software performance counters
    - Hardware performance counters

Automatic Performance Measurement

- Profiler:
  - Interrupts program during execution
  - Examines call stack
- Simulator:
  - Simulates CPU/Memory in software
  - Tries to replicate inner workings of machine
  - Often also an *Emulator* (= replicate observable functionality)
- Operating System:
  - Counts important system events (network accesses etc.)
- CPU:
  - Hardware performance counters count interesting events

Profiler

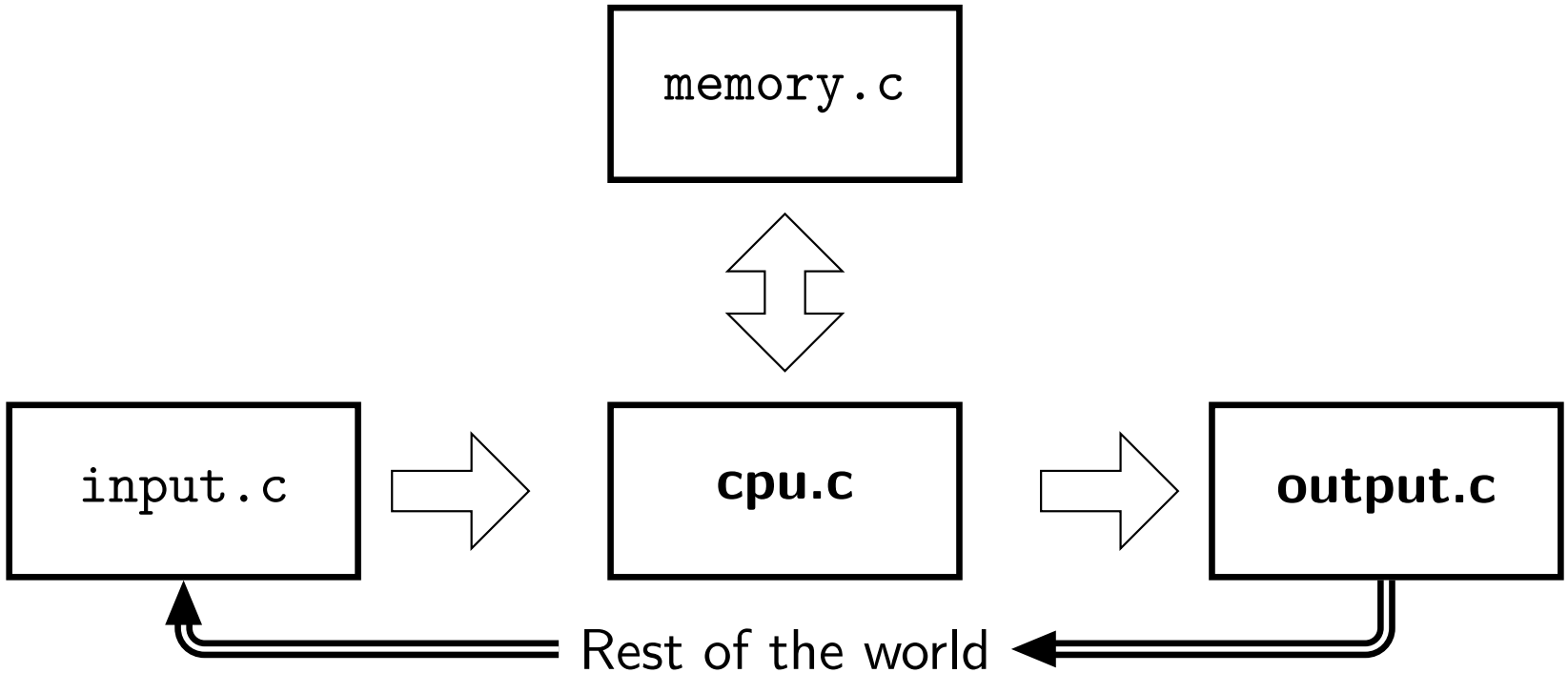
- Measures: which functions are we spending our time in?
- Approach:
  - Build stack maps
  - Execute program, interrupt regularly
  - During interrupt:
    - Examine stack
- Infer functions from stack contents

Execution Stack

`return` (alt-1)
`$fp` (alt-1)
...
...
`return` (alt-2)
`$fp` (alt-2)
...

Can be inaccurate: misses short function calls

Simulator



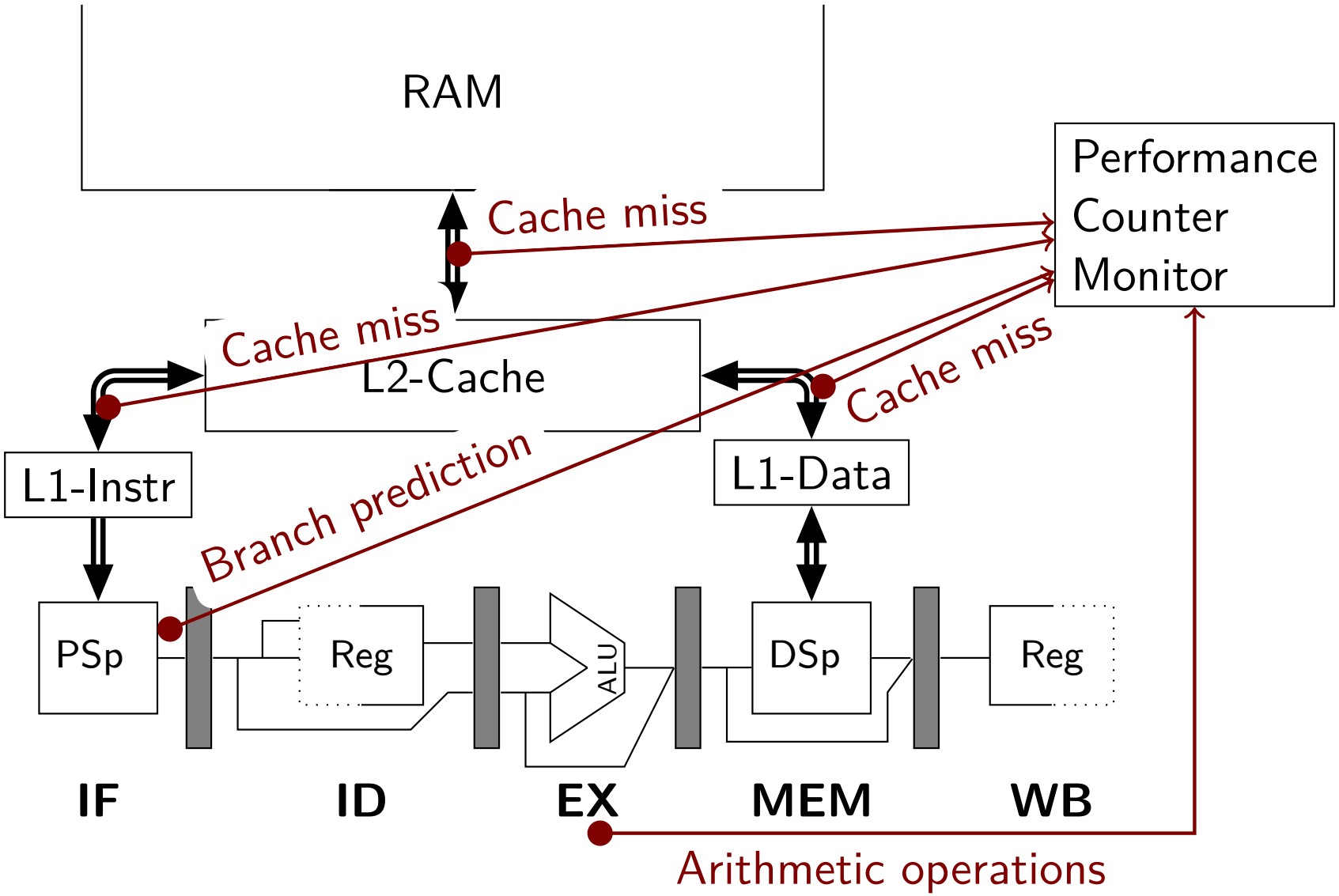
- Software simulates hardware components
- Can count events of interest (memory accesses etc.)

Modern CPUs are very complex: Simulators tend to be inaccurate

Software Performance Counters

- Complex software may use high-level properties such as:
  - How much time do we spend waiting for the harddisk?
  - How often was our program suspended by the operating system in order to let another program run?
  - How much data did we receive through the network?
- Operating systems collect many of these statistics

Hardware Performance Counters (1/2)



Hardware Performance Counters (2/2)

Special CPU registers:

- Count *performance events*
- Registers must be configured to collect specific performance events
  - Number of CPU cycles
  - Number of instructions executed
  - Number of memory accesses

  ...
- #performance event types $>$ #performance registers

May be inaccurate: not originally built for software developers

Summary

- Performance analysis may require detailed dynamic data
- **Profiler**: Probes stack contents at certain intervals
- **Simulator**:
  - Simulates hardware in software, measures
  - Tends to be inaccurate
- **Performance Counters**:
  - Software:
    - Operating System counts events of interest
  - Hardware:
    - Special registers can be configured to measure CPU-level events

Generality of Performance Measurements?

Measured performance properties are valid for...

- Selected CPU
- Selected operating system
- Compiler version and configuration
- Operating system configuration:
  - OS setup
    (e.g., dynamic scheduler)
  - Processes running in parallel
    ...
- A particular input/output setup
  - Behaviour of attached devices
  - Time of day, temperature, air pressure, ...
- CPU configuration (CPU frequency etc.)

...

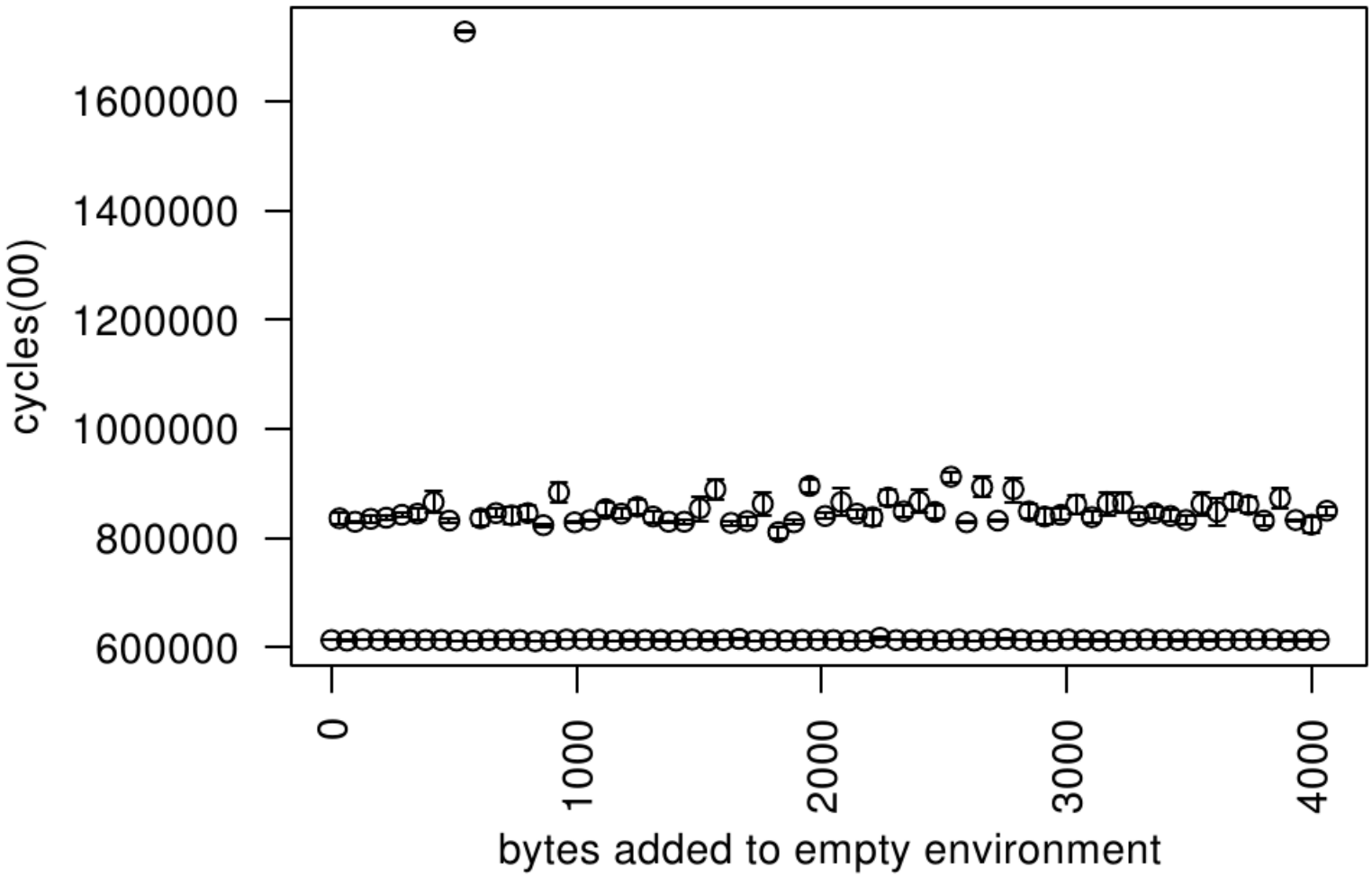
Is that all?

Unexpected Effects

- User `toddm` measures run time 0.6s
- User `amer` measures run time 0.8s
- Both measurements are stable
- Reason for discrepancy:
  - Before program start, Linux copies shell environment onto stack
  - Shell environment contains user name
  - Program is loaded into different memory addresses
    $\Rightarrow$ Memory caches can speed up memory access in one case but not the other

Changing your user name can speed up code

Unexpected Effects[1]

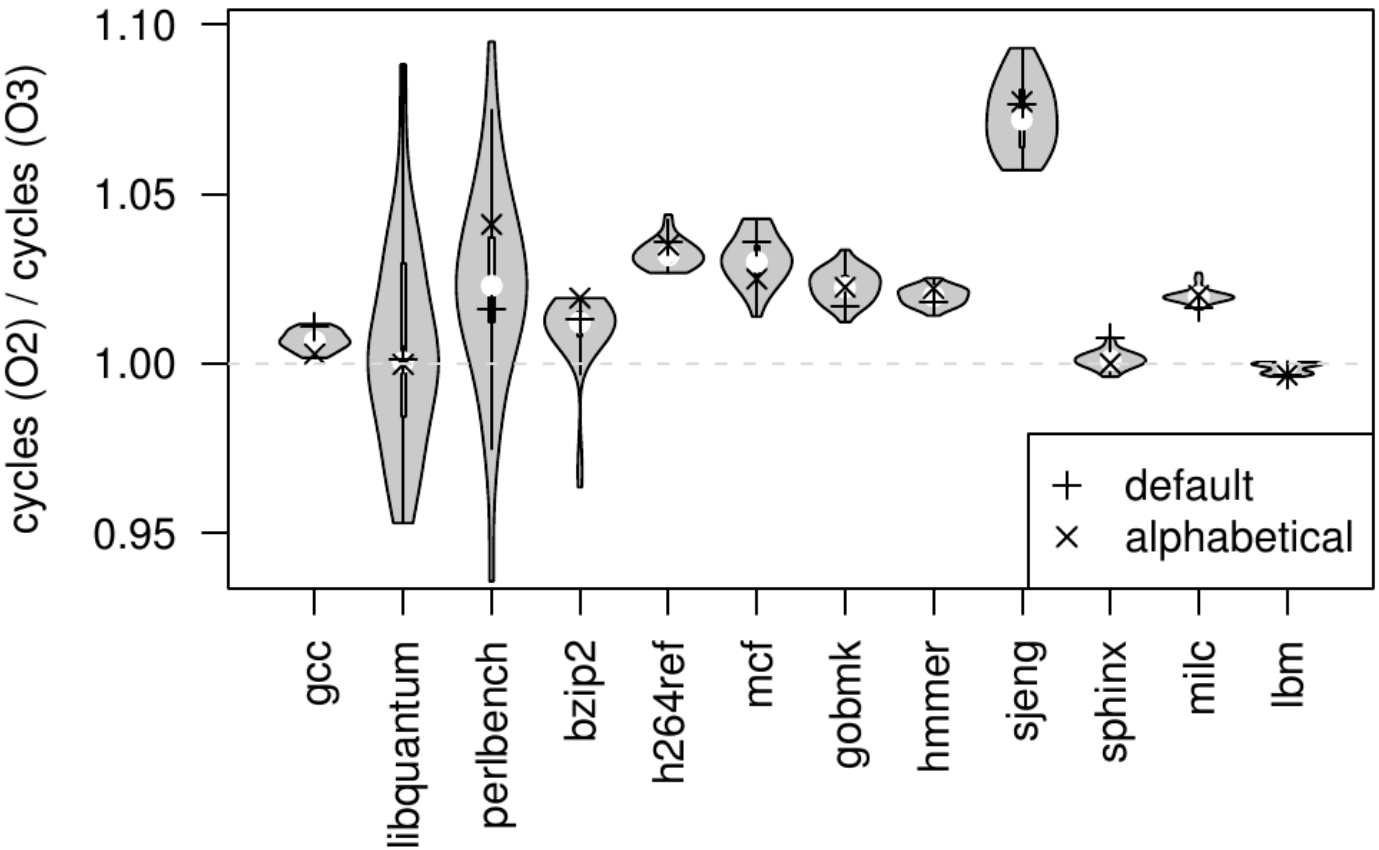


Linking Order

Is there a difference between re-ordering modules in RAM?

`gcc a.o b.o -o program` (Variant 1)

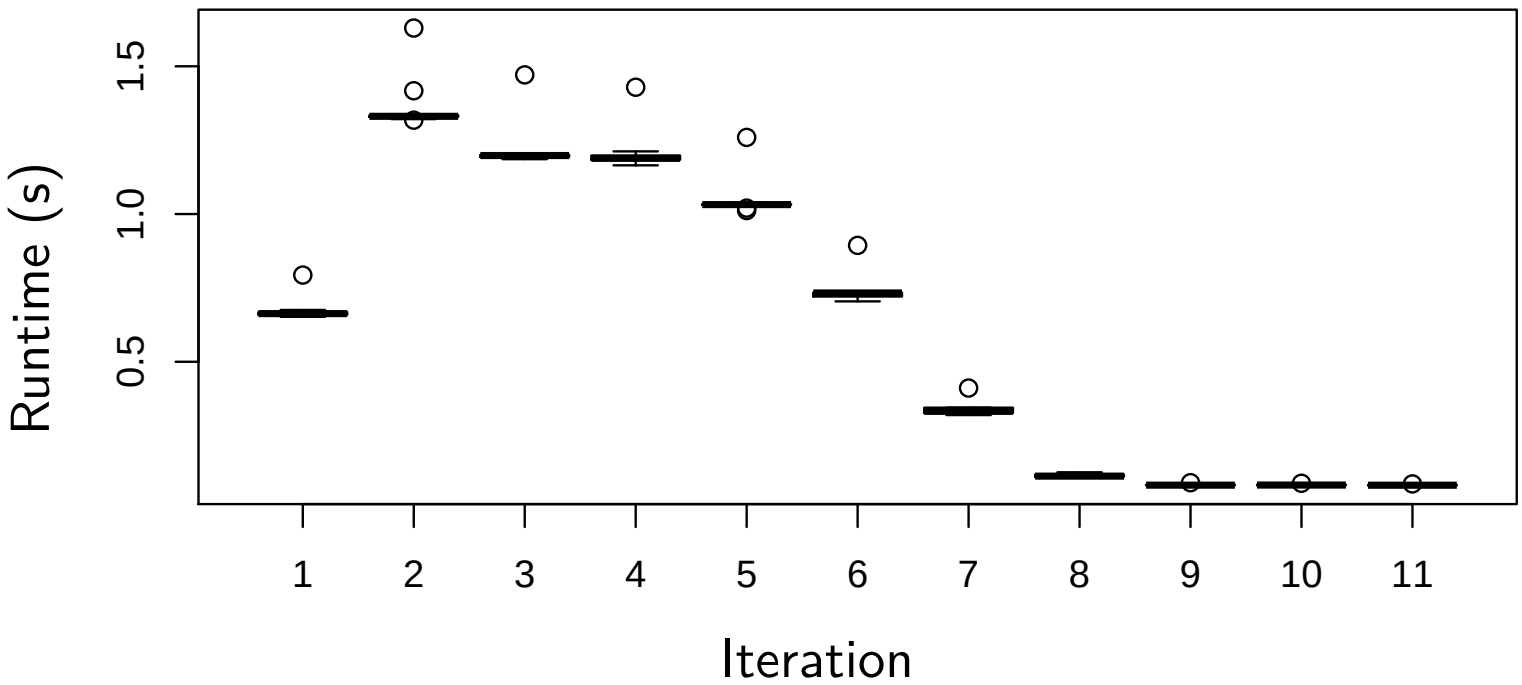
`gcc b.o a.o -o program` (Variant 2)



(Mytkowicz, Diwan, Hauswirth, Sweeney, ASPLOS'09)

Adaptive Systems

- Measurement: 11 runs



Warm-up effect

Warm-Up Effects

- Performance varies during initial runs
- Eventually reaches steady state
- Reason: Adaptive Systems
  - Hardware:
    - *Cache*: Speed up some memory accesses
    - *Branch Prediction*: Speed up some jumps
    - *Translation Lookaside Buffer*
  - Software:
    - *Operating System / Page Table*
    - *Operating System / Scheduler*
    - *Just-in-Time compiler*
- What sbould we measure?
  - Latency: measure first run
    Reset system before every run
  - Throughput: later runs
    Discard initial n measurements

Ignored Parameters

- Performance affected by subtle effects
- System developers must “think like researchers” to spot potential influences

Beware of generalising measurement results!

Summary

- Modern computers are complex
  - *Caches* make memory access times hard to predict
  - *Multi-tasking* may cause sudden interruptions

  ...
- This makes measurements difficult:
  - Must carefully consider what **assumptions** we are making
  - Must measure repeatedly to gather **distribution**
  - Must check for **warm-up effects**
  - Must try to understand causes for performance changes
- Measurements are often not normally distributed
  - Mean + Standard Deviation may not describe samples well
  - If in doubt, use **box plots** or *violin plots*

Dynamic Program Analysis Utility

Dynamic Program Analysis can serve:

- *Understanding*
- *Efficiency*
- *Safety*
- *Security*

Program Understanding

Approaches:

- *Performance analysis* (`gprof`, `papi`, `perf`, ...)
- *Interactive debugging* (`gdb`, `jdb`, ...)
- Tracing
  Compute sequence of actions (trace) of interest
  - Methods
  - Parameters
  - IL/assembly instructions
  - Lines of code

  ...
- Dynamic slicing
  Reduce program to parts that were actually executed
  - Remove dead code
  - Enables further optimisations (e.g., inlining)

Tracing vs. Dynamic Slicing

Source program

```
(0)int f(int x) {
(1)   return x + 1;
(2)}

(3)int g(int x) {
(4)   return x - 1;
(5)}

(6)void main() {
(7)    int x = f(1);
(8)    int y = f(2);
(9)    if (x < 0) {
(A)      puts("fail");
(B)    } else {
(C)      printf("%d",x+y);
(D)    }
(E)}
```

Trace

```
6
7
0[x=1]
1[⇒2]
8
0[x=2]
1[⇒3]
9
B
C
```

Dynamic Slice

```
(0)int f(int x) {
(1)   return x + 1;
(2)}

(6)void main() {
(7)    int x = f(1);
(8)    int y = f(2);

(C)      printf("%d",x+y);

(E)}
```

Tracing/slicing algorithms vary in output

Efficiency

- Dynamic Optimisation
  - Utilise run-time knowledge to optimise
- Speculative Optimisation
  - Type or value seems to be constant?
    - Speculate: it *is* constant
    - Optimise accordingly
  - Add *guard*: is assumption correct?
  - *Deoptimise* when guard fails
  - Common example: *method inlining*
- Challenge: Dynamic analysis introduces overhead
  - Focus efforts on *hot* methods (frequently running)

Safety

- Dynamic type checking
  - Out-of-bounds checks

    `a[i]`
  - Narrowing conversions

    `Object` obj = ...;

    `String` str = (String) obj;
- Assertions
  - Preconditions

    Checked before subroutine call
  - Postconditions

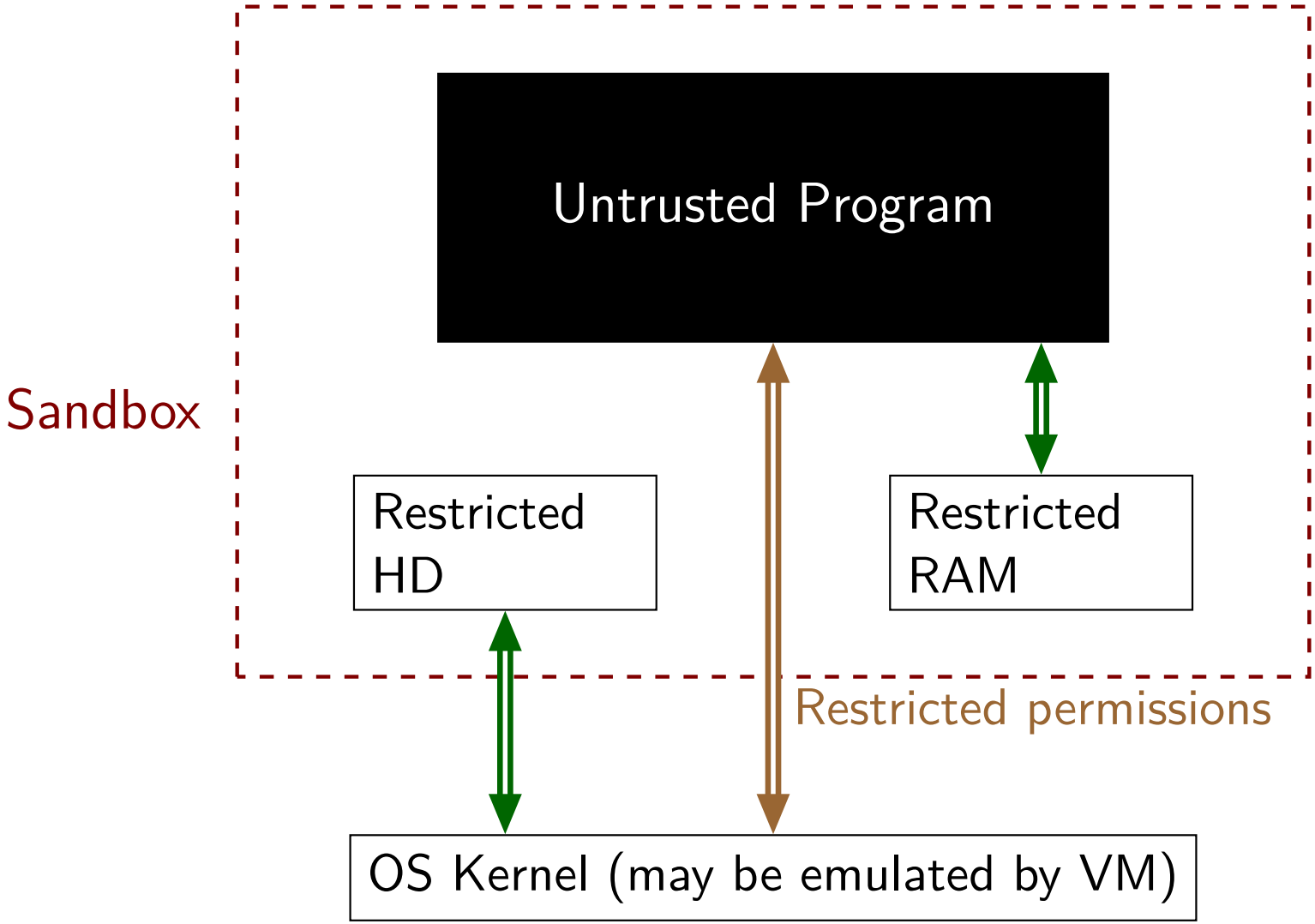
    Checked at end of subroutine call
- Invariants

  Checked between subroutine calls in same module / object

Security

- Which part of program are not trustworthy?
  - Externally loaded code?
  - Externally obtained data?
  - Runtime environment?
- Untrusted code:
  - Confine ((chroot), sandboxing)
- Untrusted data:
  - Sanitise, track
  - Beware: can escalate to untrusted code

Sandboxing: Confining Untrusted Code



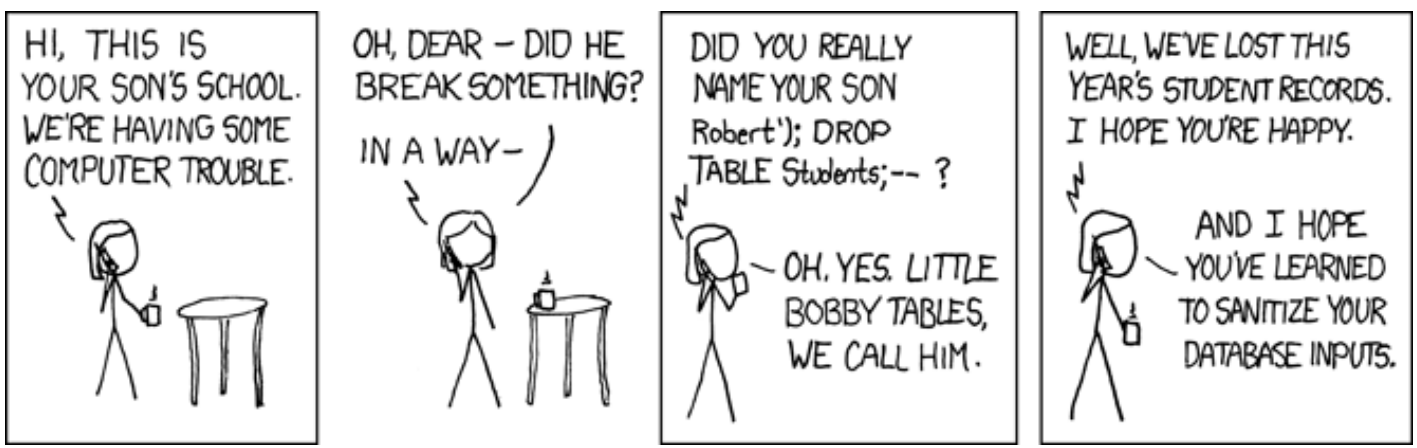
Summary

- Dynamic analysis contributes techniques to all typical clients of program analysis
- *Understanding*:
  - Interactive debugging
  - **Tracing** and **Dynamic Slicing**
- *Efficiency*:
  - **Dynamic** and **speculative optimisation**
- *Safety*:
  - **Dynamic type checking**
  - **Dynamic assertion checking**
- *Security*:
  - **Sandboxing**, i.e., executing in restricted execution environment
  - **Dynamic Taint analysis**

Tainted Values (1/2)

Python

```
username = request.GET['user']
...
q = sql.query("SELECT * from Users WHERE name='"
              + username + "'")
user_data = q.run
```

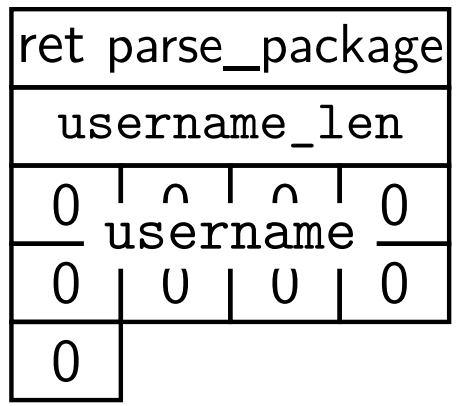


Tainted Values (2/2)

C

```
int parse_package(s* out, uint8* data) {
  char username[9] = { 0 };
  int username_len = data[0];
  // spec says:  length <= 8
  memcpy(username, data+1, username_len);
  ...
}
```

Stack

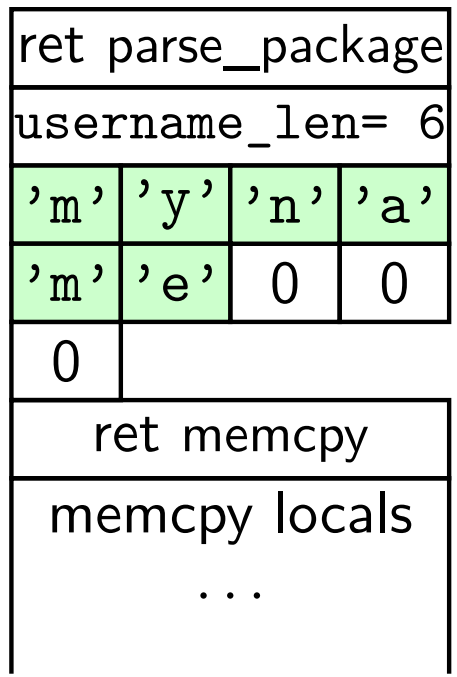


Tainted Values (2/2)

C

```
int parse_package(s* out, uint8* data) {
  char username[9] = { 0 };
  int username_len = data[0];
  // spec says:  length <= 8
  memcpy(username, data+1, username_len);
  ...
}
```

Stack

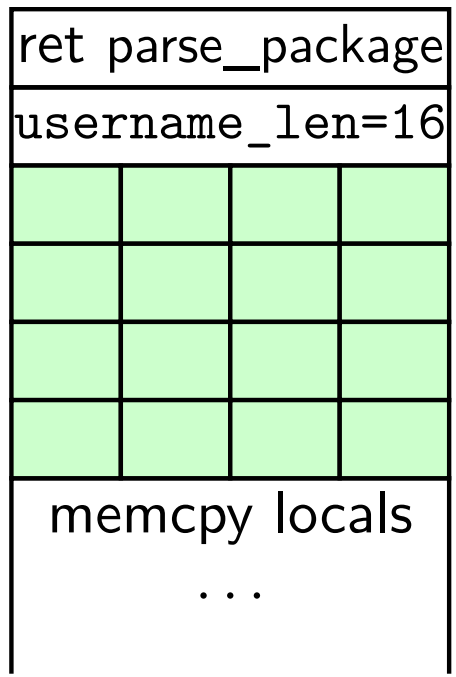


Tainted Values (2/2)

C

```
int parse_package(s* out, uint8* data) {
  char username[9] = { 0 };
  int username_len = data[0];
  // spec says:  length <= 8
  memcpy(username, data+1, username_len);
  ...
}
```

Stack



Tracing 'Tainted' Values

Taint Analysis:

- Track *tainted* values
- Remove taint if values are *sanitised*
- Detect if they reach sensitive *sinks*
- NB: Static taint analysis may also be possible

Unsafe input

- **Taint source**: Network ops
- **Sanitiser**: SQL string escape
- **Taint sink**: SQL query string

Leaking secrets

- **Taint source**: Plaintext passwd.
- **Sanitiser**: cryptographic hash
- **Taint sink**: Network ops

Dynamic Taint Analysis

```
query_l = "SELECT ...'"
query_r = "'"
username = request.GET['user']
...
query_str = query_l + username
query_str = query_str + query_r
q = sql.query(query_str)
```

```
query_l = "SELECT ..."ϵ
query_r = "'"ϵ
username = "..."t

query_str = "..."t
query_str = "..."t
Fault!
```

Dynamic Taint Analysis

Strategy:

- Annotate tainted values with *taint tags* or *shadow values*

```
s = read_network() // string in s will be tainted
t = "foo" + "bar"  // string in t will be untainted
```

- Extend operators to propagate taint:

$\oplus$	ϵ	t
ϵ	ϵ	t
t	t	t

$$\texttt{"foo"}^v[1] = \texttt{"o"}^v$$

$$\texttt{"foo"}^v + \texttt{"bar"}^w = \texttt{"foobar"}^{v \oplus w}$$

- Check taint sinks for tainted input
- Needs instrumentation (shadow values) or explicit support by runtime (e.g., Perl, Ruby)

Conditionals

- Should conditionals propagate taint?
- Usually such *control dependencies* don't propagate taint

Python

```
if secret_password == '':
    network_send('Account disabled, cannot log in');
```

Attackers vs. Taint Ananlysis

Is taint analysis 'sound enough' to detect attempts to expose sensitive data?

- Often-proposed technique: Taint analysis in Dalvik VM
- *Can attackers subvert this analysis?*

C

```
for (i = 0; i < 16; ++i) {
  for (k = 0; k < 8; ++k) {
    if (secret_password[i] & 1 << k) {
      network_send("Meaninless Message");
    } else {
      network_send("Something Else");
    }
} }
```

System Command Attack

C

```
char d_secret[1024];
strcpy(d_secret, "/tmp/");
strcat(d_secret, secret); // taint d_secret

int iopipes[2];
pipe(iopipes);
...
if (fork()) { // create child process
  // connect pipes
  execv("/bin/rm", d_secret); // call external 'rm'
}
char[1024] buf; // untained!
read(iopipes[0], ...); // read output from 'rm'
```

System call will print e.g.:

```
rm:  cannot remove '/tmp/mysecretstring':  No such file or
directory
```

Side Channel Attacks

Many more attacks possible:

- Timing attacks:
  - Two threads
  - One sends signal to other, with delays
  - Delay loop length dependent on secret
- File length attack:
  - Write dummy file
  - File length (or other metadata) encodes secret
- Graphics buffer attack:
  - Write to screen
  - Read back with OCR
  - Or adjust widget position / font size to encode secret

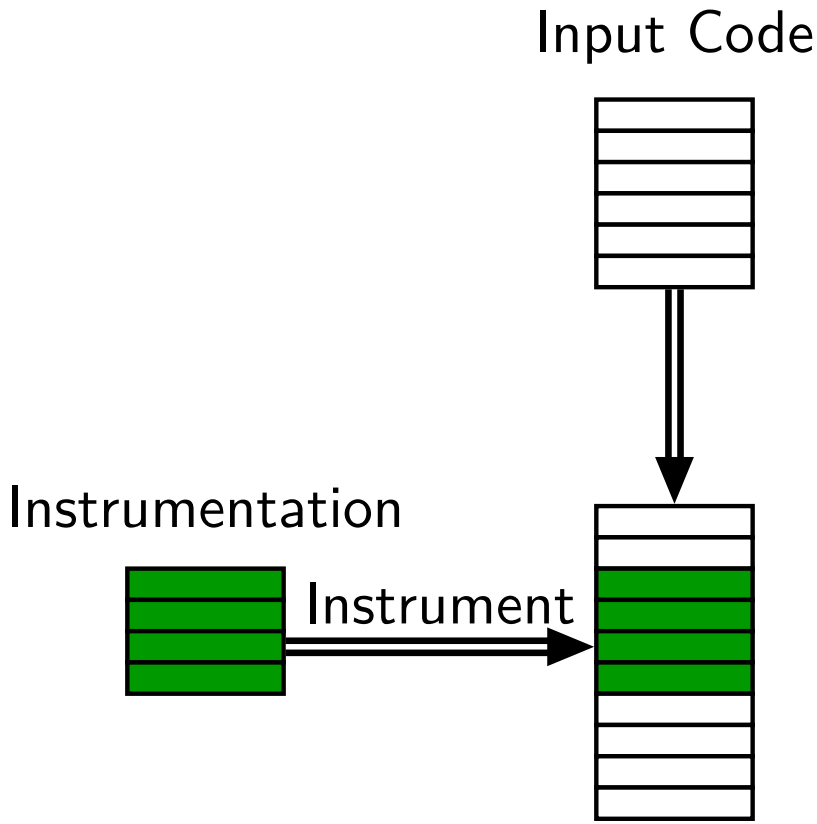
Summary

- Dynamic taint analysis tracks **tainted** values (from **taint sources**)
- Tags also referred to as **shadow values**
- Removes taint if values are **sanitised**
- Detects attempts to use tainted values in **taint sinks**
- Still many weaknesses in analysis:
  - Control-dependence attacks
  - System command attacks
  - Side-channel attacks
- Can be strengthened with *symbolic* techniques (later lectures!)

Dynamic Binary Analysis

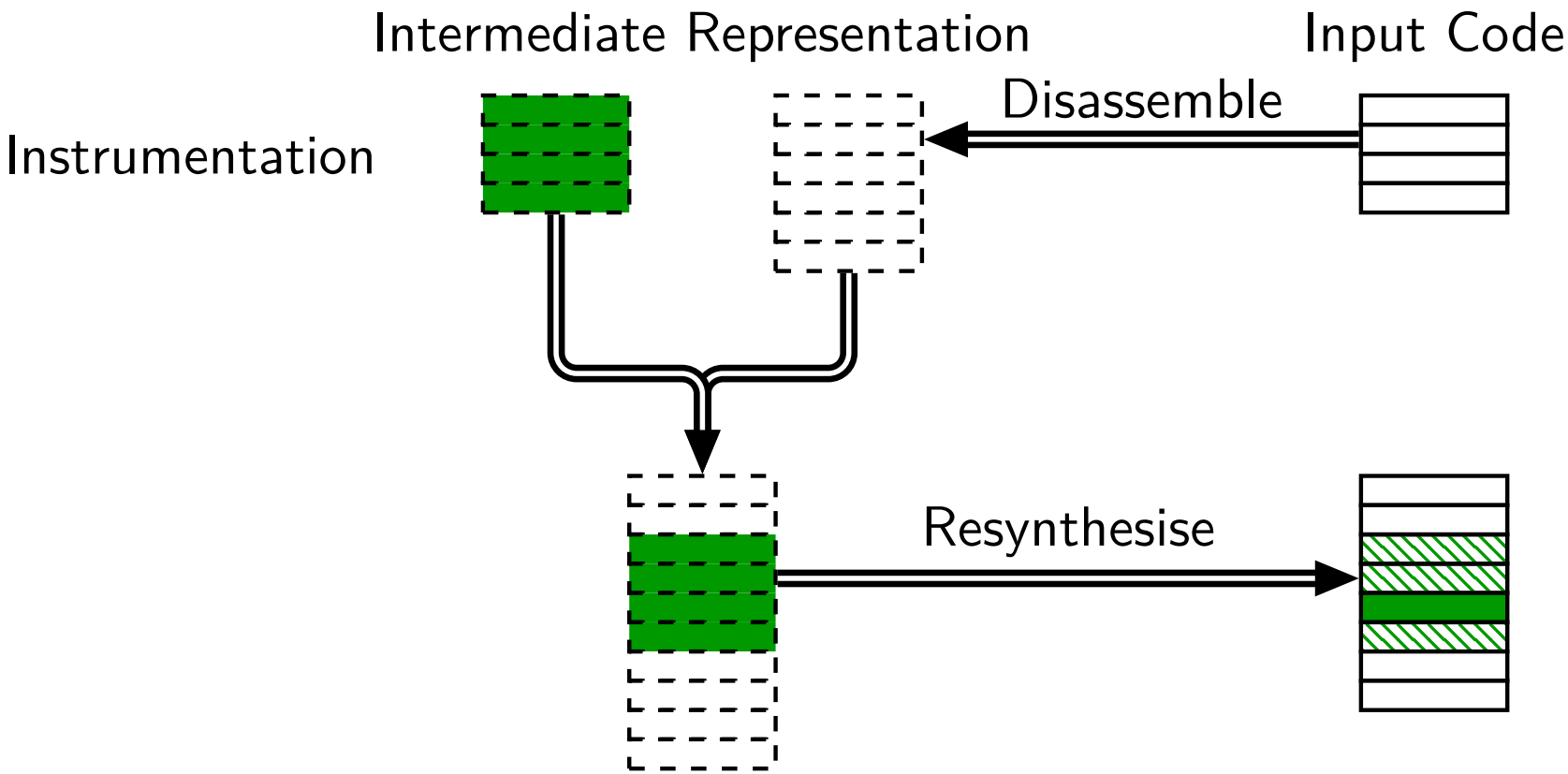
- *Binary Analysis*: Analyse binary executables
  - Applicable to any executable program
  - Only requires binary code
  - Unaware of source language
- *Dynamic Binary Analysis*
  - Analyser runs concurrently with program-under-analysis
  - Can adaptively instrument / analyse / intercede

Dynamic Binary Instrumentation (1/3)



Copy-and-Annotate

Dynamic Binary Instrumentation (2/3)



Disassemble-and-Resynthesise

Dynamic Binary Instrumentation (3/3)

- *Copy-and-Annotate* (e.g., `pin`):
  - Inserts code into binary
  - Inserted code must maintain state (registers!)
- *Disassemble-and-Resynthesise* (e.g., `valgrind`, `qemu`):
  - Decomposes program into IR
  - Instrumentation on IR-level
  - Easier/faster to track shadow values in some cases
    - *Shadow registers*
    - *Shadow memory*
    - Must model *system calls* for proper tracking

Application: Finding Memory Errors

- Reads from uninitialised memory in C can trigger undefined behaviour
- Approach: Track information: which bits are uninitialised?
- Requires *shadow registers*, *shadow values*
- Almost every instruction must be instrumented

Shadow values	Program
x: 	`short x;`
x: 	`x |= 0x7;`
x: 	`if (x & 0x10) {`
	`...`

Example: Valgrind's Memcheck

- Valgrind is Disassemble-and-Resynthesise-style Binary Instrumentation tool
- Memcheck: tracks memory initialisation (mostly) at bit level
  - Less precise for floating point registers
- Valgrind uses dynamic translation:
  - Translate & instrument blocks of code at address until return / branch
  - Instrumented code jumps back into Valgrind core for lookup / new translation

Challenges

- System calls
  - System calls may affect shadow values (e.g., propagate taintedness)
  - Must be modelled for precision
- Self-modifying code
  - Used e.g. in GNU libc
  - Must be detected, force eviction of old code (expensive checks!)

Valgrind

- Binary instrumenter
- Available platforms:
  - x86/Linux (partial) and Darwin
  - AMD64/Linux and Darwin
  - PPC64/Linux, PPC64LE/Linux ($\leq$ Power8)
  - S390X/Linux
  - ARM(64)/Linux ($\geq$ ARMv7)
  - MIPS32/Linux, MIPS64/Linux
  - Solaris
  - Android
- Analyses (focus on *Simulation*):
  - Call analysis
  - Cache analysis
  - Memcheck

Qemu

- Binary instrumenter and translator
- Focus on *emulation*
- Runs kernel + user space
- Translate from one ISA to another (e.g., run ARM on ADM64)
- Emulates system:
  - Graphics, networking, sound, input devices, USB, ...
- Almost two dozen platforms supported

Summary

- **Binary instrumentation** is a form of low-level dynamic analysis
- Two main schemes:
  - **Copy-and-Annotate**: insert new code
  - **Disassemble-and-Resynthesise**: merge analysis subject code with annotation code
- Shadow values supported through **shadow registers** and **shadow memory**

Slowdown in Valgrind (1/4)

- Performance comparison often against *baseline*:
  A 'norm' that we compare against
- *Speedup* of some alternative n (against a baseline):
  # of times that n can execute workload while baseline executes workload once

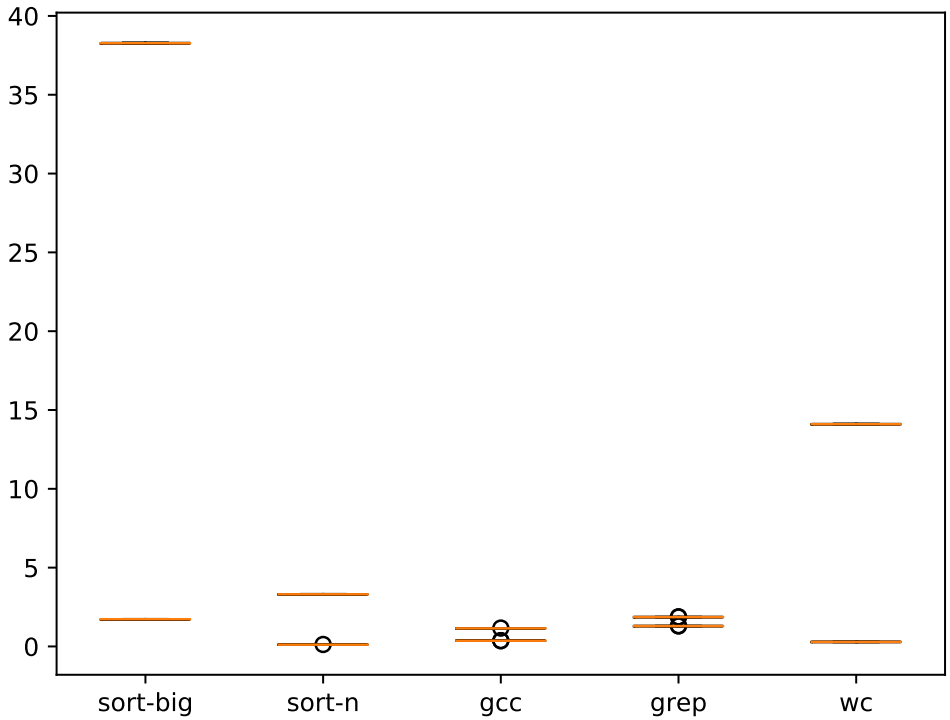
$$\text{speedup}(n) = \frac{\text{baseline time}}{\text{time for } n}$$

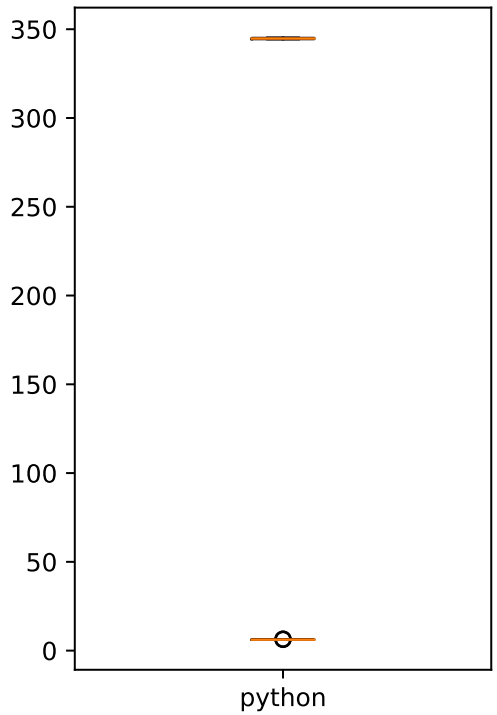
Conversely:

$$\text{slowdown}(n) = \frac{1}{\text{speedup}(n)}$$

Slowdown in Valgrind (2/4)

Comparison of execution time with/without valgrind (valgrind on top):





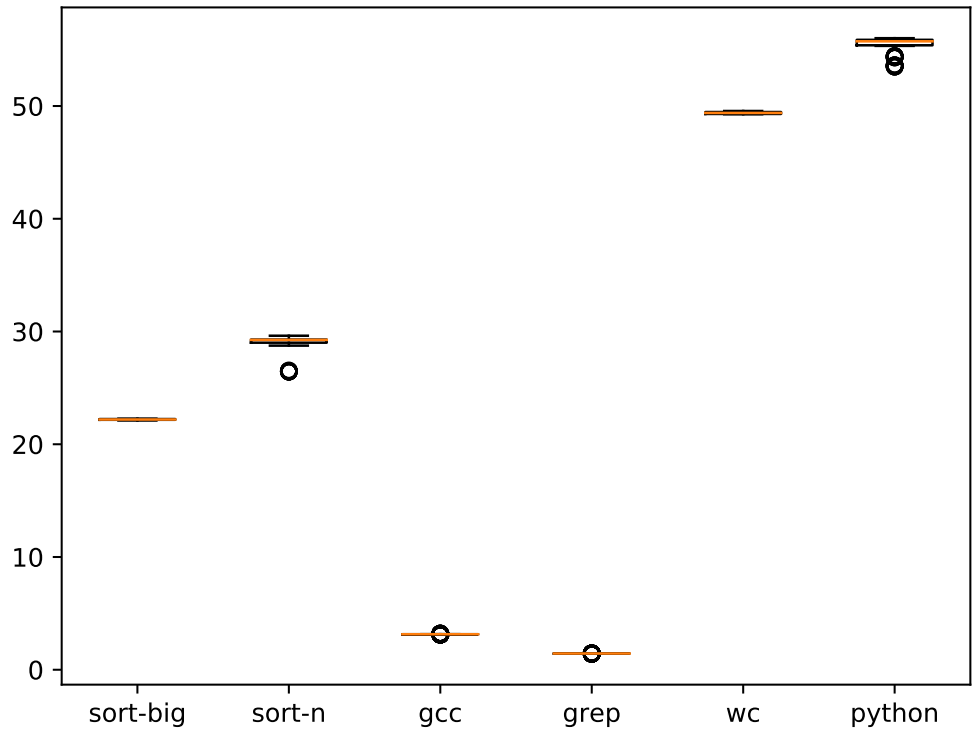
Slowdown distribution?

Slowdown in Valgrind (3/4)

- 2 samples per program:
  - 11 baseline runs
  - 11 valgrind runs
- All statistically independent

$\Rightarrow$ Can compare each against each other

- $11 \times 11 = 121$ measurements



Slowdown in Valgrind (4/4)

Median slowdown:

sort-big	22.19
sort-n	29.26
gcc	3.15
grep	1.44
wc	49.38
python	55.74

What's the average slowdown over everything?

Summarising benchmark suites: normalisation

- Normalise runtimes against *baseline*
- Example: SPECint (CPU benchmark): VAX 11/780
- *Speedup* ist a form of normalisation

Program	**Runtime**	**Speedup**	**System**
P_1	500	1	baseline
P_2	40	1	baseline
P_1	100	5	A
P_2	2	20	A
P_1	80	6	B
P_2	20	2	B

- Speedups are *proportions*:
- *Geometric* mean:

System A $\sqrt{5 \times 20} = 10$

System B $\sqrt{6 \times 2} \approx 3{,}46\ldots$

The geometric mean

- $t(P_k, M)$: Runtime of program P_k on system M
- $\frac{t(P_k,baseline)}{t(P_k,M)}$: speedup of program P_k on system M over baseline

Geometric mean:

$$\mu_g = \sqrt[n]{\frac{t(P_1,\text{baseline})}{t(P_1,A)} \times \ldots \times \frac{t(P_n,\text{baseline})}{t(P_n,A)}}$$

Geometric standard deviation:

$$\sigma_g = e^{\sqrt{\frac{(\ln(\frac{t(P_k,\text{baseline})}{t(P_1,M)}) - \ln(\mu_g))^2 + \ldots + (\ln(\frac{t(P_k,\text{baseline})}{t(P_n,M)}) - \ln(\mu_g))^2}{n}}}$$

Advantages of the Geometric Mean

$$\frac{\sqrt[n]{\frac{t(P_1,\text{baseline})}{t(P_1,A)} \times \ldots \times \frac{t(P_n,\text{baseline})}{t(P_n,A)}}}{\sqrt[n]{\frac{t(P_1,\text{baseline})}{t(P_1,B)} \times \ldots \times \frac{t(P_n,\text{baseline})}{t(P_n,B)}}}$$

$$= \sqrt[n]{\frac{\frac{t(P_1,\text{baseline})}{t(P_1,A)} \times \ldots \times \frac{t(P_n,\text{baseline})}{t(P_n,A)}}{\frac{t(P_1,\text{baseline})}{t(P_1,B)} \times \ldots \times \frac{t(P_n,\text{baseline})}{t(P_n,B)}}}$$

$$= \sqrt[n]{\frac{t(P_1,B)}{t(P_1,A)} \times \ldots \times \frac{t(P_n,B)}{t(P_n,A)}}$$

When directly comparing the geometric means of systems A and B, the baseline is irrelevant

Measuring Performance

1. Does compiler A produce faster code than compiler B?
2. Does machine A run code faster than machine B?

- Application
- Benchmark suite
- Synthetic benchmark suite (unrealistic)
- Small programs (unrealistic)
- (Micro)kernels (unrealistic)
- MIPS/FLOPS (operations "'per second"'– but *what* operations? How representative?)

Benchmark Suites

- Standardised collections of software with different ‘typical’ loads
- Advantages:
  - Standardisation simplifies comparability
  - Differentiated pieces of software avoid over-specialisation / unrealistic optimisations / lack of realism
- Disadvantages:
  - Workloads for software may not be representative of actual application use
  - Software and programming practices evolve, benchmark static suites may become obsolete
- Exapmles:
  - SPEC (CPUs, focus on C)
  - TPC (Databases)
  - DaCapo, XCorpus (Java applications)

Benchmark Suites (Example)

Consist of several programs, e.g., ***DaCapo****:*

avrora	Multiprocessor simulation
batik	Draws SVG images
eclipse	Eclipse IDE performance test
fop	Translates XSL-FO file into PDF or PS file
h2	Banking transaction simulator
pmd	Static checker for Java
sunflow	Ray tracer (computes images)
tomcat	Web server
xalan	Translates XML to HTML

- +5 more

Assumption: All programs 'equally important'

Summary

- Ideal measurement: Runtime of the intended program in all possible configurations
- If not practical:
  - Prepresentative benchmark suites
  - Measure minimal changes between configuration
    $\Rightarrow$ Analyse effect of individual changes
- Summarise benchmark suite results: geometric mean:

$$\sqrt[n]{\frac{t(P_1,A)}{t(P_1,\text{baseline})} \times \ldots \times \frac{t(P_n,A)}{t(P_n,\text{baseline})}}$$

Review

- Performance Counters
- Challenges in Dynamic Performance Analysis
- Taint Analysis
- Binary Instrumentation
- Benchmark Suites

To be continued...

- Type Analysis
- Operational Semantics