EDAP15: Program Analysis

DYNAMIC PROGRAM ANALYSIS 2

Christoph Reichenbach

Automatic Performance Measurement

- Profiler:
  - Interrupts program during execution
  - Examines call stack

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)

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.)

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?

Execution Stack

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

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)
...

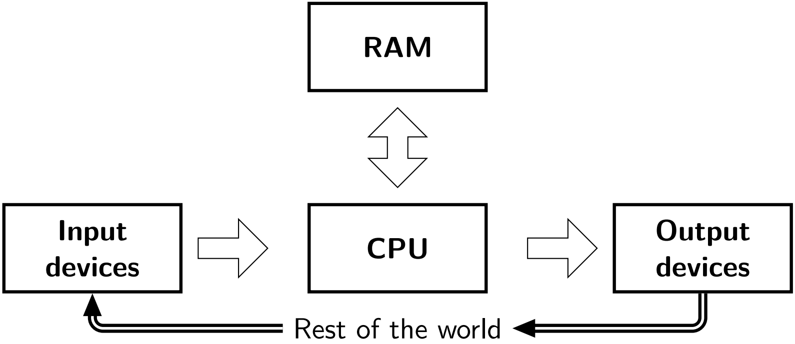
Profiler

- 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

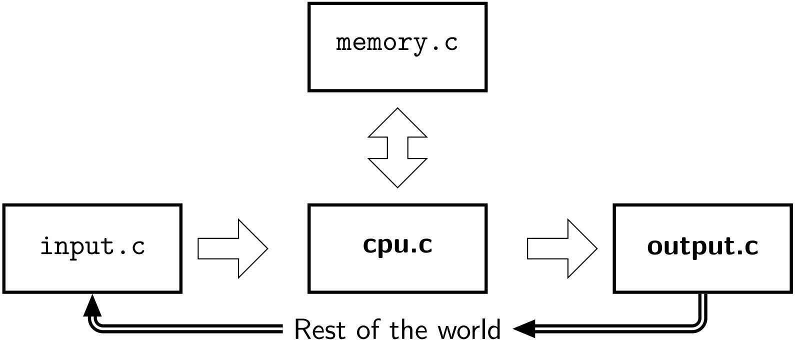
Execution Stack
`return` (alt-1)
`$fp` (alt-1)
...
...
`return` (alt-2)
`$fp` (alt-2)
...

Can be inaccurate: misses short function calls

Simulator

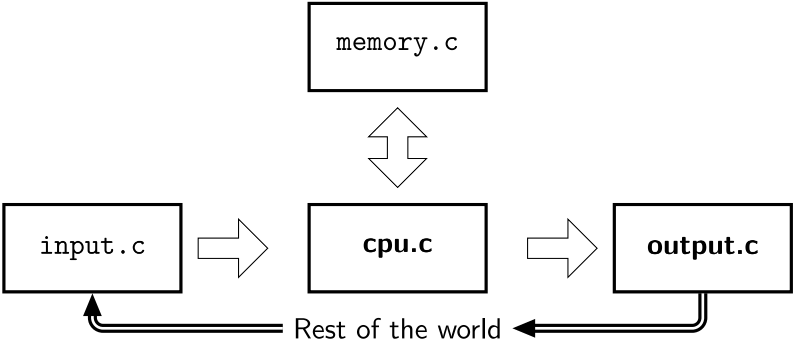


Simulator



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

Simulator



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

Modern CPUs are very complex: Simulators tend to be inaccurate

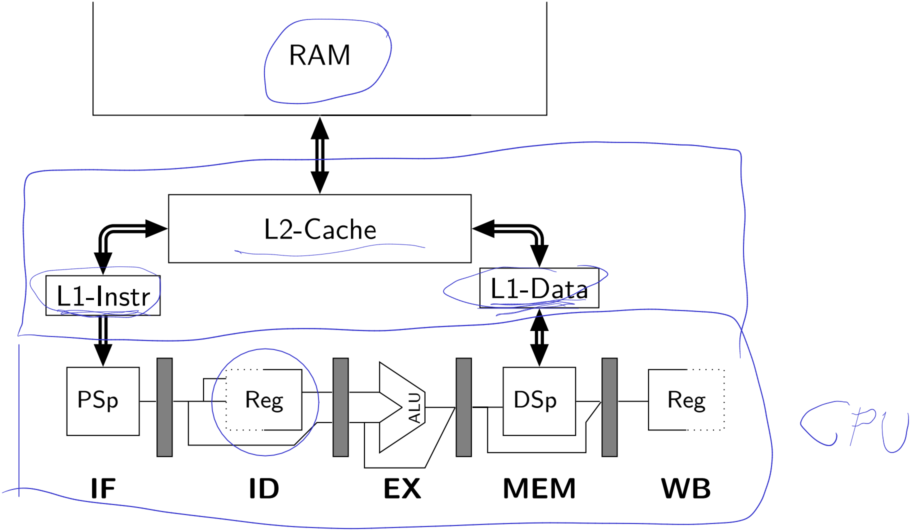
Software Performance Counters

- 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?

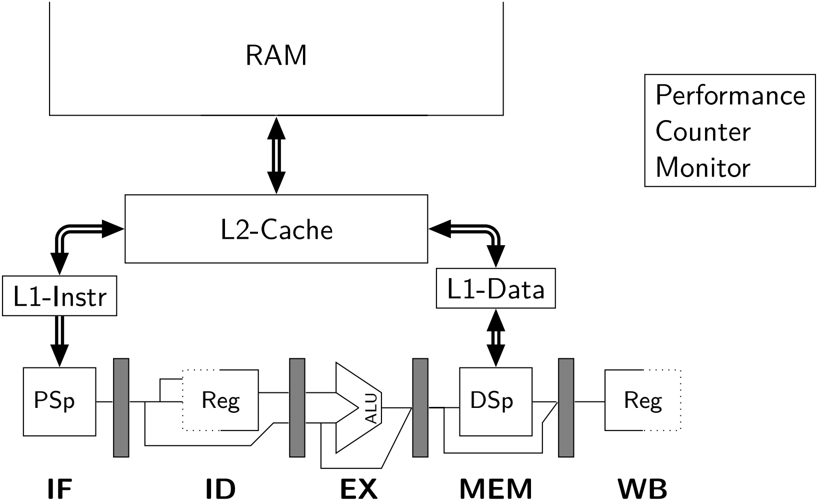
Software Performance Counters

- 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

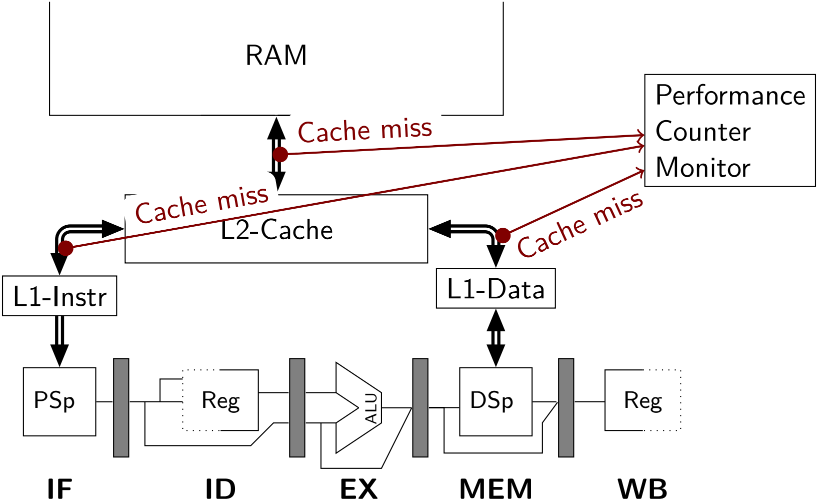
Hardware Performance Counters (1/2)



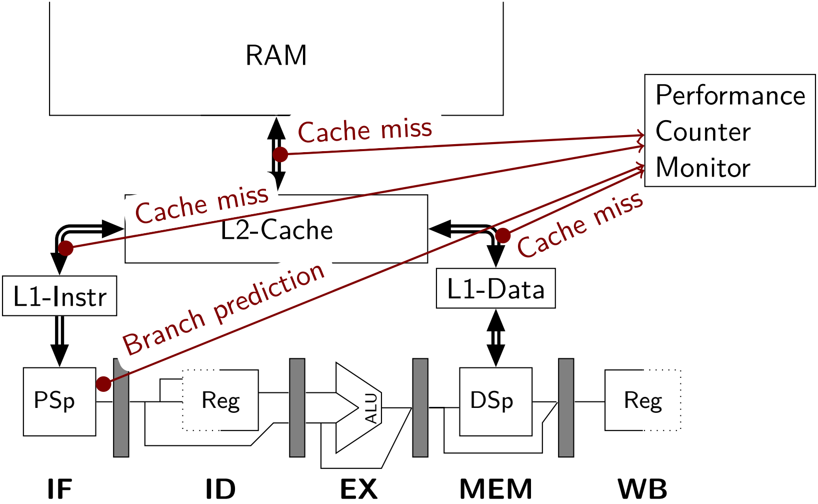
Hardware Performance Counters (1/2)



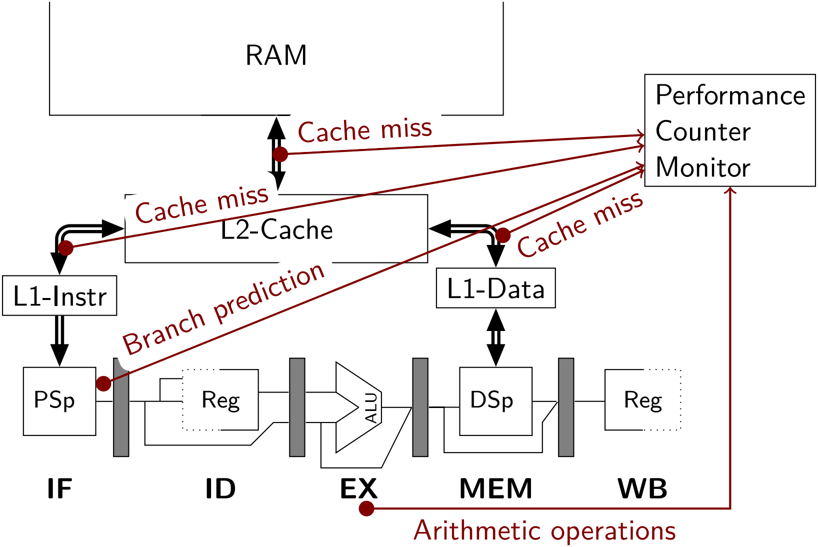
Hardware Performance Counters (1/2)



Hardware Performance Counters (1/2)



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

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

Is that all?

Generality of Performance Measurements?

Measured performance properties are valid for...

- Selected CPU
- Selected operating system
- Compiler version and configuration

Is that all?

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

  ...

Is that all?

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, ...

Is that all?

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

Unexpected Effects

- User `toddm` measures run time 0.6s
- User `amer` measures run time 0.8s

Unexpected Effects

- User `toddm` measures run time 0.6s
- User `amer` measures run time 0.8s
- Both measurements are stable

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

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

Unexpected Effects

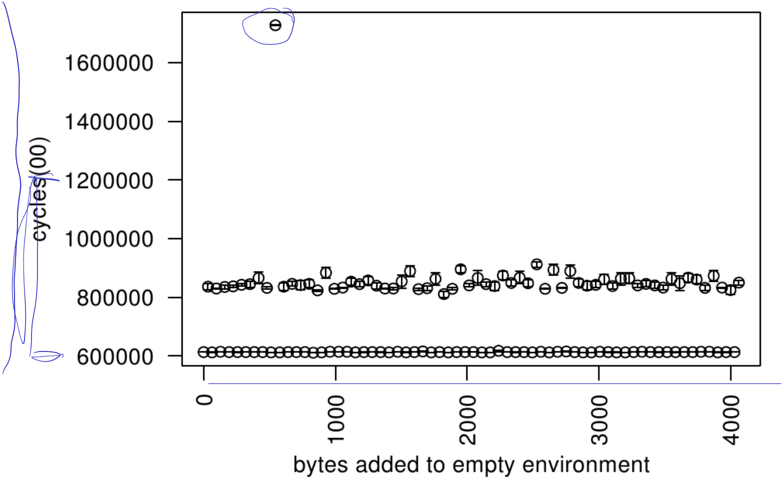
- User `toddm` measures run time 0.6s
- User `amer` measures run time 0.8s
- Both measurements are stable
- Reason for discrepancy:
  - Before program start, Linux copies shell environment onto stack
  - Shell environment contains user name
  - Program is loaded into different memory addresses
    $\Rightarrow$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

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



Mytkowicz, Diwan, Hauswirth, Sweeney: "Producing wrong data without doing anything obviously wrong", in ASPLOS 2009

Linking Order

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

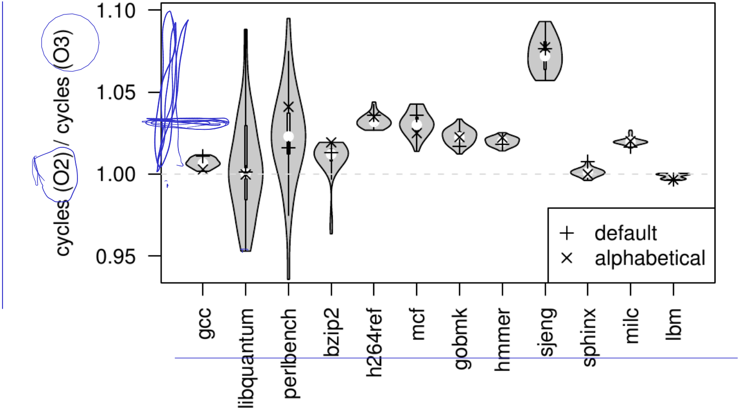
`gcc a.o b.o -o program` (Variant 1)

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

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)
```



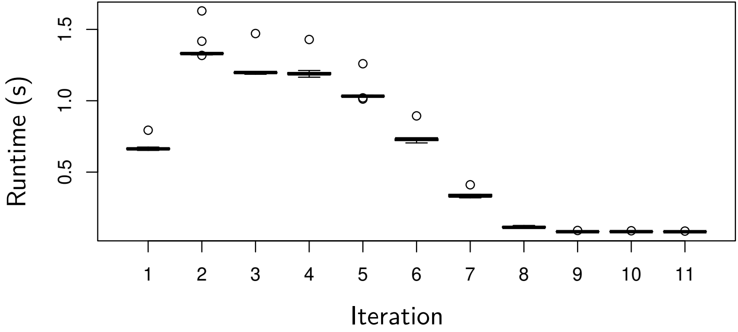
(Mytkowicz, Diwan, Hauswirth, Sweeney, ASPLOS'09)

Adaptive Systems

- Measurement: 11 runs

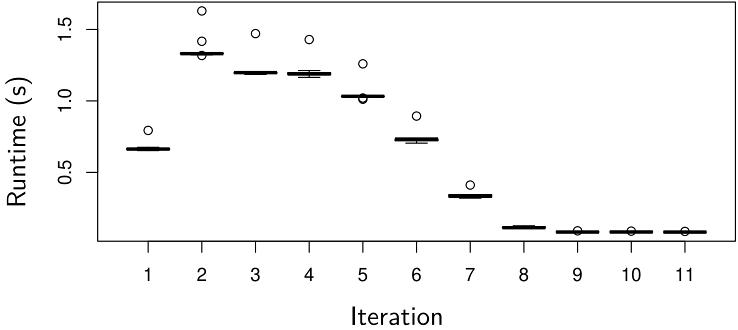
Adaptive Systems

- Measurement: 11 runs



Adaptive Systems

- Measurement: 11 runs



Warm-up effect

Warm-Up Effects

- Performance varies during initial runs
- Eventually reaches steady state

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*

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

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*

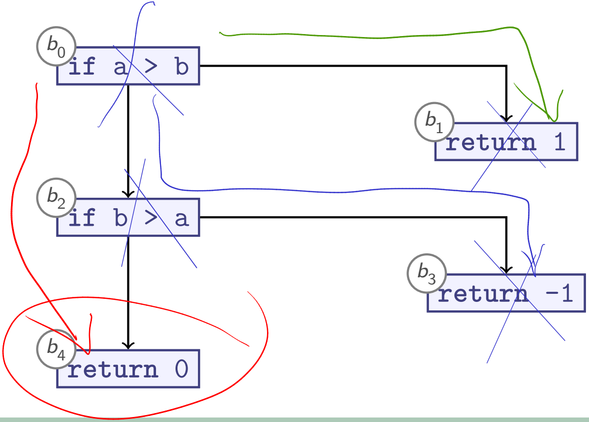
Unit Tests

Teal

```
fun cmp(a, b) = {
  if a > b {
    return 1;
  }
  if a < b {
    return -1;
  }
  return 0;
}

fun test() = {
  assert cmp(1, 2) == -1;
  assert cmp(2, 1) == 1;
}
```

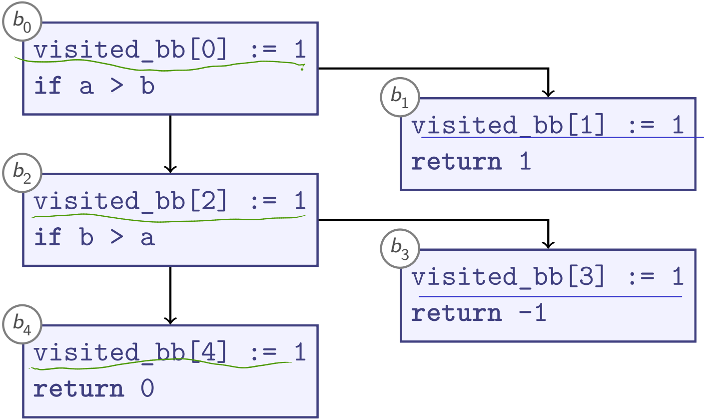
Unit Test Quality



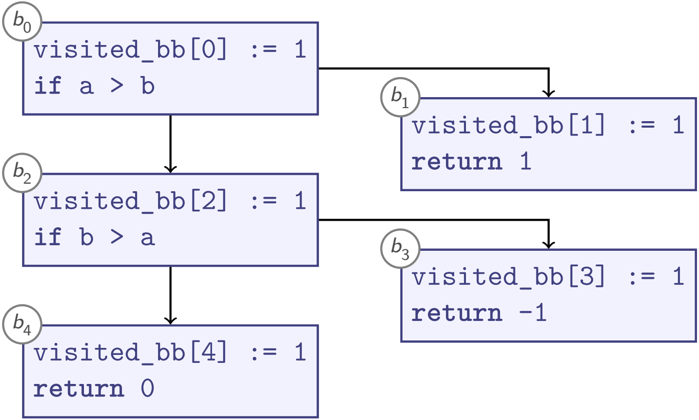
Teal

```
fun test() = {
  assert cmp(1, 2) == -1;
  assert cmp(2, 1) == 1;
}
```

Test Coverage



Test Coverage



- Test coverage = fraction of `visited_bb` elements updated

Test Coverage Properties

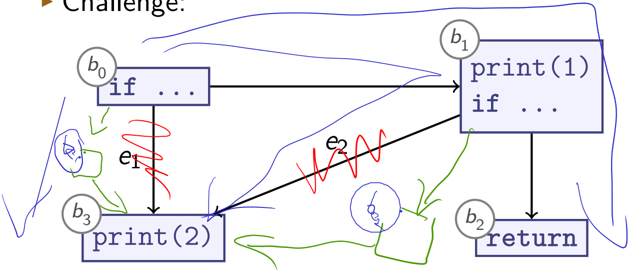
- **Statement Coverage**: is each statement executed?
  $\iff$ each Basic Block is executed

Test Coverage Properties

- **Statement Coverage**: is each statement executed?
  $\iff$ each Basic Block is executed
- **Edge Coverage**: is each CFG edge taken?

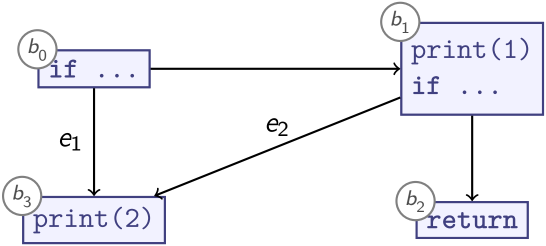
Test Coverage Properties

- **Statement Coverage**: is each statement executed?
  $\iff$ each Basic Block is executed
- **Edge Coverage**: is each CFG edge taken?
  - Challenge:

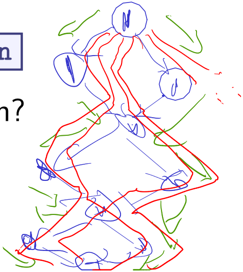


Test Coverage Properties

- **Statement Coverage**: is each statement executed?
  $\iff$ each Basic Block is executed
- **Edge Coverage**: is each CFG edge taken?
  - Challenge:

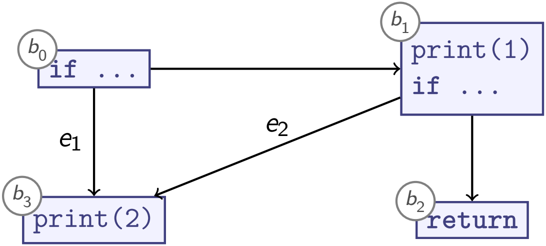


- **Path Coverage**: is each CFG path taken?



Test Coverage Properties

- **Statement Coverage**: is each statement executed?
  $\iff$ each Basic Block is executed
- **Edge Coverage**: is each CFG edge taken?
  - Challenge:



- **Path Coverage**: is each CFG path taken?
  - Need to limit Number of loop iterations checked
  - Must restart tracking block coverage on every method entry

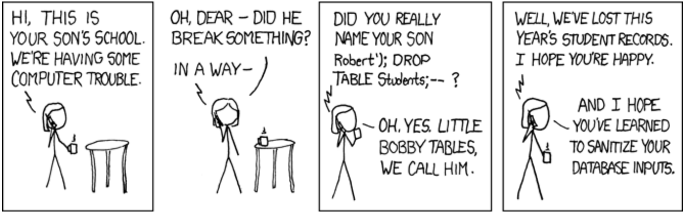
Summary

- **Unit Tests** are a simple form of dynamic program analysis
  - Minimal tooling needed
  - Custom checks
  - Limited to what underlying language can express directly
- **Test Coverage** tells us how much of our code gets analysed by at least one unit test
- Implement by setting markers on relevant basic blocks
- Different criteria, such as:
  - **Statement Coverage**
  - **Edge Coverage**: may require helper BBs
  - **Path Coverage**: paths through CFG (usually excluding loops)

Tainted Values (1/2)

Python

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



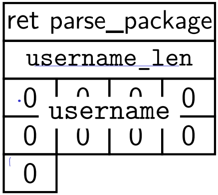
Tainted Values (2/2)

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);
  ...
}
```

Tainted Values (2/2)

Stack



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);
  ...
}
```

Tainted Values (2/2)

C

```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

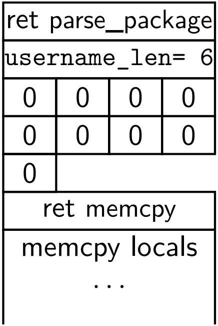
ret parse_package			
username_len= 6			
0	0	0	0
0	0	0	0
0			

Tainted Values (2/2)

C

```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

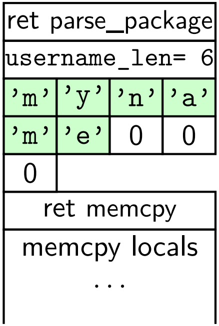


Tainted Values (2/2)

C

```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

```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

ret parse_package			
username_len			
0	0	0	0
0	0	0	0
0			

Tainted Values (2/2)

C

```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

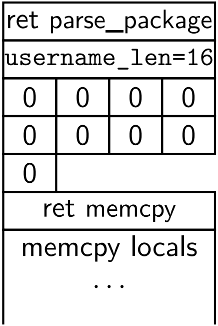
ret parse_package			
username_len=16			
0	0	0	0
0	0	0	0
0			

Tainted Values (2/2)

C

```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

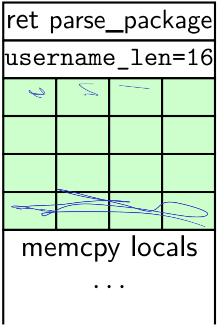


Tainted Values (2/2)

Stack

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);
  ...
}
```



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)
```

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 ..."
query_r = "'"
username = "..."

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 ...'"$^{\epsilon}$
query_r = "'"$^{\epsilon}$
username = "..."

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)
```

$\texttt{query\_l} = \texttt{"SELECT ..."}^{\epsilon}$

$\texttt{query\_r} = \texttt{"'"}^{\epsilon}$

$\texttt{username} = \texttt{"..."}^{t}$

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 ..."$^{\epsilon}$

query_r = "'"$^{\epsilon}$

username = "..."^{t}

query_str = "..."^{t}

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 ..."$^{\epsilon}$
query_r = "'"$^{\epsilon}$
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?

Python

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

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?*

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

```
if (secret_password[i] & 1) {
  network_send("Meaninless Message");
} else {
  network_send("Something Else");
}
```

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 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

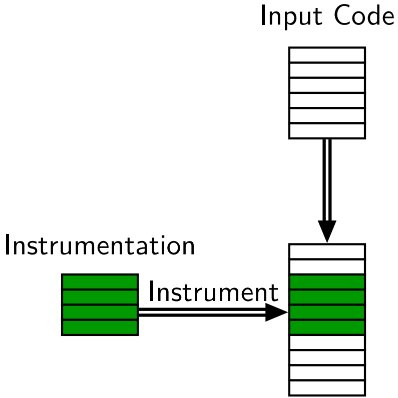
Summary

- 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

Dynamic Binary Analysis

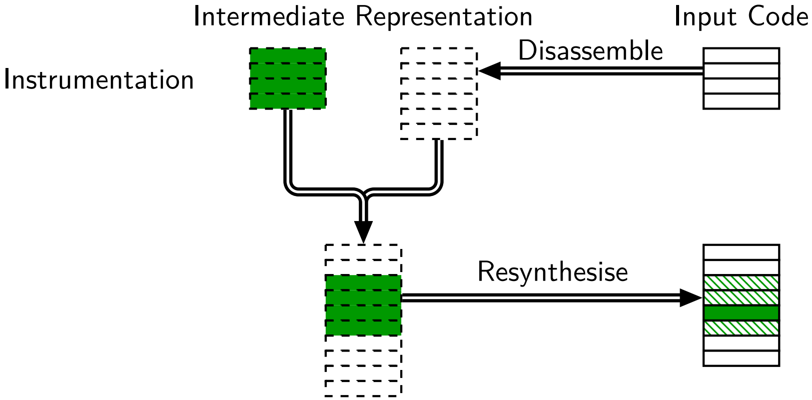
- *Binary Analysis*: Analyse binary executables
  - Applicable to any executable program
  - Only requires binary code
  - Unaware of source language
- *Dynamic Binary Analysis*
  - Analyser runs concurrently with program-under-analysis
  - Can adaptively instrument / analyse / intercede

Dynamic Binary Instrumentation (1/3)



Copy-and-Annotate

Dynamic Binary Instrumentation (2/3)



Disassemble-and-Resynthesise

Dynamic Binary Instrumentation (3/3)

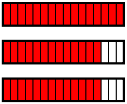
- *Copy-and-Annotate* (e.g., `pin`):
  - Inserts code into binary
  - Inserted code must maintain state (registers!)
- *Disassemble-and-Resynthesise* (e.g., `valgrind`, `qemu`):
  - Decomposes program into IR
  - Instrumentation on IR-level
  - Easier/faster to track shadow values in some cases
    -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



```
short x;
x |= 0x7;
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**