Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
Dynamic Analysis

• Sometimes we want to study or adapt the behavior of *executions* of a program
  – *Did* my program ever ...?
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever ...?
  - Why/how did ... happen?
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - *Did* my program ever ...?
  - *Why/how did* ... happen?
  - *Where* am I spending time?
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - *Did* my program ever ...?
  - *Why/how did* ... happen?
  - *Where* am I spending time?
  - *Where* might I parallelize?
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - *Did* my program ever ...?
  - *Why/how did* ... happen?
  - *Where* am I spending time?
  - *Where* might I parallelize?
  - *Tolerate* errors.
Dynamic Analysis

- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever ...?
  - Why/how did ... happen?
  - Where am I spending time?
  - Where might I parallelize?
  - Tolerate errors.
  - Manage memory / resources.
e.g. Reverse Engineering

Static CFG (from e.g. Apple Fairplay):

This is the result of a control flow flattening obfuscation.
[http://tigress.cs.arizona.edu/transformPage/docs/flatten/]
e.g. Reverse Engineering

Static CFG (from e.g. Apple Fairplay):

Dynamically Simplified CFG:
How?

- Can record the execution

![Diagram showing input and program]
How?

- Can record the execution
How?

- Can record the execution

Input → Program → Trace → Analysis
How?

- Can record the execution
How?

- Can record the execution
  - Record to a trace
  - Analyze *post mortem / offline*
  - Scalability issues: *need enough space* to store it
How?

• Can record the execution
  – Record to a trace
  – Analyze post mortem / offline
  – Scalability issues: need enough space to store it

• Can perform analysis online
How?

- Can record the execution
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- Can perform analysis online
  - *Instrument* the program
  - Modified program invokes code to 'analyze' itself
How?

- Can record the execution
  - Record to a trace
  - Analyze post mortem / offline
  - Scalability issues: need enough space to store it

- Can perform analysis online
  - *Instrument* the program
  - Modified program invokes code to 'analyze' itself

- Can do both
  - Lightweight recording
  - Instrument a replayed instance of the execution
How?

• Can record the execution
  – Record to a trace
  – Analyze post mortem / offline
  – Scalability issues: need enough space to store it

• Can perform analysis online
  – *Instrument* the program
  – Modified program invokes code to 'analyze' itself

• Can do both
  – Lightweight recording
  – Instrument a replayed instance of the execution

Some analyses only make sense *online*. Why?
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** Which *basic blocks* execute most frequently?
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal**: Which *basic blocks* execute most frequently?
- How can we modify our program to find this?
Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

- **Goal:** Which basic blocks execute most frequently?
- How can we modify our program to find this?

Start: 
```python
for i in BBs:
    count[i] = 0
```

End: 
```python
for i in BBs:
    print(count[i])
```

```python
BB:0
BB:1
BB:2
BB:3

count[2] += 1
x = foo()
y = bar()
...
```
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

```plaintext
count[0] += 1
...
count[1] += 1
...
count[2] += 1
...
count[3] += 1
...
```
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool
  - Can we do better?
Simple Idea: Basic Block Profiling

• Big concern: How efficient is it?
  – The more overhead added, the less practical the tool

```plaintext
count[0] += 1
...

count[1] += 1
...

count[2] += 1
...

count[3] += 1
...
```

– Can we do better?

```
count[0] = count[3]
```
Simple Idea: Basic Block Profiling

- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool

```
count[0] += 1
...

count[1] += 1
...

count[2] += 1
...

count[3] += 1
...
```

- Can we do better?

```
count[0] = count[3]
```

Is it possible to do even better?
Efficiency Tactics

- Abstraction
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
- Profile guided instrumentation
Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding
- Profile guided instrumentation
- Thread local analysis & inference
Goal: How often does an acyclic path execute?
Path Profiling

- **Goal**: How often does an acyclic path execute?
  - Could log the trace...

```
log(A)
log(B)
log(D)
log(C)
log(E)
log(D)
recordPath()
```
Path Profiling

**Goal:** How often does an acyclic path execute?
- Could log the trace...
- Could *encode the paths*

<table>
<thead>
<tr>
<th>Path</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABDE   F</td>
<td>0</td>
</tr>
<tr>
<td>ABDF</td>
<td>1</td>
</tr>
<tr>
<td>ABCDE   F</td>
<td>2</td>
</tr>
<tr>
<td>ABCDF</td>
<td>3</td>
</tr>
<tr>
<td>ACDE   F</td>
<td>4</td>
</tr>
<tr>
<td>ACDF</td>
<td>5</td>
</tr>
</tbody>
</table>

![Diagram of path enumeration and counting](image-url)
Path Profiling

- Step 1: Count the # of paths *from* each node to the exit
Path Profiling

- Step 1: Count the number of paths from each node to the exit
Path Profiling

- Step 1: Count the number of paths from each node to the exit
Path Profiling

- Step 1: Count the # of paths *from* each node to the exit

![Diagram of a network with nodes A, B, C, D, E, and F, showing the number of paths from each node to the exit: 2 paths from A, 2 paths from D, 1 path from E, and 1 path from F.](image)
Path Profiling

- Step 1: Count the # of paths from each node to the exit
Path Profiling

• Step 1: Count the # of paths from each node to the exit
Path Profiling

- Step 2: Partition the encoding space locally at each node

```plaintext
id = 0
A  6

B  4

C  2

D  2

E  1

F  1

count[id] += 1
```
Path Profiling

- Step 2: Partition the encoding space locally at each node

```
A \ 6
B \ 4 \-> C \ 2
D \ 2  \leftarrow
E \ 1 \-> F \ 1
```

$\text{id=0}$

```
K1 = ?
```

```
count[id] += 1
```
Path Profiling

- Step 2: Partition the encoding space locally at each node

\[
\begin{align*}
\text{id} &= 0 \\
\text{count}[\text{id}] &+ = 1 \\
\end{align*}
\]
Path Profiling

- Step 2: Partition the encoding space locally at each node

\[ k1 = k2 + k3 + k4 \]
Path Profiling

• Step 2: Partition the encoding space locally at each node

\[
k_1 = k_2 + k_3 + k_4
\]
Path Profiling

- Step 2: Partition the encoding space locally at each node

\[ k_1 = k_2 + k_3 + k_4 \]

\[ k_1 = (k_2 + k_3) + k_2 \]

\[ k_1 = (k_2 + k_3) + k_2 \]

\[ k_1 = k_2 + k_3 + k_4 \]

\[ \text{id}=0 \]

\[ \text{count[id]} += 1 \]
Path Profiling

- Step 2: Partition the encoding space locally at each node

```
+0
n2→n1
+k2
+k2
n3→n1
+(k2+k3)
n4→n1

id=0
A 6
id+=4
B 4
C 2
D 2
E 1
F 1

count[id]++=1
```
**Path Profiling**

- Step 2: Partition the encoding space locally at each node

```
+0
+k2
+(k2+k3)
```

```
id=0
id+=2
id+=4
```

```
exit

count[id]++=1
```
Path Profiling

- Step 2: Partition the encoding space locally at each node

```
+0           +(k2+k3)
+ k2         + k2
n1            n1

k2 n2        k3 n3        k4 n4
```

```
id=0          id+=4
B 4           C 2
id+=2
D 2
id+=1
E 1           F 1

count[id]++=1
```
Path Profiling

- Step 2: Partition the encoding space locally at each node

![Diagram showing a graph with nodes and edges labeled with variables and operations. The nodes are A, B, C, D, E, and F. The operations include additions and assignments to variables id and count[id].]
Path Profiling: Decoding

How do we know which IDs map to which paths?

```
id+=1
```

```
id=0
A
```

```
id=2
D
```

```
id=4
B
```

```
id=2
C
```

```
count[id]++=1
```

```
id+=1
E
```

```
id+=1
F
```
Path Profiling: Decoding

How do we know which IDs map to which paths?

• Naive:
  - Keep a dictionary (large)

```c
id+=1
id=2
id=4
id=0
count[id]+=1
```
Path Profiling: Decoding

How do we know which IDs map to which paths?

- Naive:
  - Keep a dictionary (large)

Why could it be large?
Path Profiling: Decoding

How do we know which IDs map to which paths?

- **Naive:**
  - Keep a dictionary (**large**)
- **Better:**
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'

```
id=0
id=4
id=2
id+=1
count[id]++=1
```
Path Profiling: Decoding

How do we know which IDs map to which paths?

- **Naive:**
  - Keep a dictionary (*large*)

- **Better:**
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'

```python
id = 0
A

id = 1
B

id = 2
C

id = 3
E

id += 1
D

count[id] += 1

id = 4
F

id = 6
```
Path Profiling: Decoding

How do we know which IDs map to which paths?

- **Naive:**
  - Keep a dictionary (large)

- **Better:**
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'

```plaintext
id = 3
id = 3
id = 3
id = 0
id = 4
id = 2
id = 2
id += 1

count[id] += 1
```
Path Profiling: Decoding

How do we know which IDs map to which paths?

- Naive:
  - Keep a dictionary (*large*)

- Better:
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'

```plaintext
id+=1
id=2
id=3
id=3
-2
id=2
id=4
id=0
id=3
id 3
c:2
d:2
e:1
f:1
count[id]++=1
```
Path Profiling: Decoding

How do we know which IDs map to which paths?

- **Naive:**
  - Keep a dictionary *(large)*
- **Better:**
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'

```
B 4
id=2
\rightarrow\quad id=3
D 2
id=2
\rightarrow\quad id=3
C 2
id=0
\rightarrow\quad id=4
A 6
```

```
id+=1
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qu
Path Profiling: Decoding

How do we know which IDs map to which paths?

- **Naive:**
  - Keep a dictionary (*large*)

- **Better:**
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'

```
id+=1
id=2
id=4
id=3
id=0
id=1
id=1
id=2
id=3
id=3
-2
-1

id+=1
```

```
count[id]++
```
## Path Profiling: Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base Time (sec)</th>
<th>PP Overhead %</th>
<th>QPT2 Overhead %</th>
<th>PP/QPT</th>
<th>Path Inc (million)</th>
<th>Edge Inc (x Path)</th>
<th>Hashed Inc %</th>
<th>Inst/Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>885.0</td>
<td>53.4</td>
<td>24.1</td>
<td>2.2</td>
<td>1002.4</td>
<td>1.5</td>
<td>27.7</td>
<td>33.2</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>571.0</td>
<td>35.6</td>
<td>18.7</td>
<td>1.9</td>
<td>4824.9</td>
<td>1.2</td>
<td>3.9</td>
<td>16.2</td>
</tr>
<tr>
<td>126.gec</td>
<td>322.0</td>
<td>96.9</td>
<td>52.8</td>
<td>1.8</td>
<td>9.4</td>
<td>1.7</td>
<td>16.8</td>
<td>15.1</td>
</tr>
<tr>
<td>129.compress</td>
<td>351.0</td>
<td>19.4</td>
<td>21.9</td>
<td>0.9</td>
<td>3015.7</td>
<td>1.5</td>
<td>0.0</td>
<td>16.6</td>
</tr>
<tr>
<td>130.li</td>
<td>480.0</td>
<td>25.4</td>
<td>26.7</td>
<td>1.0</td>
<td>3282.4</td>
<td>1.4</td>
<td>1.2</td>
<td>16.8</td>
</tr>
<tr>
<td>132.jpeg</td>
<td>749.0</td>
<td>17.4</td>
<td>16.3</td>
<td>1.1</td>
<td>1164.9</td>
<td>1.1</td>
<td>1.2</td>
<td>31.0</td>
</tr>
<tr>
<td>134.perl</td>
<td>332.0</td>
<td>72.9</td>
<td>51.5</td>
<td>1.4</td>
<td>1133.0</td>
<td>1.9</td>
<td>23.4</td>
<td>22.2</td>
</tr>
<tr>
<td>147.vortex</td>
<td>684.0</td>
<td>37.7</td>
<td>34.1</td>
<td>1.1</td>
<td>3576.3</td>
<td>1.5</td>
<td>23.7</td>
<td>20.3</td>
</tr>
<tr>
<td><strong>CINT95 Avg:</strong></td>
<td><strong>44.8</strong></td>
<td><strong>30.8</strong></td>
<td><strong>1.4</strong></td>
<td><strong>22251.1</strong></td>
<td>1.5</td>
<td>12.2</td>
<td>21.4</td>
<td></td>
</tr>
<tr>
<td>101.tomcatv</td>
<td>503.0</td>
<td>19.9</td>
<td>2.8</td>
<td>7.1</td>
<td>574.6</td>
<td>1.1</td>
<td>95.8</td>
<td>93.0</td>
</tr>
<tr>
<td>102.swim</td>
<td>691.0</td>
<td>8.4</td>
<td>0.6</td>
<td>14.5</td>
<td>163.4</td>
<td>1.0</td>
<td>0.2</td>
<td>162.9</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>465.0</td>
<td>10.1</td>
<td>5.8</td>
<td>1.7</td>
<td>558.1</td>
<td>1.2</td>
<td>21.5</td>
<td>92.8</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>811.0</td>
<td>37.7</td>
<td>5.8</td>
<td>6.5</td>
<td>1690.7</td>
<td>1.7</td>
<td>77.8</td>
<td>43.1</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>872.0</td>
<td>6.3</td>
<td>3.2</td>
<td>2.0</td>
<td>1035.2</td>
<td>1.0</td>
<td>7.7</td>
<td>133.5</td>
</tr>
<tr>
<td>110.aplpu</td>
<td>715.0</td>
<td>71.0</td>
<td>12.0</td>
<td>5.9</td>
<td>2111.4</td>
<td>1.1</td>
<td>99.1</td>
<td>44.8</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>1066.0</td>
<td>5.5</td>
<td>7.4</td>
<td>0.7</td>
<td>2952.8</td>
<td>1.1</td>
<td>0.0</td>
<td>56.5</td>
</tr>
<tr>
<td>141.apsi</td>
<td>492.0</td>
<td>7.7</td>
<td>1.8</td>
<td>4.2</td>
<td>599.3</td>
<td>1.1</td>
<td>3.5</td>
<td>84.0</td>
</tr>
<tr>
<td>145.fpppp</td>
<td>1927.0</td>
<td>14.6</td>
<td>-2.6</td>
<td>-5.6</td>
<td>395.0</td>
<td>1.8</td>
<td>42.5</td>
<td>636.0</td>
</tr>
<tr>
<td>146.wave5</td>
<td>620.0</td>
<td>16.9</td>
<td>6.1</td>
<td>2.8</td>
<td>737.3</td>
<td>1.3</td>
<td>65.0</td>
<td>74.1</td>
</tr>
<tr>
<td><strong>CFP95 Avg:</strong></td>
<td><strong>19.8</strong></td>
<td><strong>4.3</strong></td>
<td><strong>4.0</strong></td>
<td><strong>1081.8</strong></td>
<td>1.2</td>
<td>41.3</td>
<td>142.1</td>
<td></td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td><strong>30.9</strong></td>
<td><strong>16.1</strong></td>
<td><strong>2.8</strong></td>
<td><strong>1601.5</strong></td>
<td>1.3</td>
<td>28.4</td>
<td>88.4</td>
<td></td>
</tr>
</tbody>
</table>
### Path Profiling: Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base Time (sec)</th>
<th>PP Overhead %</th>
<th>QPT2 Overhead %</th>
<th>PP/QPT</th>
<th>Path Inc (million)</th>
<th>Edge Inc (x Path)</th>
<th>Hashed Inc %</th>
<th>Inst/Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>885.0</td>
<td>53.4</td>
<td>24.1</td>
<td>2.2</td>
<td>1002.4</td>
<td>1.5</td>
<td>27.7</td>
<td>33.2</td>
</tr>
<tr>
<td>124.m88kissim</td>
<td>571.0</td>
<td>35.6</td>
<td>18.7</td>
<td>1.9</td>
<td>4824.9</td>
<td>1.2</td>
<td>3.9</td>
<td>16.2</td>
</tr>
<tr>
<td>126.gec</td>
<td>322.0</td>
<td>96.9</td>
<td>52.8</td>
<td>1.8</td>
<td>9.4</td>
<td>1.7</td>
<td>16.8</td>
<td>15.1</td>
</tr>
<tr>
<td>129.compress</td>
<td>351.0</td>
<td>19.4</td>
<td>21.9</td>
<td>0.9</td>
<td>3015.7</td>
<td>1.5</td>
<td>0.0</td>
<td>16.6</td>
</tr>
<tr>
<td>130.li</td>
<td>480.0</td>
<td>25.4</td>
<td>26.7</td>
<td>1.0</td>
<td>3282.4</td>
<td>1.4</td>
<td>1.2</td>
<td>16.8</td>
</tr>
<tr>
<td>132.jpg</td>
<td>749.0</td>
<td>17.4</td>
<td>16.3</td>
<td>1.1</td>
<td>1164.9</td>
<td>1.1</td>
<td>1.2</td>
<td>31.0</td>
</tr>
<tr>
<td>134.perl</td>
<td>332.0</td>
<td>72.9</td>
<td>51.5</td>
<td>1.4</td>
<td>1133.0</td>
<td>1.9</td>
<td>23.4</td>
<td>22.2</td>
</tr>
<tr>
<td>147.vortex</td>
<td>684.0</td>
<td>37.7</td>
<td>34.1</td>
<td>1.1</td>
<td>3576.3</td>
<td>1.5</td>
<td>23.7</td>
<td>20.3</td>
</tr>
<tr>
<td><strong>CINT95 Avg:</strong></td>
<td><strong>44.8</strong></td>
<td><strong>30.8</strong></td>
<td><strong>22251.1</strong></td>
<td><strong>1.5</strong></td>
<td><strong>12.2</strong></td>
<td><strong>21.4</strong></td>
<td><strong>12.2</strong></td>
<td><strong>21.4</strong></td>
</tr>
<tr>
<td>101.tomcatv</td>
<td>503.0</td>
<td>19.9</td>
<td>2.8</td>
<td>7.1</td>
<td>574.6</td>
<td>1.1</td>
<td>95.8</td>
<td>93.0</td>
</tr>
<tr>
<td>102.swim</td>
<td>691.0</td>
<td>8.4</td>
<td>0.6</td>
<td>14.5</td>
<td>163.4</td>
<td>1.0</td>
<td>0.2</td>
<td>162.9</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>465.0</td>
<td>10.1</td>
<td>5.8</td>
<td>1.7</td>
<td>558.1</td>
<td>1.2</td>
<td>21.5</td>
<td>92.8</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>811.0</td>
<td>37.7</td>
<td>5.8</td>
<td>6.5</td>
<td>1690.7</td>
<td>1.7</td>
<td>77.8</td>
<td>43.1</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>872.0</td>
<td>6.3</td>
<td>3.2</td>
<td>2.0</td>
<td>1035.2</td>
<td>1.0</td>
<td>7.7</td>
<td>133.5</td>
</tr>
<tr>
<td>110.aplu</td>
<td>715.0</td>
<td>71.0</td>
<td>12.0</td>
<td>5.9</td>
<td>2111.4</td>
<td>1.1</td>
<td>99.1</td>
<td>44.8</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>1066.0</td>
<td>5.5</td>
<td>7.4</td>
<td>0.7</td>
<td>2952.8</td>
<td>1.1</td>
<td>0.0</td>
<td>56.5</td>
</tr>
<tr>
<td>141.apsi</td>
<td>492.0</td>
<td>7.7</td>
<td>1.8</td>
<td>4.2</td>
<td>599.3</td>
<td>1.1</td>
<td>3.5</td>
<td>84.0</td>
</tr>
<tr>
<td>145.fpppp</td>
<td>1927.0</td>
<td>14.6</td>
<td>-2.6</td>
<td>-5.6</td>
<td>395.0</td>
<td>1.8</td>
<td>42.5</td>
<td>636.0</td>
</tr>
<tr>
<td>146.wave5</td>
<td>620.0</td>
<td>16.9</td>
<td>6.1</td>
<td>2.8</td>
<td>737.3</td>
<td>1.3</td>
<td>65.0</td>
<td>74.1</td>
</tr>
<tr>
<td><strong>CFP95 Avg:</strong></td>
<td><strong>19.8</strong></td>
<td><strong>4.3</strong></td>
<td><strong>1081.8</strong></td>
<td><strong>1.2</strong></td>
<td><strong>41.3</strong></td>
<td><strong>142.1</strong></td>
<td><strong>12.2</strong></td>
<td><strong>21.4</strong></td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td><strong>30.9</strong></td>
<td><strong>16.1</strong></td>
<td><strong>1601.5</strong></td>
<td><strong>1.3</strong></td>
<td><strong>28.4</strong></td>
<td><strong>88.4</strong></td>
<td><strong>12.2</strong></td>
<td><strong>21.4</strong></td>
</tr>
</tbody>
</table>
## Path Profiling: Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base Time (sec)</th>
<th>PP Overhead %</th>
<th>QPT2 Overhead %</th>
<th>PP/QPT</th>
<th>Path Inc (million)</th>
<th>Edge Inc (x Path)</th>
<th>Hashed Inc %</th>
<th>Inst/Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>885.0</td>
<td>53.4</td>
<td>24.1</td>
<td>2.2</td>
<td>1002.4</td>
<td>1.5</td>
<td>27.7</td>
<td>33.2</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>571.0</td>
<td>35.6</td>
<td>18.7</td>
<td>1.9</td>
<td>4824.9</td>
<td>1.2</td>
<td>3.9</td>
<td>16.2</td>
</tr>
<tr>
<td>126.gec</td>
<td>322.0</td>
<td>96.9</td>
<td>52.8</td>
<td>1.8</td>
<td>9.4</td>
<td>1.7</td>
<td>16.8</td>
<td>15.1</td>
</tr>
<tr>
<td>129.compress</td>
<td>351.0</td>
<td>19.4</td>
<td>21.9</td>
<td>0.9</td>
<td>3015.7</td>
<td>1.5</td>
<td>0.0</td>
<td>16.6</td>
</tr>
<tr>
<td>130.li</td>
<td>480.0</td>
<td>25.4</td>
<td>26.7</td>
<td>1.0</td>
<td>3282.4</td>
<td>1.4</td>
<td>1.2</td>
<td>16.8</td>
</tr>
<tr>
<td>132.jpeg</td>
<td>749.0</td>
<td>17.4</td>
<td>16.3</td>
<td>1.1</td>
<td>1164.9</td>
<td>1.1</td>
<td>1.2</td>
<td>31.0</td>
</tr>
<tr>
<td>134.perl</td>
<td>332.0</td>
<td>72.9</td>
<td>51.5</td>
<td>1.4</td>
<td>1133.0</td>
<td>1.9</td>
<td>23.4</td>
<td>22.2</td>
</tr>
<tr>
<td>147.vortex</td>
<td>684.0</td>
<td>37.7</td>
<td>34.1</td>
<td>1.1</td>
<td>3576.3</td>
<td>1.5</td>
<td>23.7</td>
<td>20.3</td>
</tr>
<tr>
<td>CINT95 Avg:</td>
<td>44.8</td>
<td>30.8</td>
<td>1.4</td>
<td>22251.1</td>
<td>1.5</td>
<td>12.2</td>
<td>21.4</td>
<td></td>
</tr>
<tr>
<td>101.tomcatv</td>
<td>503.0</td>
<td>19.9</td>
<td>2.8</td>
<td>7.1</td>
<td>574.6</td>
<td>1.1</td>
<td>95.8</td>
<td>93.0</td>
</tr>
<tr>
<td>102.swim</td>
<td>691.0</td>
<td>8.4</td>
<td>0.6</td>
<td>14.5</td>
<td>163.4</td>
<td>1.0</td>
<td>0.2</td>
<td>162.9</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>465.0</td>
<td>10.1</td>
<td>5.8</td>
<td>1.7</td>
<td>558.1</td>
<td>1.2</td>
<td>21.5</td>
<td>92.8</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>811.0</td>
<td>37.7</td>
<td>5.8</td>
<td>6.5</td>
<td>1690.7</td>
<td>1.7</td>
<td>77.8</td>
<td>43.1</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>872.0</td>
<td>6.3</td>
<td>3.2</td>
<td>2.0</td>
<td>1035.2</td>
<td>1.0</td>
<td>7.7</td>
<td>133.5</td>
</tr>
<tr>
<td>110.aplu</td>
<td>715.0</td>
<td>71.0</td>
<td>12.0</td>
<td>5.9</td>
<td>2111.4</td>
<td>1.1</td>
<td>99.1</td>
<td>44.8</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>1066.0</td>
<td>5.5</td>
<td>7.4</td>
<td>0.7</td>
<td>2952.8</td>
<td>1.1</td>
<td>0.0</td>
<td>56.5</td>
</tr>
<tr>
<td>141.apsi</td>
<td>492.0</td>
<td>7.7</td>
<td>1.8</td>
<td>4.2</td>
<td>599.3</td>
<td>1.1</td>
<td>3.5</td>
<td>84.0</td>
</tr>
<tr>
<td>145.fpppp</td>
<td>1927.0</td>
<td>14.6</td>
<td>-2.6</td>
<td>-5.6</td>
<td>395.0</td>
<td>1.8</td>
<td>42.5</td>
<td>636.0</td>
</tr>
<tr>
<td>146.wave5</td>
<td>620.0</td>
<td>16.9</td>
<td>6.1</td>
<td>2.8</td>
<td>737.3</td>
<td>1.3</td>
<td>65.0</td>
<td>74.1</td>
</tr>
<tr>
<td>CFP95 Avg:</td>
<td>19.8</td>
<td>4.3</td>
<td>4.0</td>
<td>1081.8</td>
<td>1.2</td>
<td>41.3</td>
<td>142.1</td>
<td></td>
</tr>
<tr>
<td>Average:</td>
<td>30.9</td>
<td>16.1</td>
<td>2.8</td>
<td>1601.5</td>
<td>1.3</td>
<td>28.4</td>
<td>88.4</td>
<td></td>
</tr>
</tbody>
</table>
## Path Profiling: Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base Time (sec)</th>
<th>PP Overhead %</th>
<th>QPT2 Overhead %</th>
<th>PP/QPT</th>
<th>Path Inc (million)</th>
<th>Edge Inc (x Path)</th>
<th>Hashed Inc %</th>
<th>Inst/Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>885.0</td>
<td>53.4</td>
<td>24.1</td>
<td>2.2</td>
<td>1002.4</td>
<td>1.5</td>
<td>27.7</td>
<td>33.2</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>571.0</td>
<td>35.6</td>
<td>18.7</td>
<td>1.9</td>
<td>4824.9</td>
<td>1.2</td>
<td>3.9</td>
<td>16.2</td>
</tr>
<tr>
<td>126.gec</td>
<td>322.0</td>
<td>96.9</td>
<td>52.8</td>
<td>1.8</td>
<td>9.4</td>
<td>1.7</td>
<td>16.8</td>
<td>15.1</td>
</tr>
<tr>
<td>129.compress</td>
<td>351.0</td>
<td>19.4</td>
<td>21.9</td>
<td>0.9</td>
<td>3015.7</td>
<td>1.5</td>
<td>0.0</td>
<td>16.6</td>
</tr>
<tr>
<td>130.li</td>
<td>480.0</td>
<td>25.4</td>
<td>26.7</td>
<td>1.0</td>
<td>3282.4</td>
<td>1.4</td>
<td>1.2</td>
<td>16.8</td>
</tr>
<tr>
<td>132.jpeg</td>
<td>749.0</td>
<td>17.4</td>
<td>16.3</td>
<td>1.1</td>
<td>1164.9</td>
<td>1.1</td>
<td>1.2</td>
<td>31.0</td>
</tr>
<tr>
<td>134.perl</td>
<td>332.0</td>
<td>72.9</td>
<td>51.5</td>
<td>1.4</td>
<td>1133.0</td>
<td>1.9</td>
<td>23.4</td>
<td>22.2</td>
</tr>
<tr>
<td>147.vortex</td>
<td>684.0</td>
<td>37.7</td>
<td>34.1</td>
<td>1.1</td>
<td>3576.3</td>
<td>1.5</td>
<td>23.7</td>
<td>20.3</td>
</tr>
<tr>
<td><strong>CINT95 Avg:</strong></td>
<td><strong>44.8</strong></td>
<td><strong>30.8</strong></td>
<td><strong>1.4</strong></td>
<td></td>
<td><strong>22251.1</strong></td>
<td><strong>1.5</strong></td>
<td><strong>12.2</strong></td>
<td><strong>21.4</strong></td>
</tr>
<tr>
<td>101.tomcatv</td>
<td>503.0</td>
<td>19.9</td>
<td>2.8</td>
<td>7.1</td>
<td>574.6</td>
<td>1.1</td>
<td>95.8</td>
<td>93.0</td>
</tr>
<tr>
<td>102.swim</td>
<td>691.0</td>
<td>8.4</td>
<td>0.6</td>
<td>14.5</td>
<td>163.4</td>
<td>1.0</td>
<td>0.2</td>
<td>162.9</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>465.0</td>
<td>10.1</td>
<td>5.8</td>
<td>1.7</td>
<td>558.1</td>
<td>1.2</td>
<td>21.5</td>
<td>92.8</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>811.0</td>
<td>37.7</td>
<td>5.8</td>
<td>6.5</td>
<td>1690.7</td>
<td>1.7</td>
<td>77.8</td>
<td>43.1</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>872.0</td>
<td>6.3</td>
<td>3.2</td>
<td>2.0</td>
<td>1035.2</td>
<td>1.0</td>
<td>7.7</td>
<td>133.5</td>
</tr>
<tr>
<td>110.applu</td>
<td>715.0</td>
<td>71.0</td>
<td>12.0</td>
<td>5.9</td>
<td>2111.4</td>
<td>1.1</td>
<td>99.1</td>
<td>44.8</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>1066.0</td>
<td>5.5</td>
<td>7.4</td>
<td>0.7</td>
<td>2952.8</td>
<td>1.1</td>
<td>0.0</td>
<td>56.5</td>
</tr>
<tr>
<td>141.apsi</td>
<td>492.0</td>
<td>7.7</td>
<td>1.8</td>
<td>4.2</td>
<td>599.3</td>
<td>1.1</td>
<td>3.5</td>
<td>84.0</td>
</tr>
<tr>
<td>145.fpppp</td>
<td>1927.0</td>
<td>14.6</td>
<td>-2.6</td>
<td>-5.6</td>
<td>395.0</td>
<td>1.8</td>
<td>42.5</td>
<td>636.0</td>
</tr>
<tr>
<td>146.wave5</td>
<td>620.0</td>
<td>16.9</td>
<td>6.1</td>
<td>2.8</td>
<td>737.3</td>
<td>1.3</td>
<td>65.0</td>
<td>74.1</td>
</tr>
<tr>
<td><strong>CFP95 Avg:</strong></td>
<td><strong>19.8</strong></td>
<td><strong>4.3</strong></td>
<td><strong>4.0</strong></td>
<td></td>
<td><strong>1081.8</strong></td>
<td><strong>1.2</strong></td>
<td><strong>41.3</strong></td>
<td><strong>142.1</strong></td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td><strong>30.9</strong></td>
<td><strong>16.1</strong></td>
<td><strong>2.8</strong></td>
<td></td>
<td><strong>1601.5</strong></td>
<td><strong>1.3</strong></td>
<td><strong>28.4</strong></td>
<td><strong>88.4</strong></td>
</tr>
</tbody>
</table>
### Path Profiling: Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base Time (sec)</th>
<th>PP Overhead %</th>
<th>QPT2 Overhead %</th>
<th>PP/QPT</th>
<th>Path Inc (million)</th>
<th>Edge Inc (x Path)</th>
<th>Hashed Inc %</th>
<th>Inst/Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>885.0</td>
<td>53.4</td>
<td>24.1</td>
<td>2.2</td>
<td>1002.4</td>
<td>1.5</td>
<td>27.7</td>
<td>33.2</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>571.0</td>
<td>35.6</td>
<td>18.7</td>
<td>1.9</td>
<td>4824.9</td>
<td>1.2</td>
<td>3.9</td>
<td>16.2</td>
</tr>
<tr>
<td>126.gec</td>
<td>322.0</td>
<td>96.9</td>
<td>52.8</td>
<td>1.8</td>
<td>9.4</td>
<td>1.7</td>
<td>16.8</td>
<td>15.1</td>
</tr>
<tr>
<td>129.compress</td>
<td>351.0</td>
<td>19.4</td>
<td>21.9</td>
<td>0.9</td>
<td>3015.7</td>
<td>1.5</td>
<td>0.0</td>
<td>16.6</td>
</tr>
<tr>
<td>130.li</td>
<td>480.0</td>
<td>25.4</td>
<td>26.7</td>
<td>1.0</td>
<td>3282.4</td>
<td>1.4</td>
<td>1.2</td>
<td>16.8</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>749.0</td>
<td>17.4</td>
<td>16.3</td>
<td>1.1</td>
<td>1164.9</td>
<td>1.1</td>
<td>1.2</td>
<td>31.0</td>
</tr>
<tr>
<td>134.perl</td>
<td>332.0</td>
<td>72.9</td>
<td>51.5</td>
<td>1.4</td>
<td>1133.0</td>
<td>1.9</td>
<td>23.4</td>
<td>22.2</td>
</tr>
<tr>
<td>147.vortex</td>
<td>684.0</td>
<td>37.7</td>
<td>34.1</td>
<td>1.1</td>
<td>3576.3</td>
<td>1.5</td>
<td>23.7</td>
<td>20.3</td>
</tr>
<tr>
<td><strong>CINT95 Avg:</strong></td>
<td><strong>44.8</strong></td>
<td><strong>30.8</strong></td>
<td><strong>1.4</strong></td>
<td></td>
<td><strong>22251.1</strong></td>
<td><strong>1.5</strong></td>
<td><strong>12.2</strong></td>
<td><strong>21.4</strong></td>
</tr>
<tr>
<td>101.tomcatv</td>
<td>503.0</td>
<td>19.9</td>
<td>2.8</td>
<td>7.1</td>
<td>574.6</td>
<td>1.1</td>
<td>95.8</td>
<td>93.0</td>
</tr>
<tr>
<td>102.swift</td>
<td>162.9</td>
<td>162.9</td>
<td>162.9</td>
<td>162.9</td>
<td>162.9</td>
<td>162.9</td>
<td>162.9</td>
<td>162.9</td>
</tr>
<tr>
<td>103.su27</td>
<td>92.8</td>
<td>92.8</td>
<td>92.8</td>
<td>92.8</td>
<td>92.8</td>
<td>92.8</td>
<td>92.8</td>
<td>92.8</td>
</tr>
<tr>
<td>104.hyde</td>
<td>43.1</td>
<td>43.1</td>
<td>43.1</td>
<td>43.1</td>
<td>43.1</td>
<td>43.1</td>
<td>43.1</td>
<td>43.1</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>872.0</td>
<td>6.3</td>
<td>3.2</td>
<td>2.0</td>
<td>1035.2</td>
<td>1.0</td>
<td>7.7</td>
<td>133.5</td>
</tr>
<tr>
<td>110.aplu</td>
<td>715.0</td>
<td>71.0</td>
<td>12.0</td>
<td>5.9</td>
<td>2111.4</td>
<td>1.1</td>
<td>99.1</td>
<td>44.8</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>1066.0</td>
<td>5.5</td>
<td>7.4</td>
<td>0.7</td>
<td>2952.8</td>
<td>1.1</td>
<td>0.0</td>
<td>56.5</td>
</tr>
<tr>
<td>141.apsi</td>
<td>492.0</td>
<td>7.7</td>
<td>1.8</td>
<td>4.2</td>
<td>599.3</td>
<td>1.1</td>
<td>3.5</td>
<td>84.0</td>
</tr>
<tr>
<td>145.fpppp</td>
<td>1927.0</td>
<td>14.6</td>
<td>-2.6</td>
<td>-5.6</td>
<td>395.0</td>
<td>1.8</td>
<td>42.5</td>
<td>636.0</td>
</tr>
<tr>
<td>146.wave5</td>
<td>620.0</td>
<td>16.9</td>
<td>6.1</td>
<td>2.8</td>
<td>737.3</td>
<td>1.3</td>
<td>65.0</td>
<td>74.1</td>
</tr>
<tr>
<td><strong>CFP95 Avg:</strong></td>
<td><strong>19.8</strong></td>
<td><strong>4.3</strong></td>
<td><strong>4.0</strong></td>
<td></td>
<td><strong>1081.8</strong></td>
<td><strong>1.2</strong></td>
<td><strong>41.3</strong></td>
<td><strong>142.1</strong></td>
</tr>
<tr>
<td><strong>Average:</strong></td>
<td><strong>30.9</strong></td>
<td><strong>16.1</strong></td>
<td><strong>2.8</strong></td>
<td></td>
<td><strong>1601.5</strong></td>
<td><strong>1.3</strong></td>
<td><strong>28.4</strong></td>
<td><strong>88.4</strong></td>
</tr>
</tbody>
</table>

What can/can't you infer from these results?
## Path Profiling: Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Base Time (sec)</th>
<th>PP Overhead %</th>
<th>QPT Overhead %</th>
<th>PP/QPT</th>
<th>Path Inc (million)</th>
<th>Edge Inc (x Path)</th>
<th>Hashed Inc %</th>
<th>Inst/Inc</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>885.0</td>
<td>53.4</td>
<td>24.1</td>
<td>2.2</td>
<td>1002.4</td>
<td>1.5</td>
<td>27.7</td>
<td>33.2</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>571.0</td>
<td>35.6</td>
<td>18.7</td>
<td>1.9</td>
<td>4824.9</td>
<td>1.2</td>
<td>3.9</td>
<td>16.2</td>
</tr>
<tr>
<td>126.gec</td>
<td>322.0</td>
<td>96.9</td>
<td>52.8</td>
<td>1.8</td>
<td>9.4</td>
<td>1.7</td>
<td>16.8</td>
<td>15.1</td>
</tr>
<tr>
<td>129.compress</td>
<td>351.0</td>
<td>19.4</td>
<td>21.9</td>
<td>0.9</td>
<td>3015.7</td>
<td>1.5</td>
<td>0.0</td>
<td>16.6</td>
</tr>
<tr>
<td>130.li</td>
<td>480.0</td>
<td>25.4</td>
<td>26.7</td>
<td>1.0</td>
<td>3282.4</td>
<td>1.4</td>
<td>1.2</td>
<td>16.8</td>
</tr>
<tr>
<td>132.jpg</td>
<td>749.0</td>
<td>17.4</td>
<td>16.3</td>
<td>1.1</td>
<td>1164.9</td>
<td>1.1</td>
<td>1.2</td>
<td>31.0</td>
</tr>
<tr>
<td>134.perl</td>
<td>332.0</td>
<td>72.9</td>
<td>51.5</td>
<td>1.4</td>
<td>1133.0</td>
<td>1.9</td>
<td>23.4</td>
<td>22.2</td>
</tr>
<tr>
<td>147.vortex</td>
<td>684.0</td>
<td>37.7</td>
<td>34.1</td>
<td>1.1</td>
<td>3576.3</td>
<td>1.5</td>
<td>23.7</td>
<td>20.3</td>
</tr>
<tr>
<td><strong>CINT95 Avg:</strong></td>
<td>44.8</td>
<td>30.8</td>
<td>1.4</td>
<td>22251.1</td>
<td>1.5</td>
<td>12.2</td>
<td>21.4</td>
<td></td>
</tr>
</tbody>
</table>

### What can/can't you infer from these results?

### What would you add or change to the evaluation?
Path Profiling

Are there cases where this approach fails?
Path Profiling

- What about loops / cycles?
Path Profiling

• What about loops / cycles?
  – Does the existing approach work?
Path Profiling

• What about loops / cycles?
  – Does the existing approach work?
  – How could we resolve it?
Path Profiling

• What about loops / cycles?
  – Does the existing approach work?
  – How could we resolve it?
Path Profiling

• What about loops / cycles?
  – Does the existing approach work?
  – How could we resolve it?
Path Profiling

• What about loops / cycles?
  – Does the existing approach work?
  – How could we resolve it?
Path Profiling

- What about loops / cycles?
  - Does the existing approach work?
  - How could we resolve it?
Path Profiling

- What about loops / cycles?
  - Does the existing approach work?
  - How could we resolve it?
Path Profiling

- What about loops / cycles?
  - Does the existing approach work?
  - How could we resolve it?

What do these edges encode?
Path Profiling

- Path profiling is a *dynamic* analysis
  - It analyzes an actual execution
Path Profiling

• Path profiling is a *dynamic* analysis
  – It analyzes an actual execution
  – “What were frequent paths for this input”
Path Profiling

• Path profiling is a *dynamic* analysis
  – It analyzes an actual execution
    – “What were frequent paths for this input”
    – “What were frequent paths for this set of inputs”
Path Profiling

• Path profiling is a *dynamic* analysis
  – It analyzes an actual execution
    – “What were frequent paths for this input”
    – “What were frequent paths for this set of inputs”

• What if you don’t have an input for the behavior you want to analyze?
Approximation

Modeled program behaviors

Possible Program Behavior
Approximation

Modeled program behaviors

Consider some behaviors possible when they are not.
Approximation

Modeled program behaviors

Overapproximate

Possible Program Behavior

Underapproximate

Ignore some behaviors that are possible.
Approximation

Modeled program behaviors

- Overapproximate
- Possible Program Behavior
- Underapproximate
- One Execution
Approximation

- Dynamic Analysis
  - Analyzed $\subseteq$ Feasible
Approximation

- Dynamic Analysis
  - Analyzed $\subseteq$ Feasible
Approximation

- Dynamic Analysis
  - Analyzed $\subseteq$ Feasible
  - As # tests $\uparrow$, Analyzed $\rightarrow$ Feasible
How / When to Instrument

- **Source / IR Instrumentation**
  - LLVM, CIL, Soot, Wala, ...
  - During (re)compilation
  - Requires an analysis dedicated build
How / When to Instrument

- **Source / IR Instrumentation**
  - LLVM, CIL, Soot, Wala, ...
  - During (re)compilation
  - Requires an analysis dedicated build

- **Static Binary Rewriting**
  - Uroboros, DynamoRIO, SecondWrite,
  - Applies to arbitrary binaries
  - Imprecise IR info, but more complete *binary* behavior
How / When to Instrument

• **Source / IR Instrumentation**
  – LLVM, CIL, Soot, Wala, ...
  – During (re)compilation
  – Requires an analysis dedicated build

• **Static Binary Rewriting**
  – Uroboros, DynamoRIO, SecondWrite,
  – Applies to arbitrary binaries
  – Imprecise IR info, but more complete binary behavior

• **Dynamic Binary Instrumentation**
  – Valgrind, Pin, Qemu (& other Vms)
  – Can adapt at runtime, but less info than IR
Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) **Instrumentation**
   - Add code to the program for data collection/analysis
Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) **Instrumentation**
   - Add code to the program for data collection/analysis

2) **Execution**
   - Run the program an analyze its actual behavior
Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) **Instrumentation**
   - Add code to the program for data collection/analysis

2) **Execution**
   - Run the program and analyze its actual behavior

3) (Optional) **Postmortem Analysis**
   - Perform any analysis that can be deferred after termination
Phases of Dynamic Analysis

In general, 2-3 phases occur:

1) **Instrumentation**
   - Add code to the program for data collection/analysis

2) **Execution**
   - Run the program and analyze its actual behavior

3) (Optional) **Postmortem Analysis**
   - Perform any analysis that can be deferred after termination

*Very, very* common mistake to mix 1 & 2.
Static Instrumentation

1) Compile whole program to IR
2) Instrument / add code directly to the IR
3) Generate new program that performs tracing/analysis
4) Execute
Dynamic Binary Instrumentation

1) **Compile** program as usual

2) **Run** program under analysis framework
   (Valgrind, PIN, Qemu, etc)
   - Instrument & execute in same command:
   - Fetch & instrument each basic block individually
   - Execute each basic block

```
valgrind --tool=memcheck ./myBuggyProgram
```
Example: Address Sanitizer

- **Address Sanitizer** is a built-in dynamic analysis component in the clang compiler
- Static instrumentation
Example: Address Sanitizer

- **Address Sanitizer** is a built-in dynamic analysis component in the clang compiler
- Static instrumentation
- Finds:
  - Use-after-free
  - \{heap, stack, global\}-buffer overflows
Example: Address Sanitizer

- **Address Sanitizer** is a built-in dynamic analysis component in the clang compiler
- Static instrumentation
- Finds:
  - Use-after-free
  - \{heap,stack,global\}-buffer overflows
- Used extensively in Google programs like Chrome
Example: Address Sanitizer

How?

• Replaces `malloc & free`
Example: Address Sanitizer

How?

- Replaces `malloc & free`
- Memory around allocated chunks is *poisoned*

```c
ptr = malloc(sizeof(MyStruct));
```

![Diagram showing memory allocation and poisoning]
Example: Address Sanitizer

How?

- Replaces `malloc & free`
- Memory around malloced chunks is *poisoned*
- *Freed* memory is *poisoned*

```c
free(ptr);
```
Example: Address Sanitizer

How?

- Replaces `malloc & free`
- Memory around malloced chunks is poisoned
- Freed memory is poisoned
- Space around buffers is poisoned

```c
void foo() {
    int buffer[5];
}
```
Example: Address Sanitizer

How?

- Replaces `malloc & free`
- Memory around malloced chunks is *poisoned*
- Freed memory is *poisoned*
- Space around buffers is *poisoned*
- Any access of a poisoned value reports an error.

...
Example: Address Sanitizer

How?

*address = ...

Instrumentation
Example: Address Sanitizer

How?

*address = ...

Instrumentation

if (IsPoisoned(address)) {
    ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
Example: Address Sanitizer

How?

```c
*address = ...
```

Instrumentation

```c
if (IsPoisoned(address)) {
    ReportError(address, kAccessSize, kIsWrite);
}
*address = ...;
```

Difficult! Why?

- Instrumenting every memory access is costly
- Tracking the status of all memory is tricky
Example: Address Sanitizer

Need to know whether *any byte* of application memory is poisoned.
Example: Address Sanitizer

• Maintain 2 views on memory:
Example: Address Sanitizer

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
Example: Address Sanitizer

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table

Where have you encountered this before? (Think OS)
Example: Address Sanitizer

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables
Example: Address Sanitizer

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables

- In Asan:
  - In an 8 byte chunk, only first k may be addressable

```
Memory: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Shadow: [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```
Example: Address Sanitizer

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables

- In Asan:
  - In an 8 byte chunk, only first k may be addressable
  - All 8 bytes unpoisoned: shadow value is 0.

Memory: 

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Shadow: 0
Example: Address Sanitizer

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables

- In Asan:
  - In an 8 byte chunk, only first k may be addressable
  - All 8 bytes unpoisoned: shadow value is 0.
  - All 8 bytes poisoned: shadow value is negative.

Example: Address Sanitizer

- Shadow memory is a pervasive dynamic analysis tool
  - For every bit/byte/word/chunk/allocation/page, maintain information in a compact table
  - Common in runtime support: e.g. page tables
- In Asan:
  - In an 8 byte chunk, only first k may be addressable
  - All 8 bytes unpoisoned: shadow value is 0.
  - All 8 bytes poisoned: shadow value is negative.
  - First k bytes are unpoisoned: shadow value is k.

Memory: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] Shadow: 5
Example: Address Sanitizer

- (64bit) Shadow Mapping:
  - Preallocate large block of memory
  - Shadow = (Mem >> 3) + 0x7fff8000;
Example: Address Sanitizer

- (64bit) Shadow Mapping:
  - Preallocate large block of memory
  - \( \text{Shadow} = (\text{Mem} \gg 3) + 0x7fff8000; \)

- The shadow memory itself must also be considered poisoned.

Why?!
Dynamic Analysis

- Analyze the actual/observed behaviors of a program.
Dynamic Analysis

- Analyze the actual/observed behaviors of a program.
- Modify the program's behavior in order to collect information.
Dynamic Analysis

- Analyze the actual/observed behaviors of a program.
- Modify the program's behavior in order to collect information.
- Analyze this information either online or offline.
Moving Forward

- Yet often you will want to deeply analyze a program without running it at all...