Dynamic Analysis

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

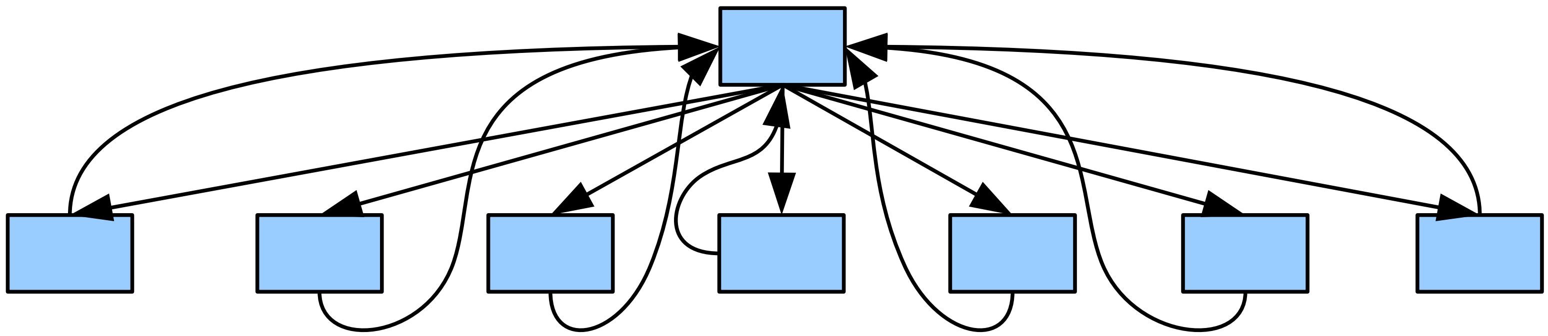
- Sometimes we want to study or adapt the behavior of *executions* of a program
  - Did my program ever …?
  - Why/how did … happen?
  - Where am I spending time?
  - Where might I parallelize?
  - Tolerate errors.

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):

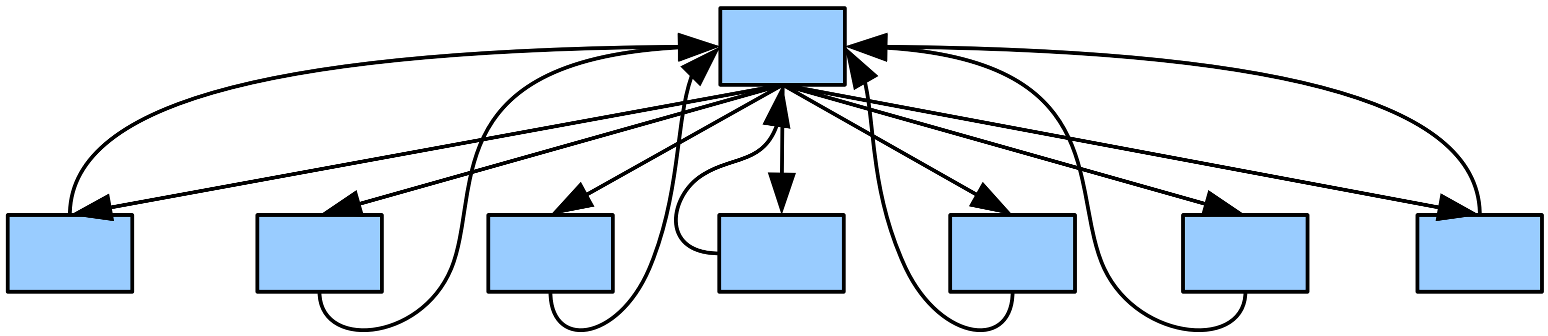


This is the result of a control flow flattening obfuscation.

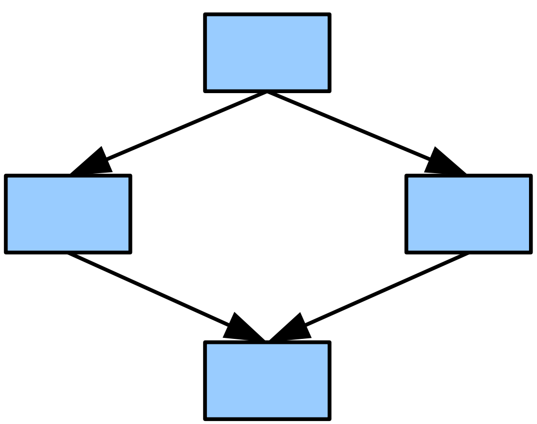
[http://tigress.cs.arizona.edu/transformPage/docs/flatten/]

e.g. Reverse Engineering

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

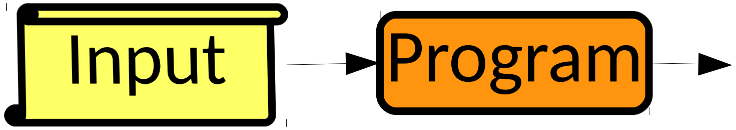


Dynamically Simplified CFG:



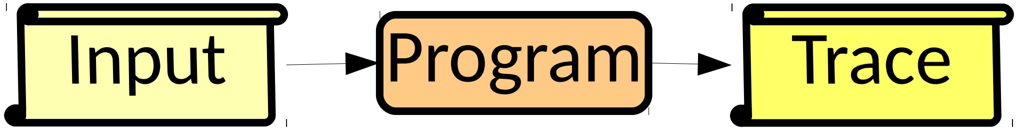
How?

- Can record the execution



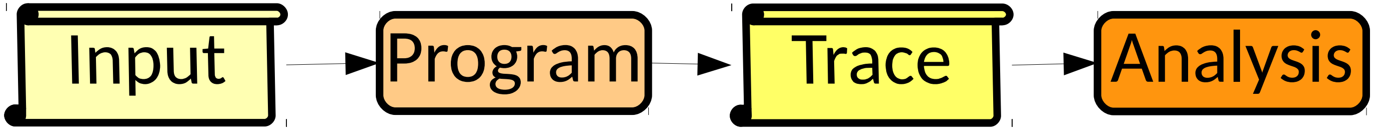
How?

- Can record the execution



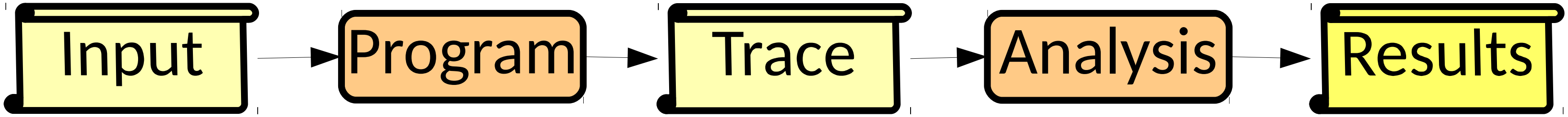
How?

- Can record the execution



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

Simple Idea: Basic Block Profiling

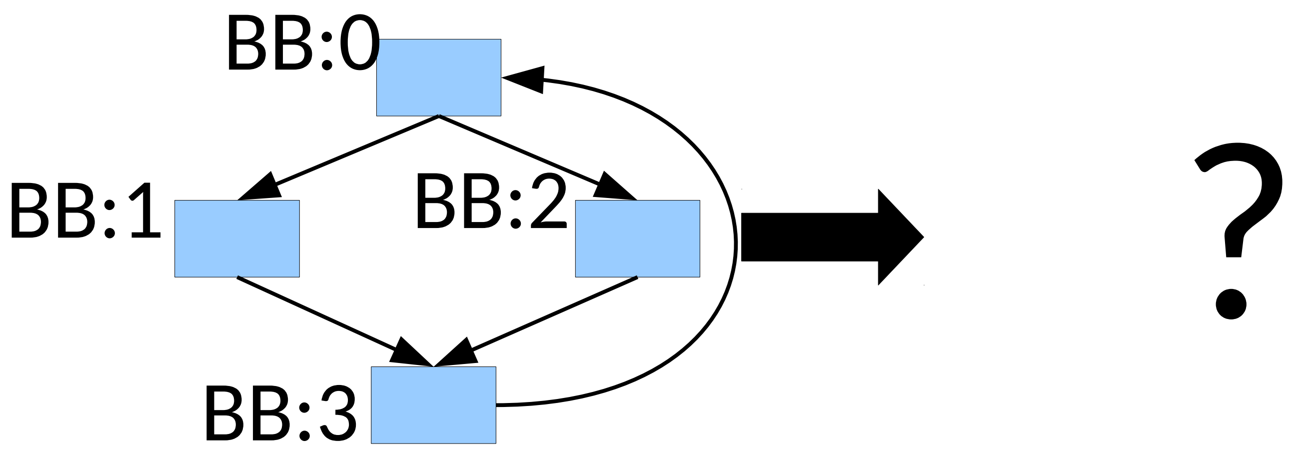
Knowing where we are spending time is useful:

- **Goal:** *Which basic blocks execute most frequently?*

Simple Idea: Basic Block Profiling

Knowing where we are spending time is useful:

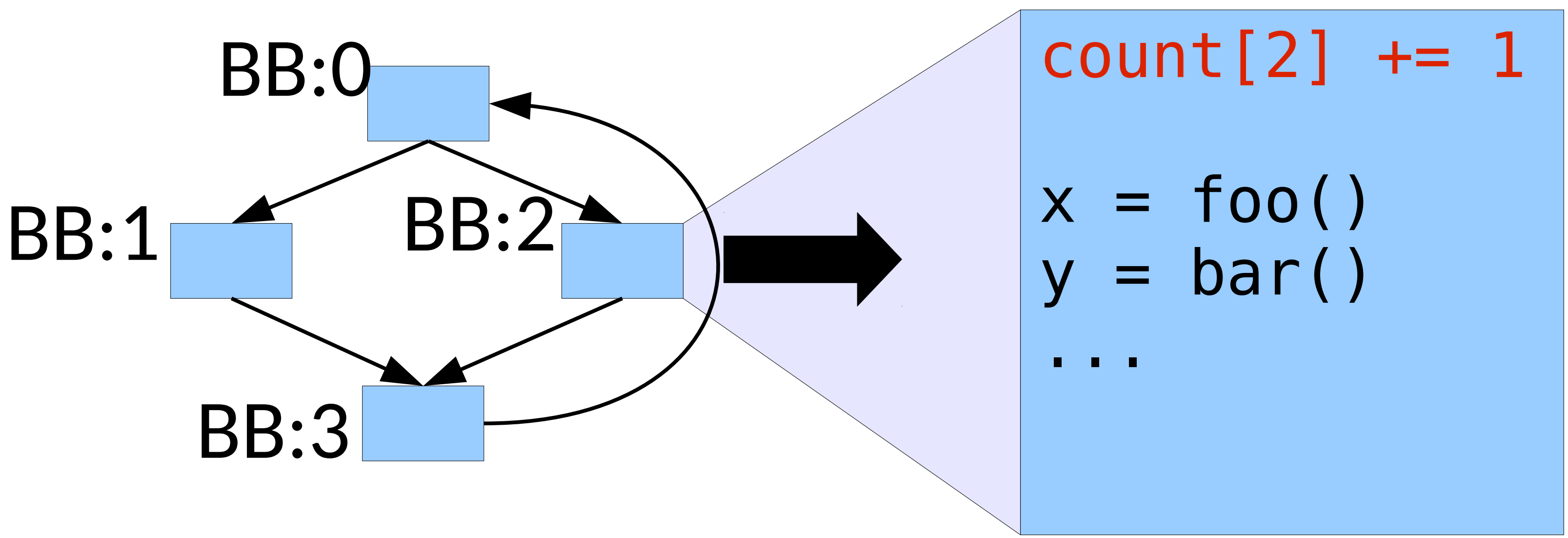
- **Goal:** *Which basic blocks execute most frequently?*
- How can we modify our program to find this?



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:

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

End:

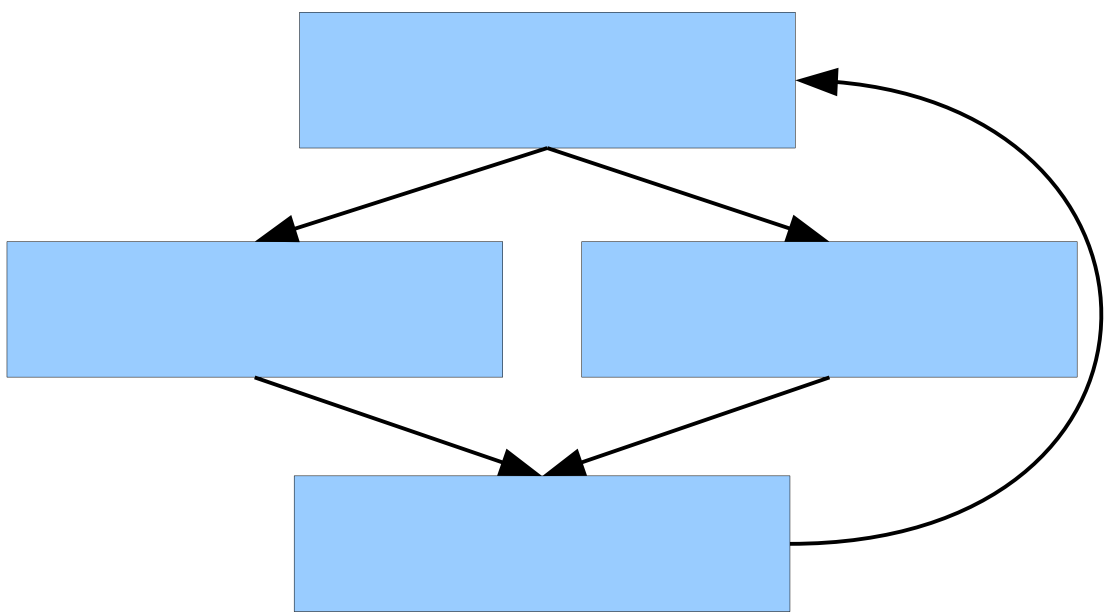
```
for i in BBs:
  print(count[i])
```

Simple Idea: Basic Block Profiling

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

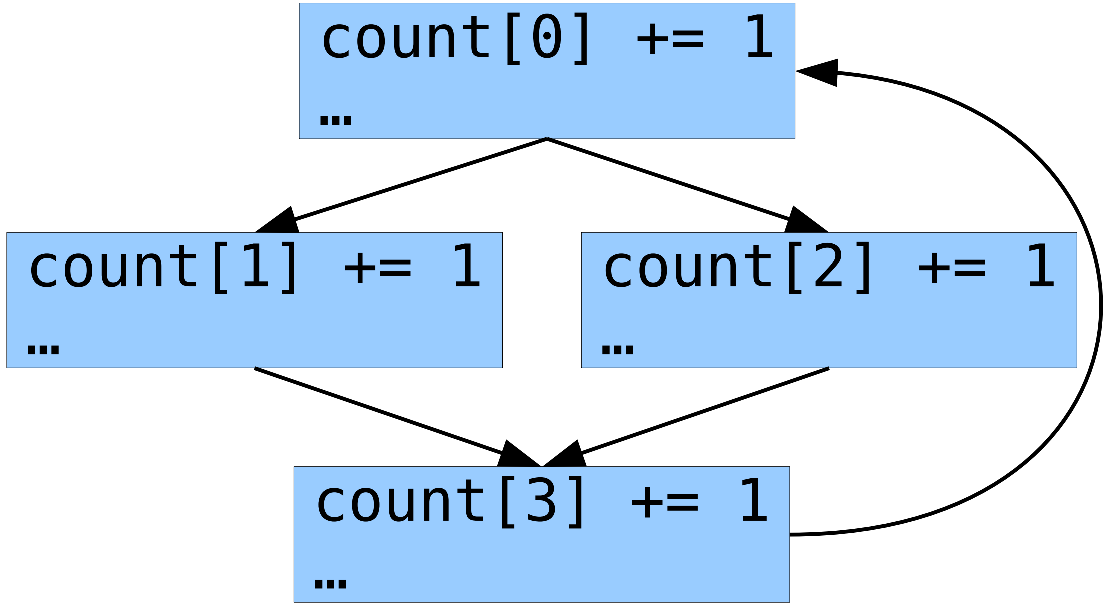
Simple Idea: Basic Block Profiling

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



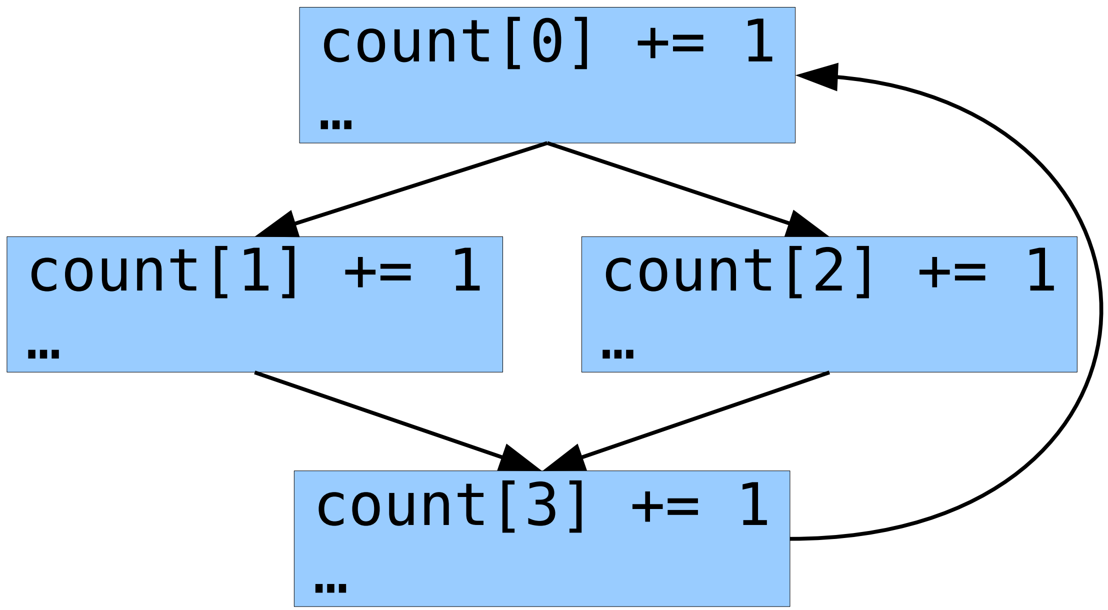
Simple Idea: Basic Block Profiling

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



Simple Idea: Basic Block Profiling

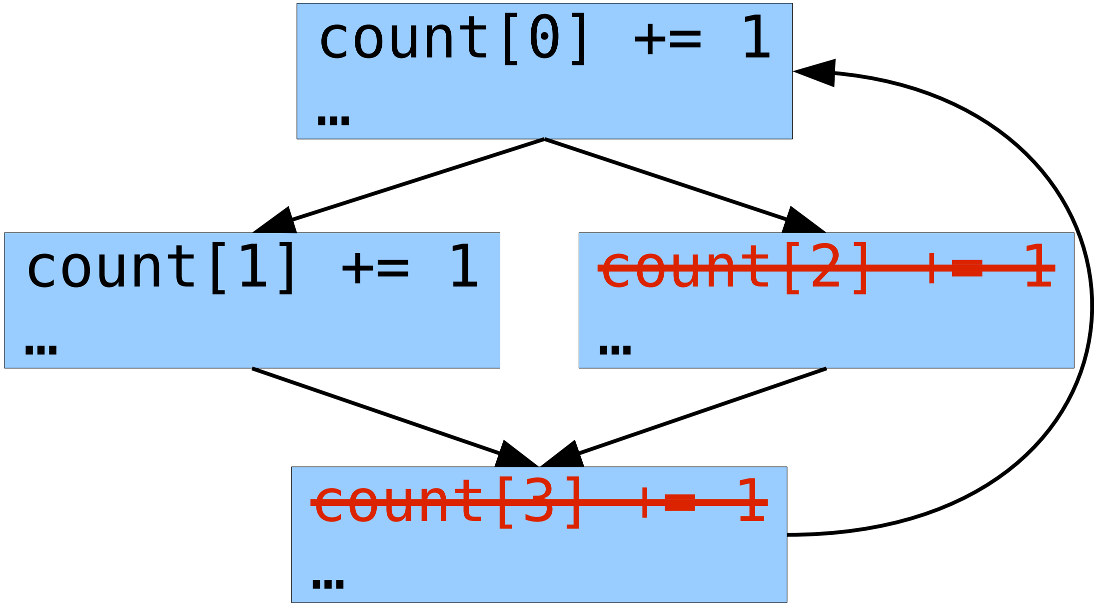
- Big concern: How efficient is it?
  - The more overhead added, the less practical the tool



  - Can we do better?

Simple Idea: Basic Block Profiling

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

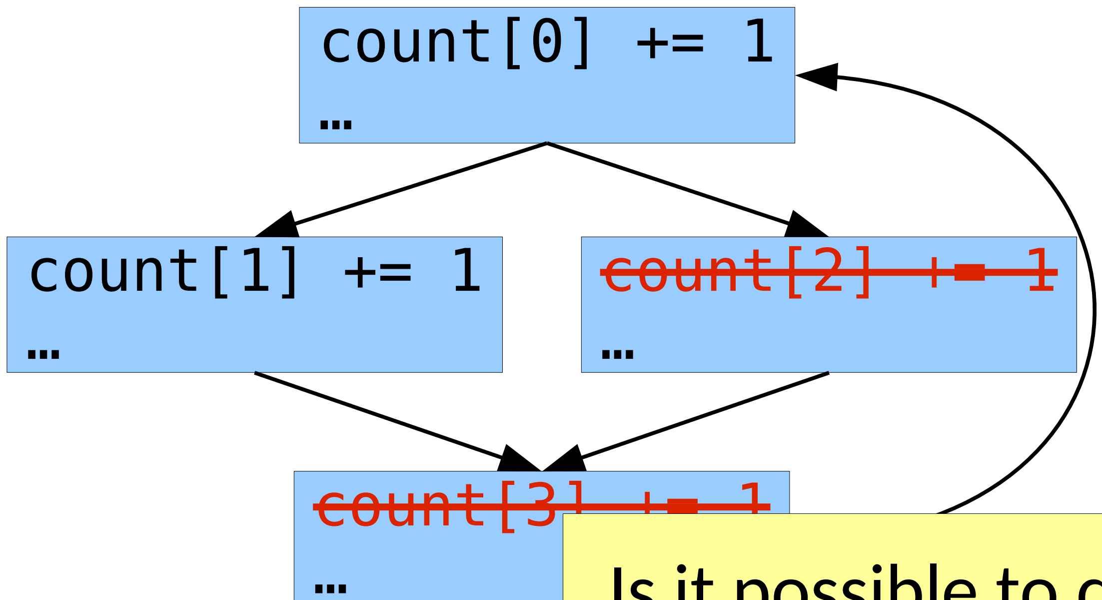


  - Can we do better?

```
count[0] = count[3]
count[2] = count[0] - count[1]
```

Simple Idea: Basic Block Profiling

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



Is it possible to do even better?

  - Can we do better?

```
count[0] = count[3]
count[2] = count[0] - count[1]
```

Efficiency Tactics

- Abstraction

Efficiency Tactics

- Abstraction
- Identify & avoid redundant information

Efficiency Tactics

- Abstraction
- Identify & avoid redundant information
- Sampling

Efficiency Tactics

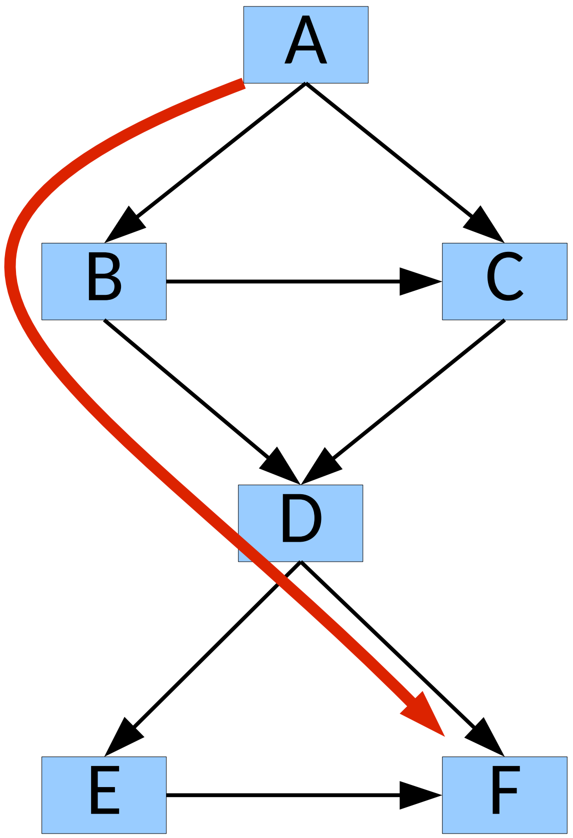
- Abstraction
- Identify & avoid redundant information
- Sampling
- Compression / encoding

Efficiency Tactics

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

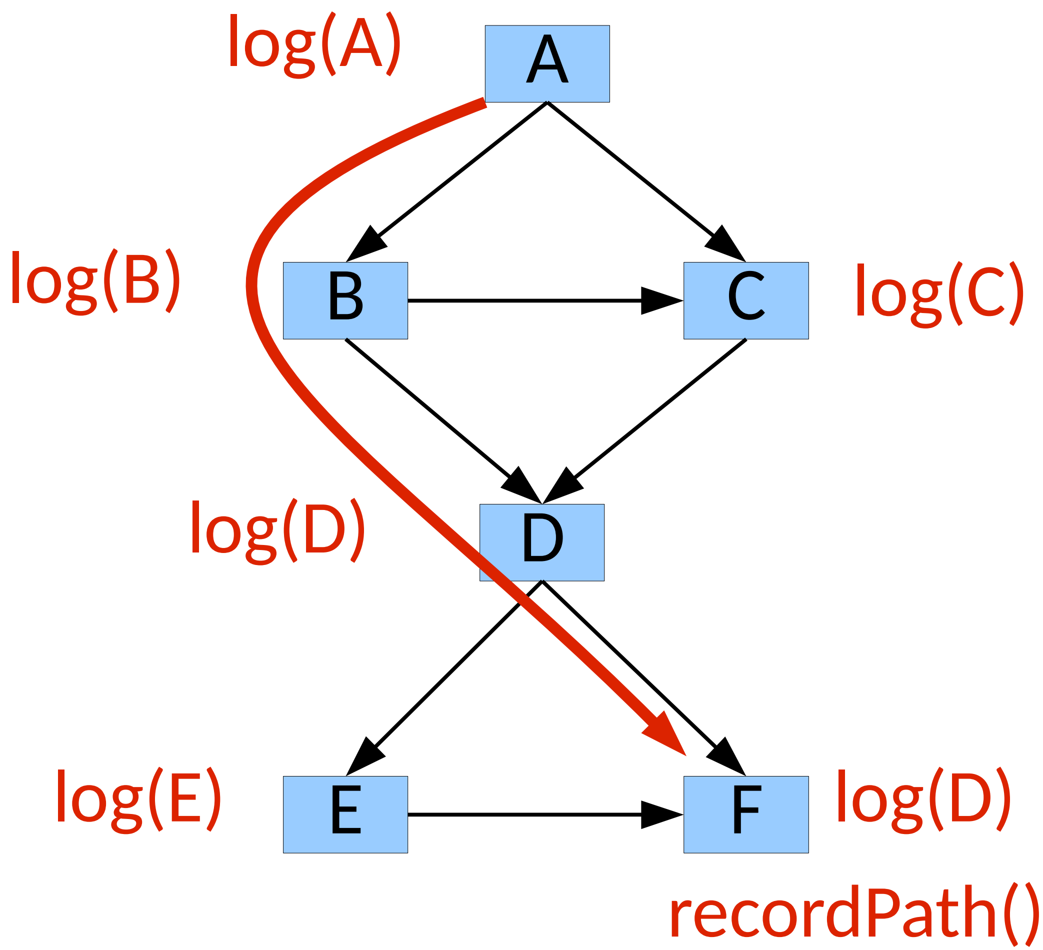
Path Profiling

- **Goal:** How often does an acyclic path execute?



Path Profiling

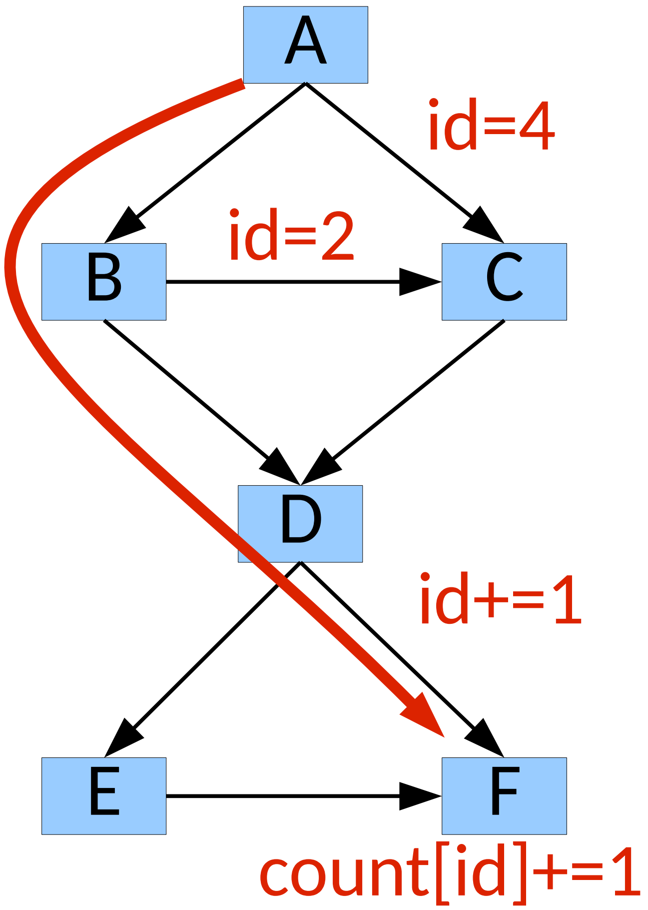
- **Goal:** How often does an acyclic path execute?
  - Could log the trace…



Path Profiling

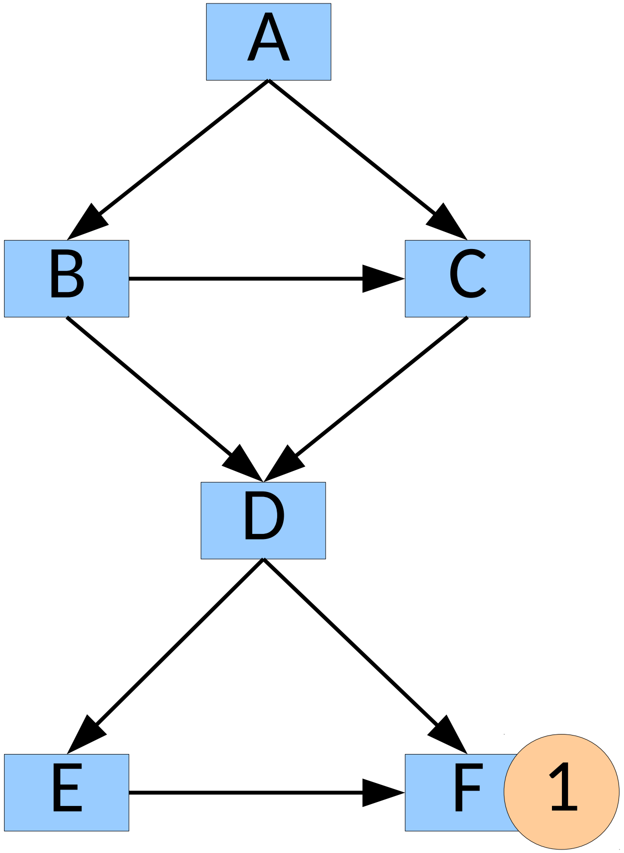
- **Goal:** How often does an acyclic path execute?
  - Could log the trace…
  - Could *encode the paths*

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



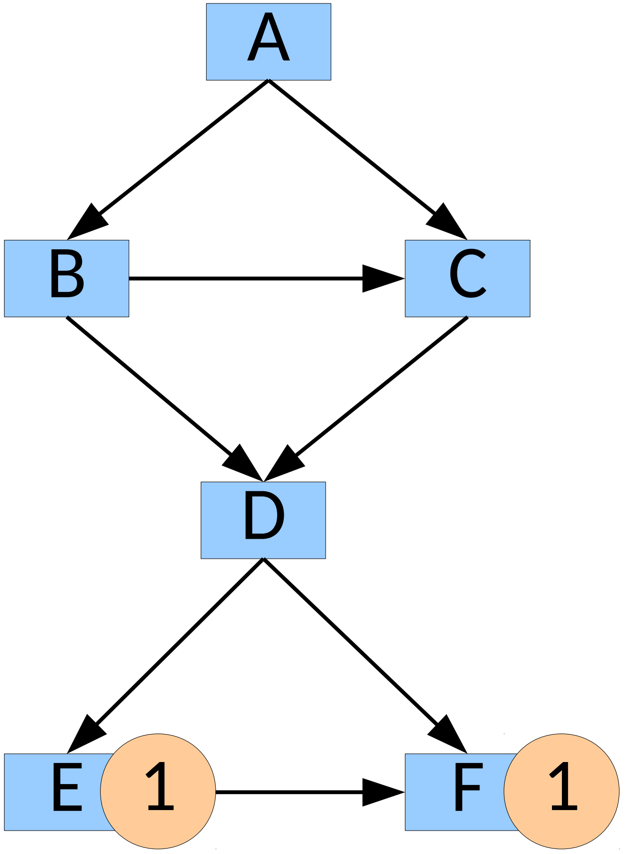
Path Profiling

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



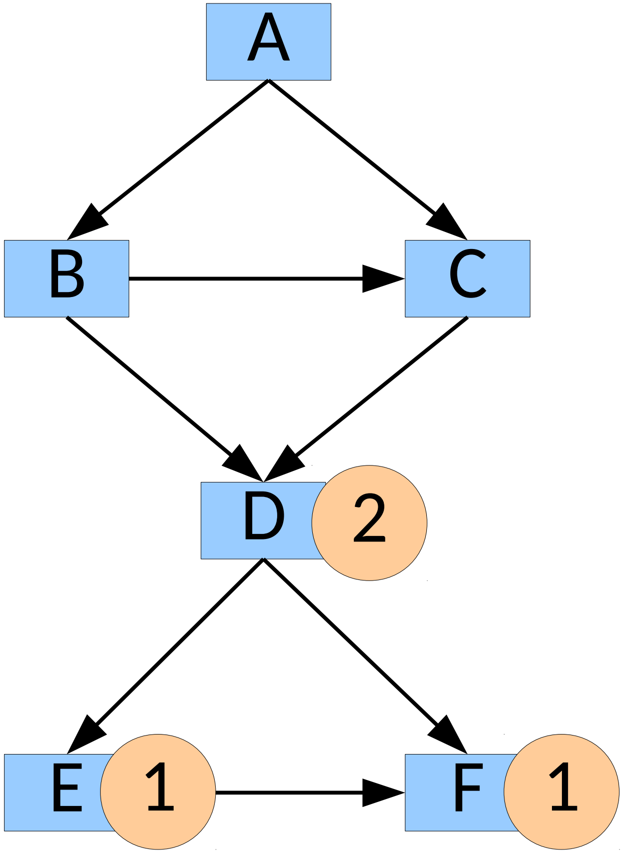
Path Profiling

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



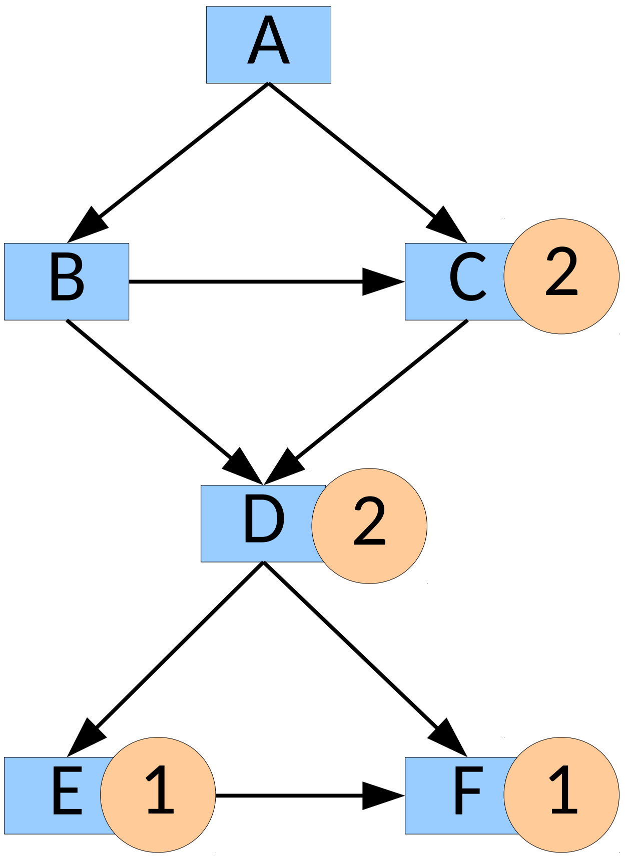
Path Profiling

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



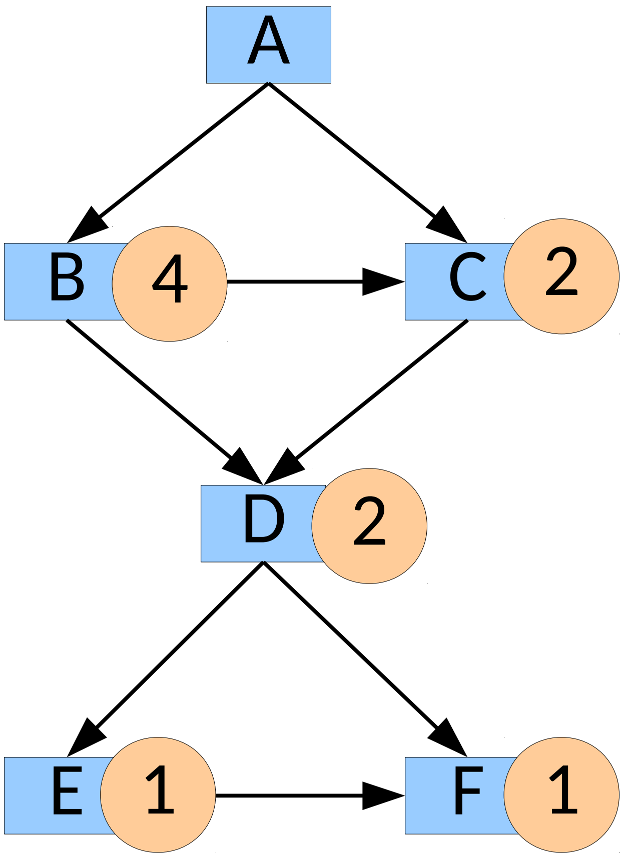
Path Profiling

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



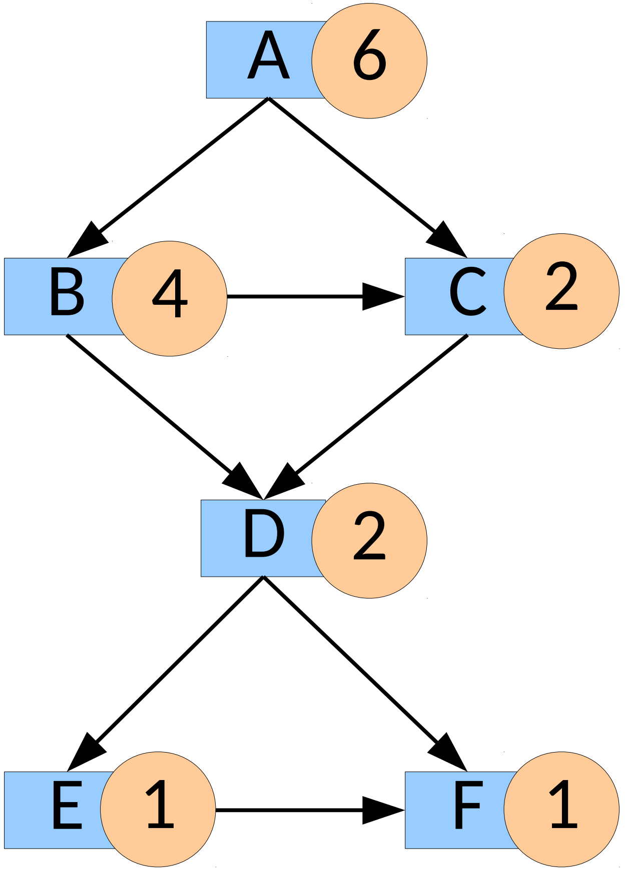
Path Profiling

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



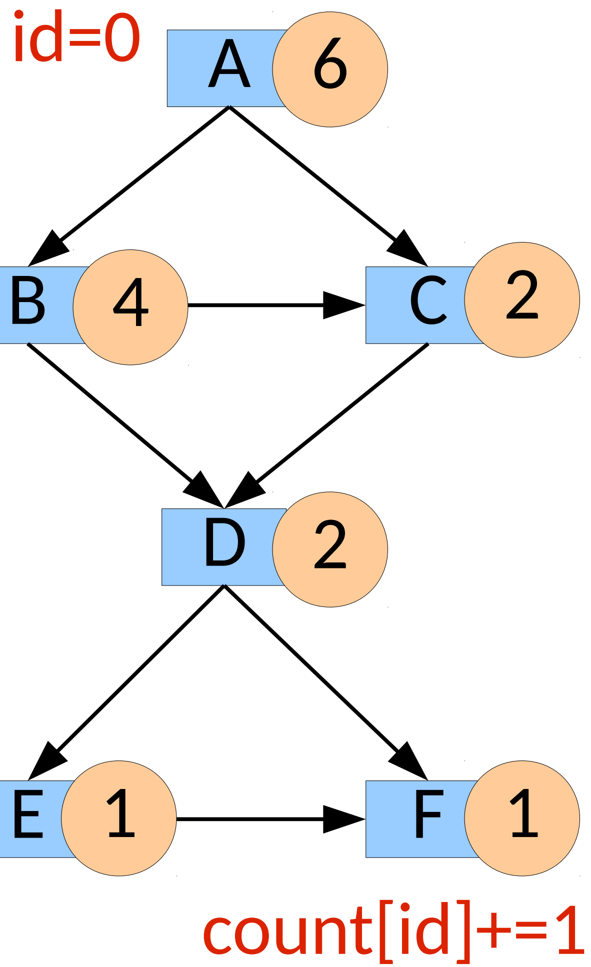
Path Profiling

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



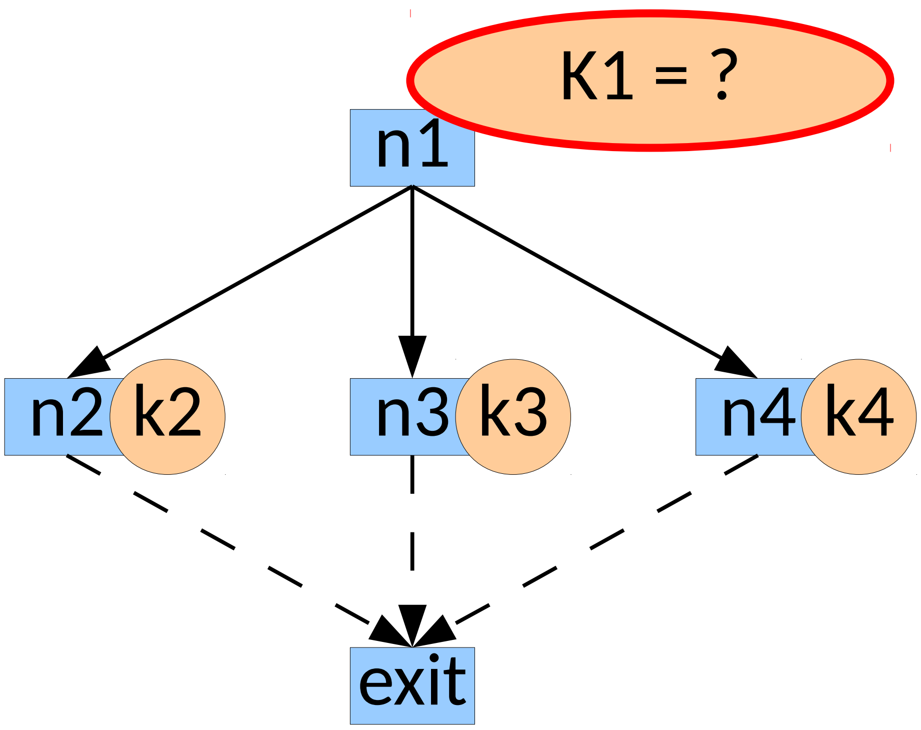
Path Profiling

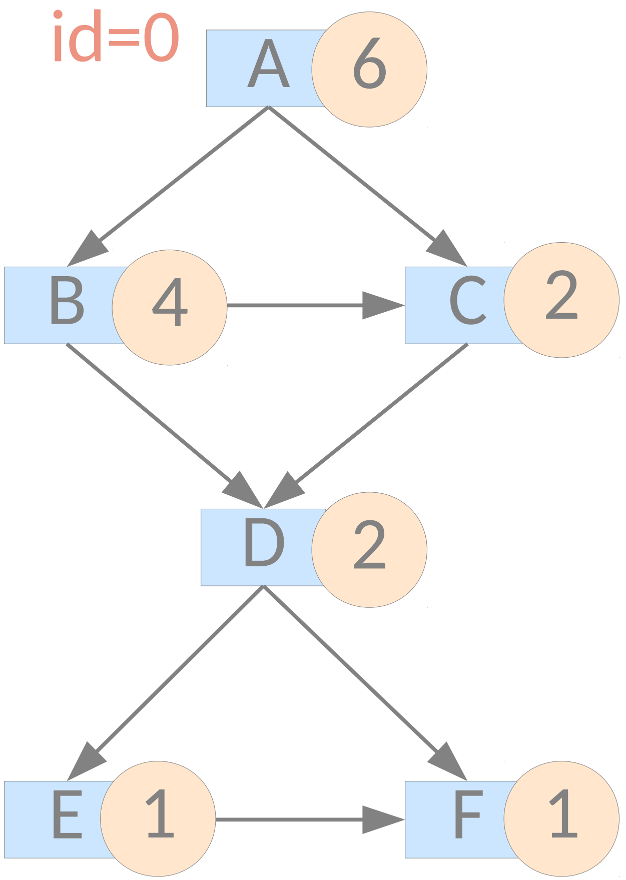
- Step 2: Partition the encoding space locally at each node



Path Profiling

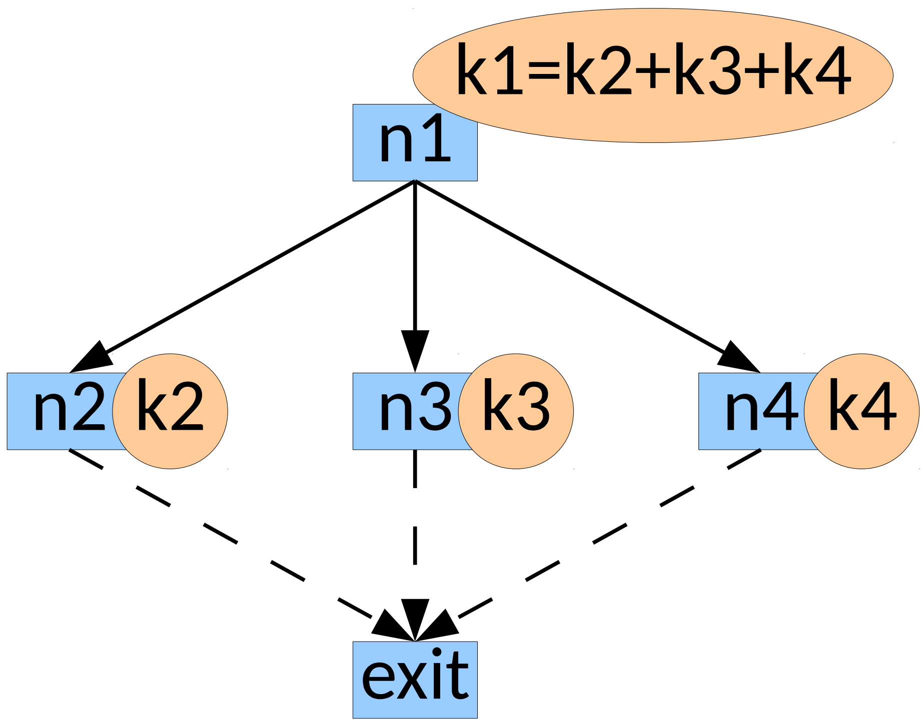
- Step 2: Partition the encoding space locally at each node

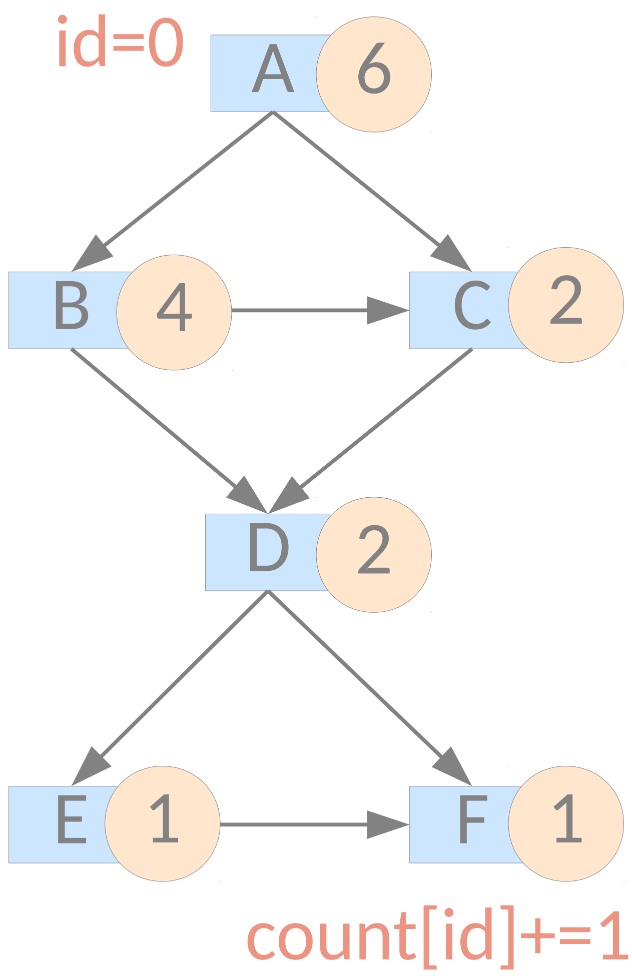




Path Profiling

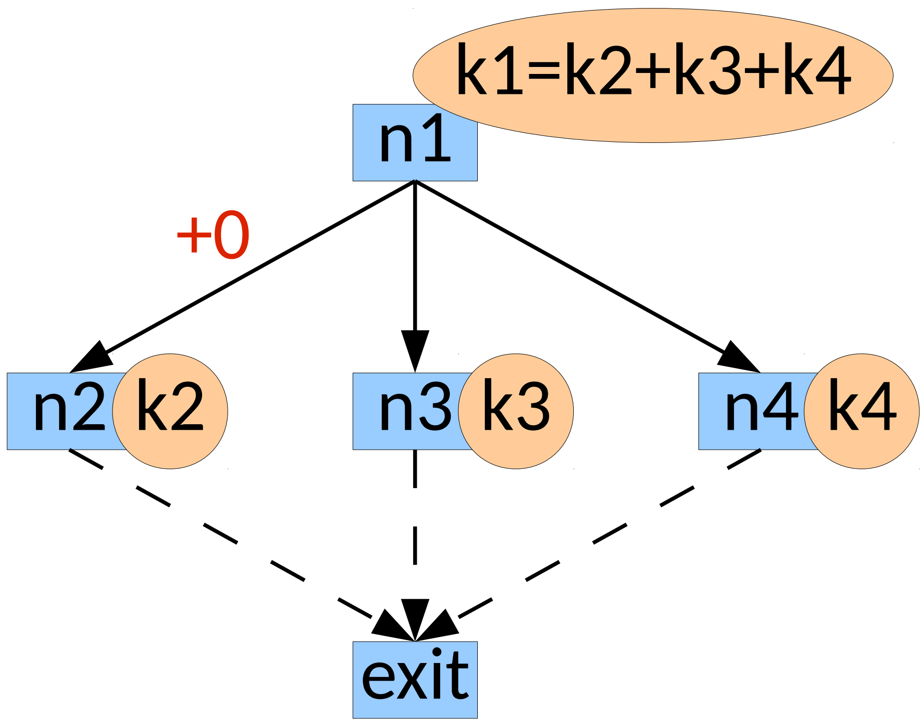
- Step 2: Partition the encoding space locally at each node

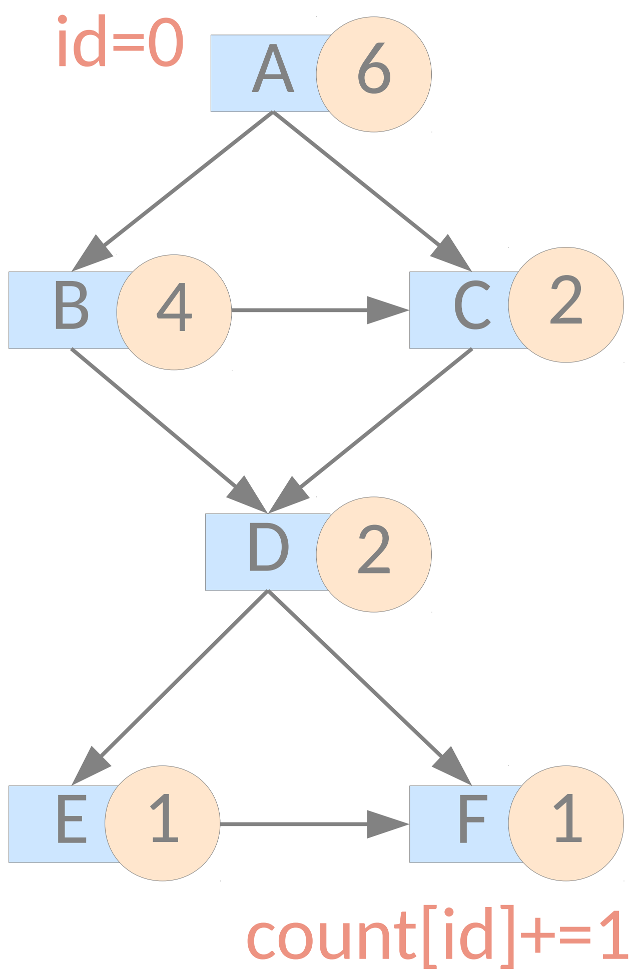




Path Profiling

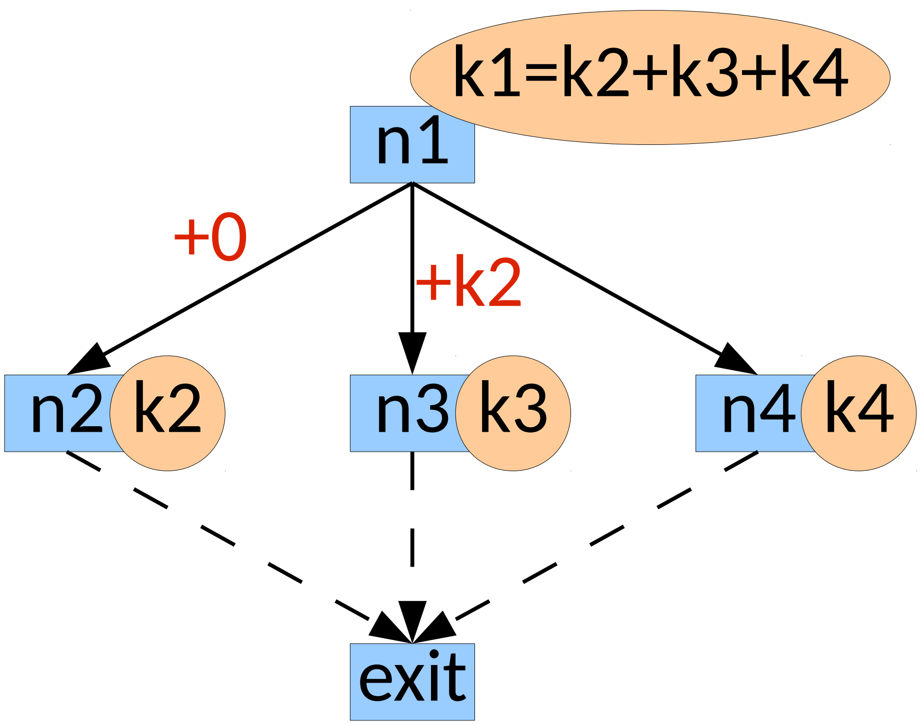
- Step 2: Partition the encoding space locally at each node

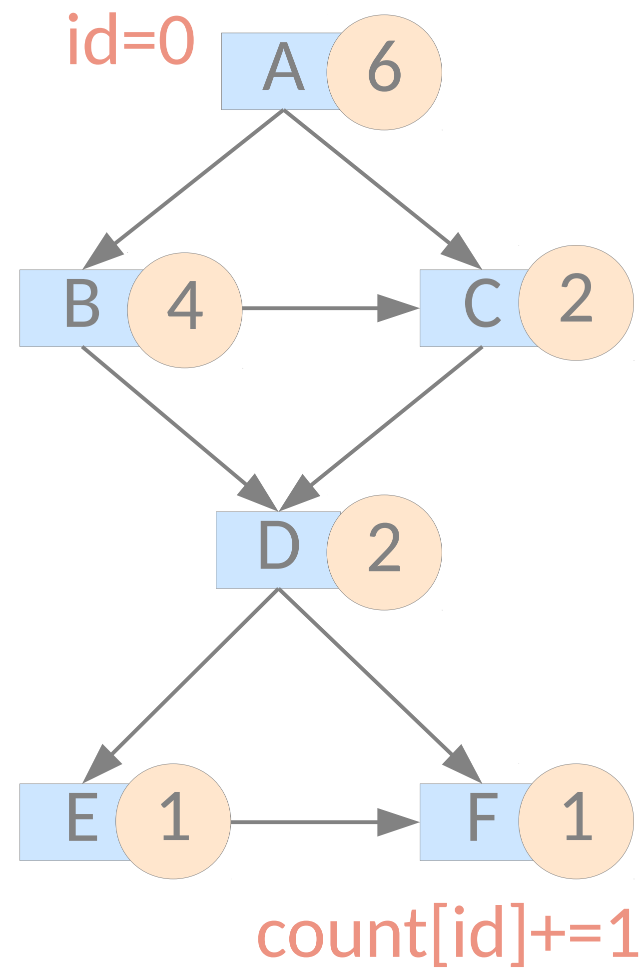




Path Profiling

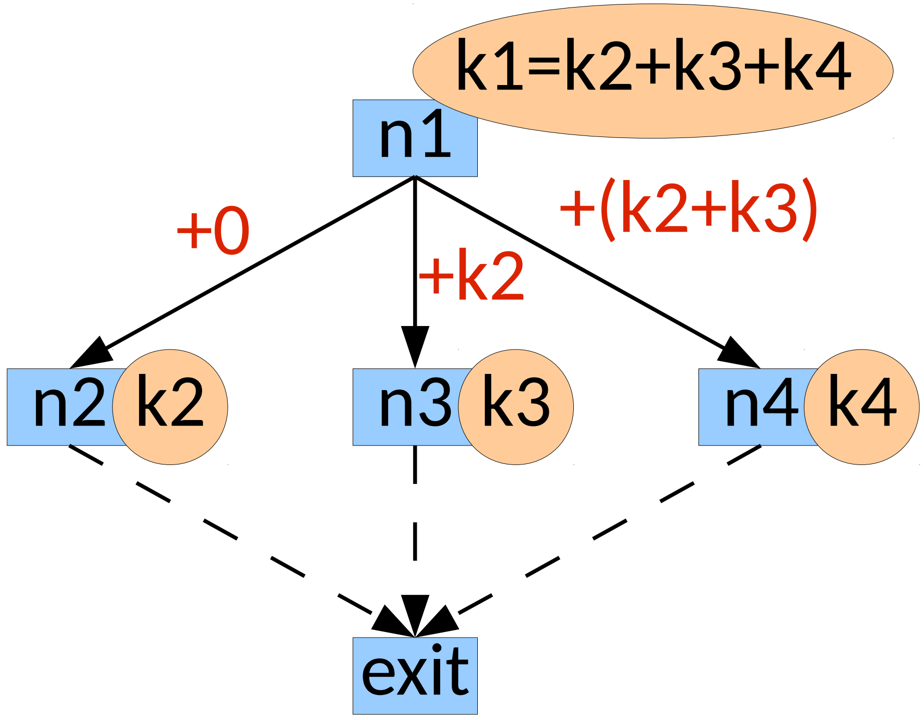
- Step 2: Partition the encoding space locally at each node

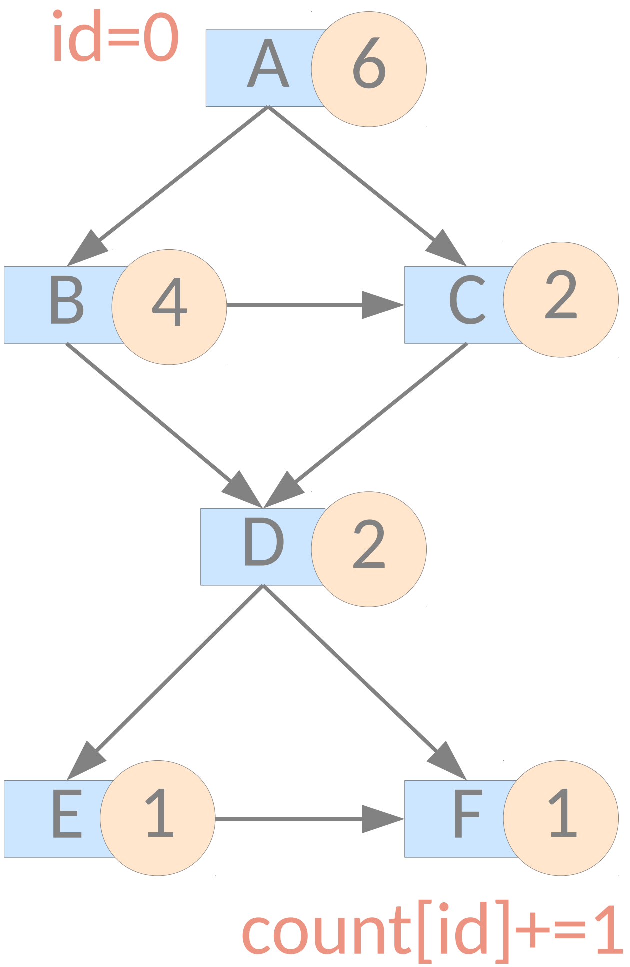




Path Profiling

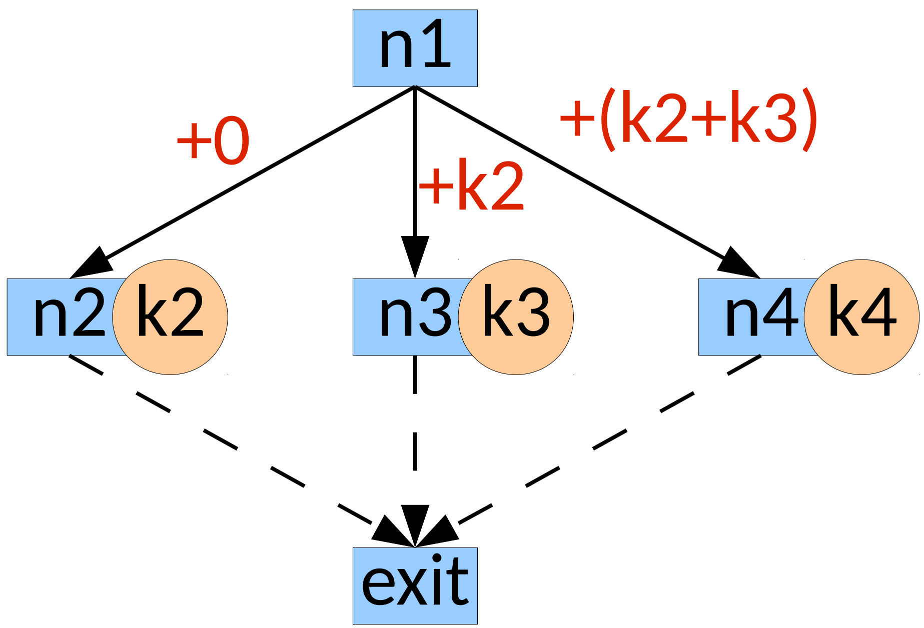
- Step 2: Partition the encoding space locally at each node

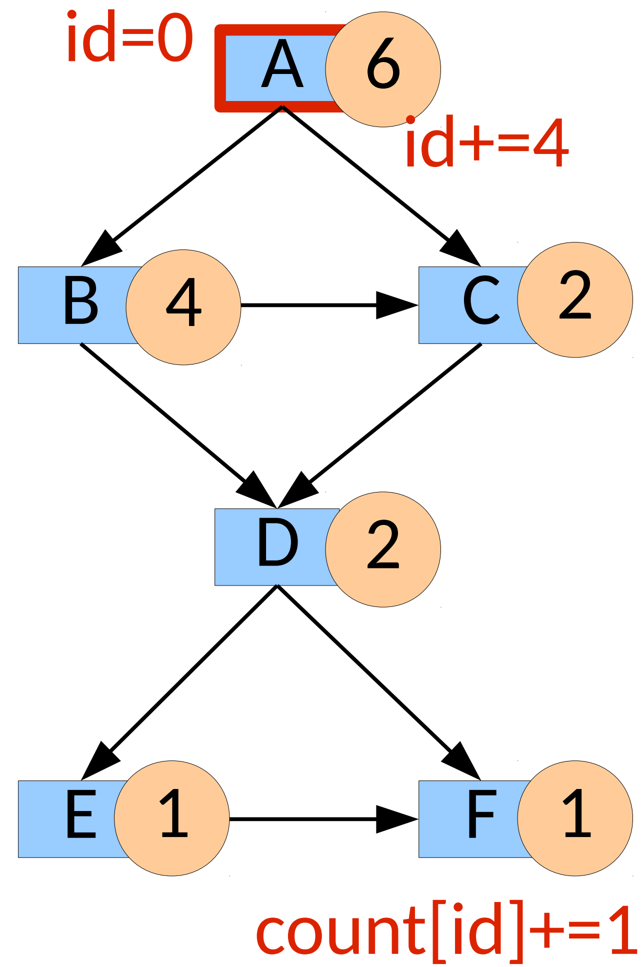




Path Profiling

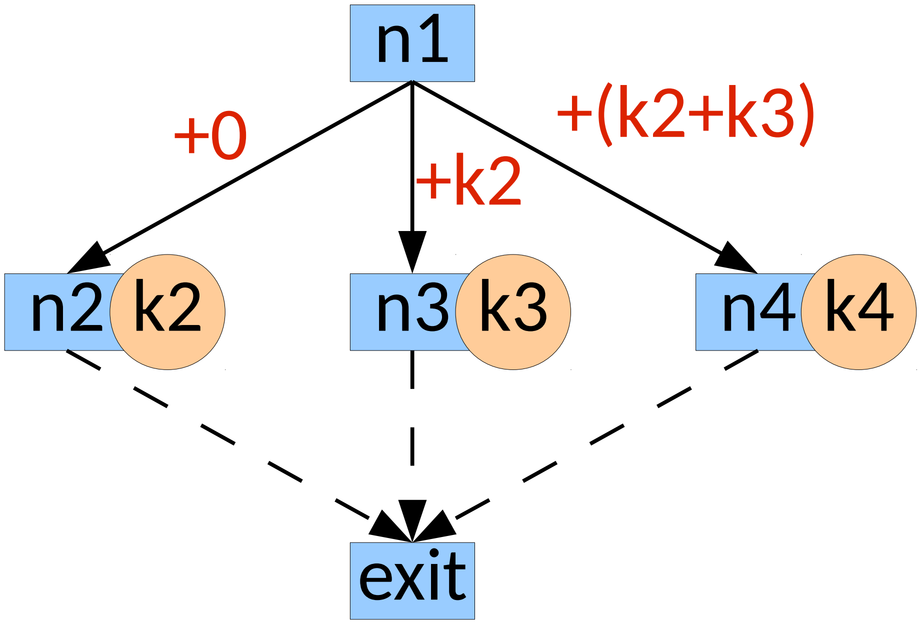
- Step 2: Partition the encoding space locally at each node

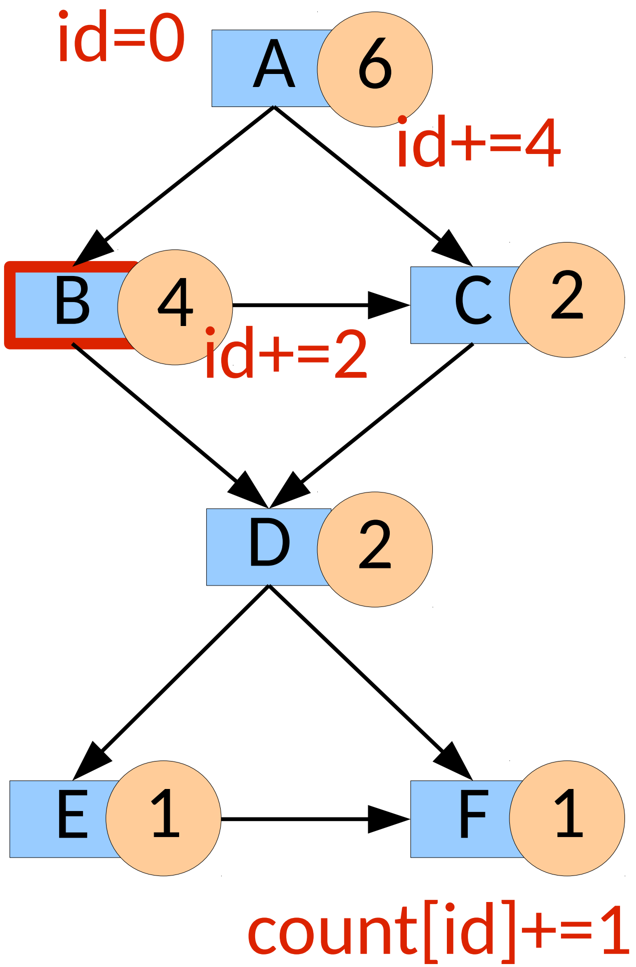




Path Profiling

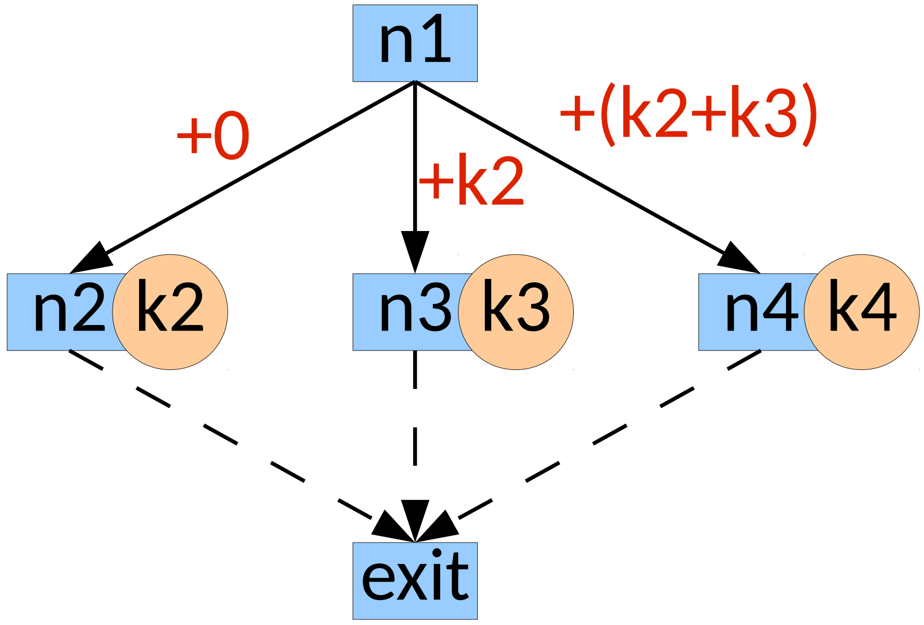
- Step 2: Partition the encoding space locally at each node

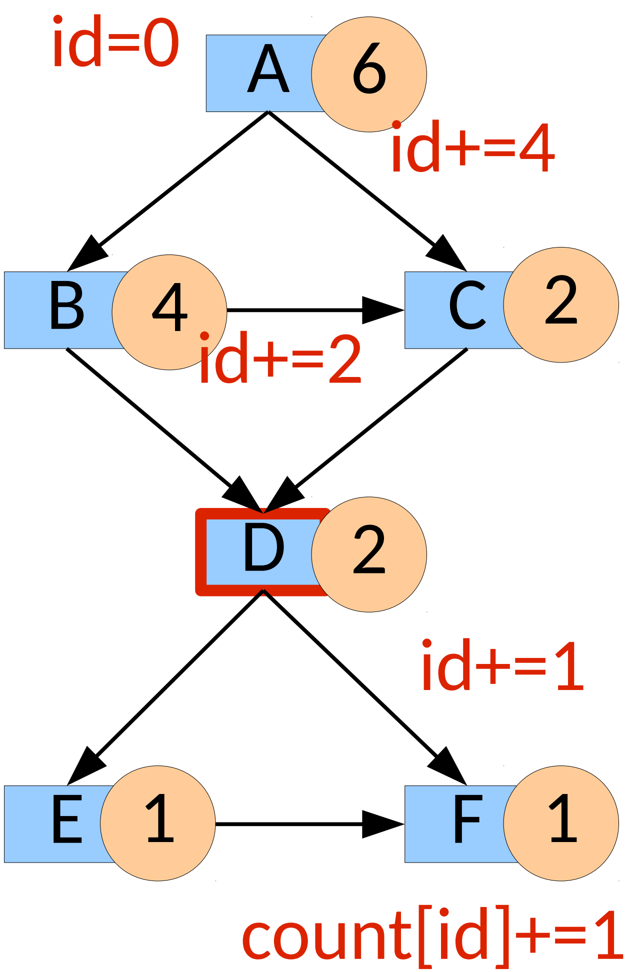




Path Profiling

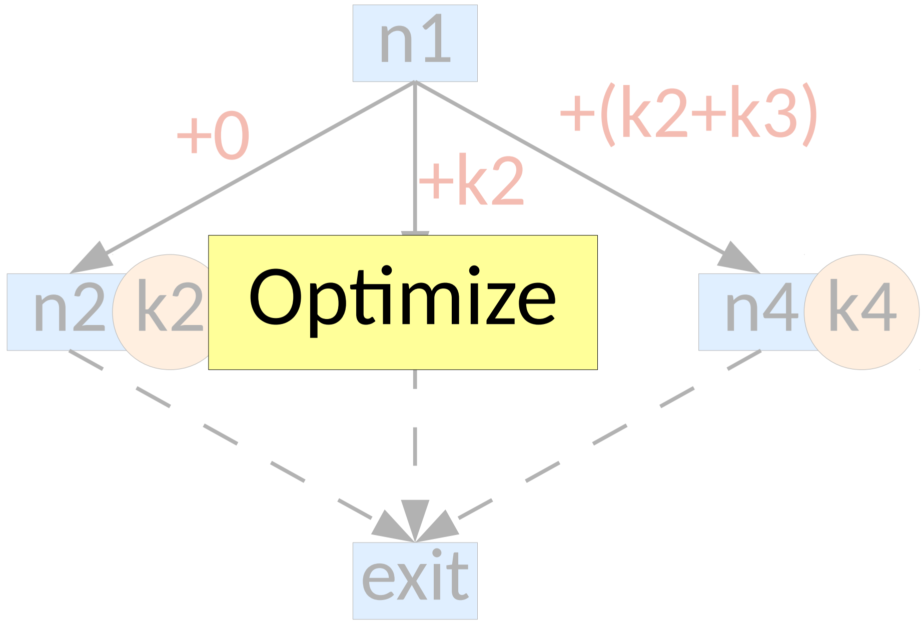
- Step 2: Partition the encoding space locally at each node

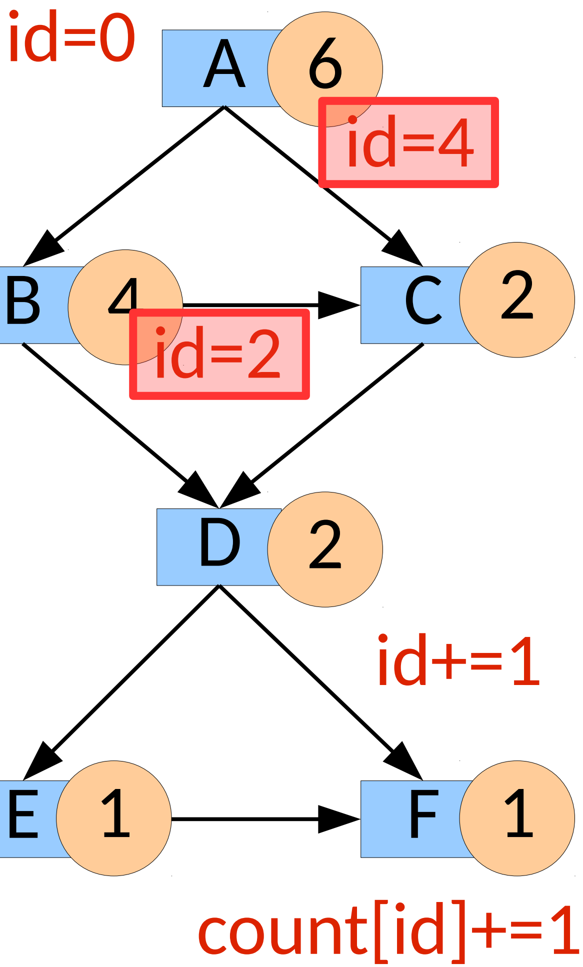




Path Profiling

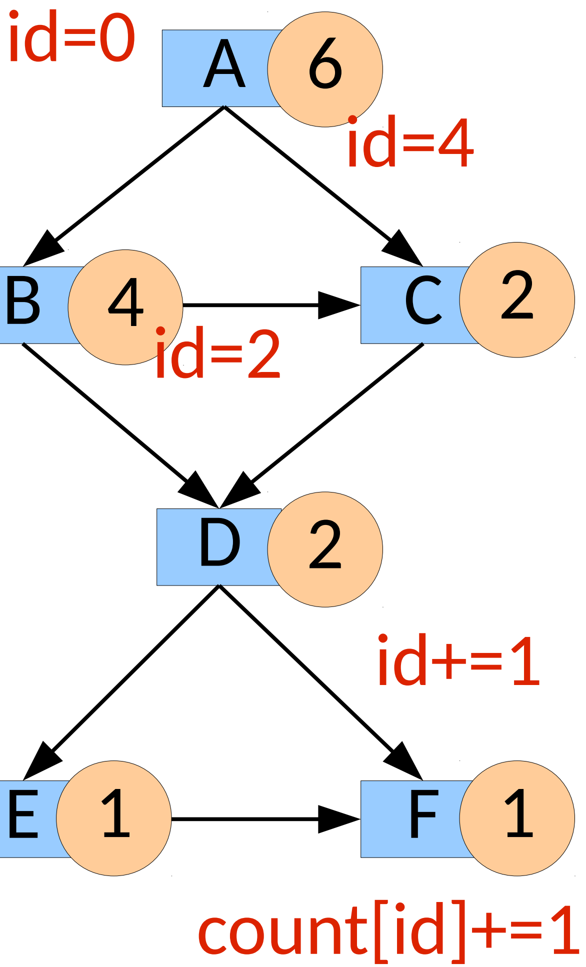
- Step 2: Partition the encoding space locally at each node





Path Profiling: Decoding

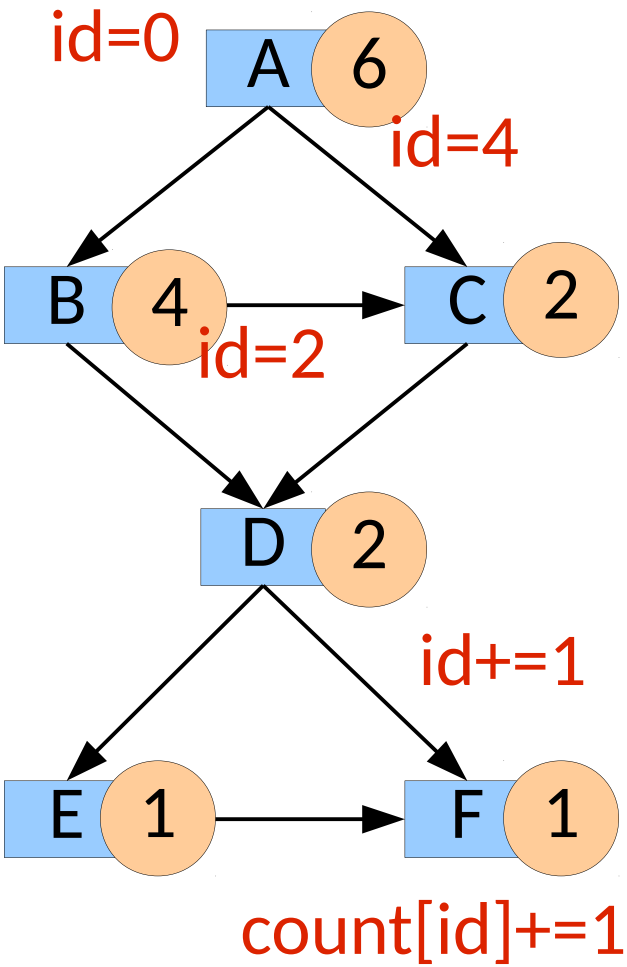
How do we know which IDs map to which paths?



Path Profiling: Decoding

How do we know which IDs map to which paths?

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

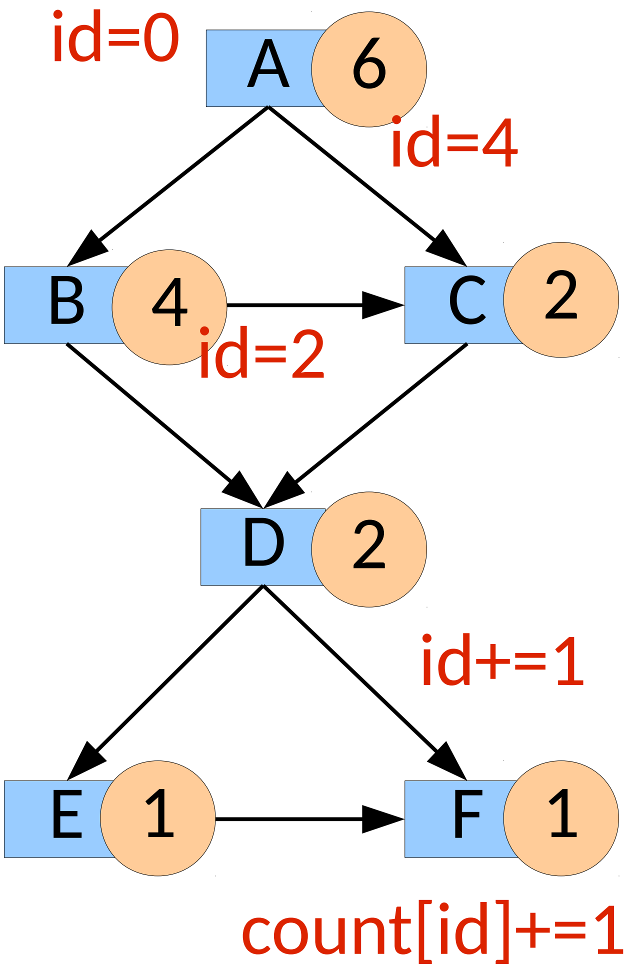


Path Profiling: Decoding

How do we know which IDs map to which paths?

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

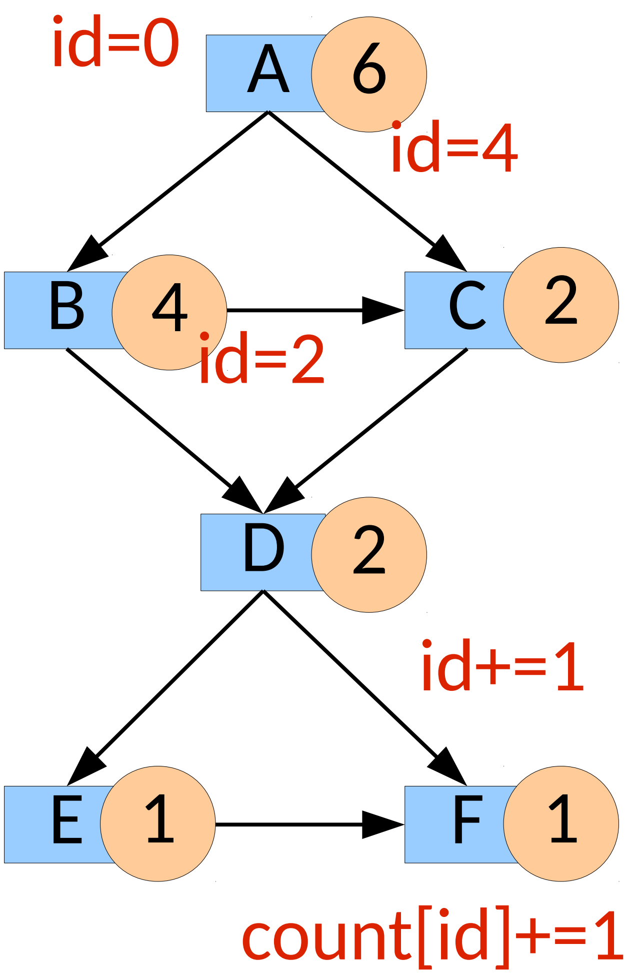
Why could it be large?



Path Profiling: Decoding

How do we know which IDs map to which paths?

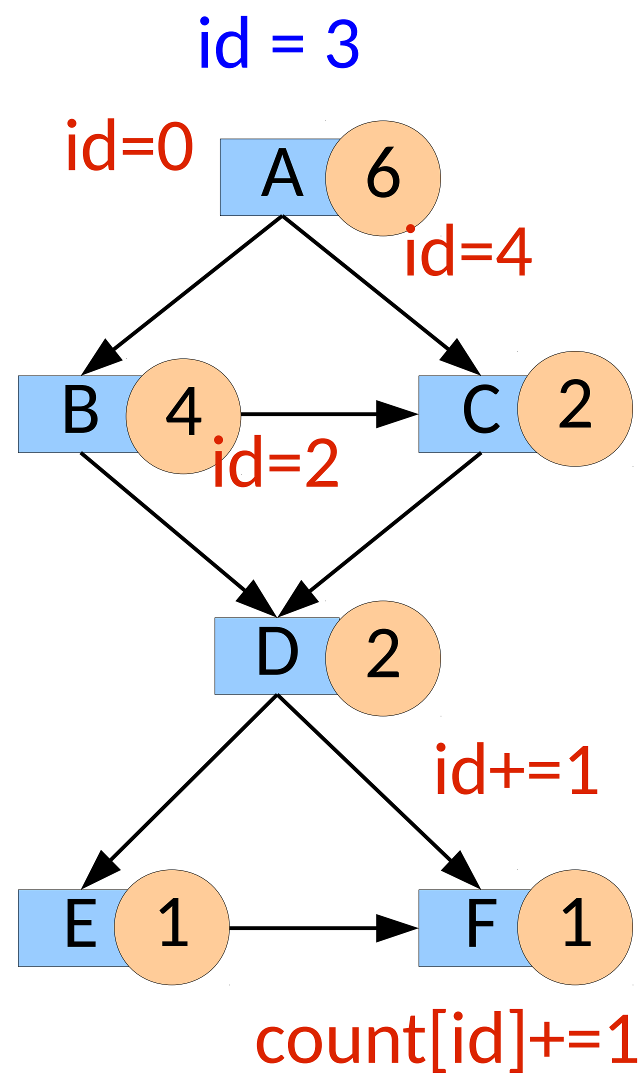
- Naive:
  - Keep a dictionary (***large***)
- Better:
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'



Path Profiling: Decoding

How do we know which IDs map to which paths?

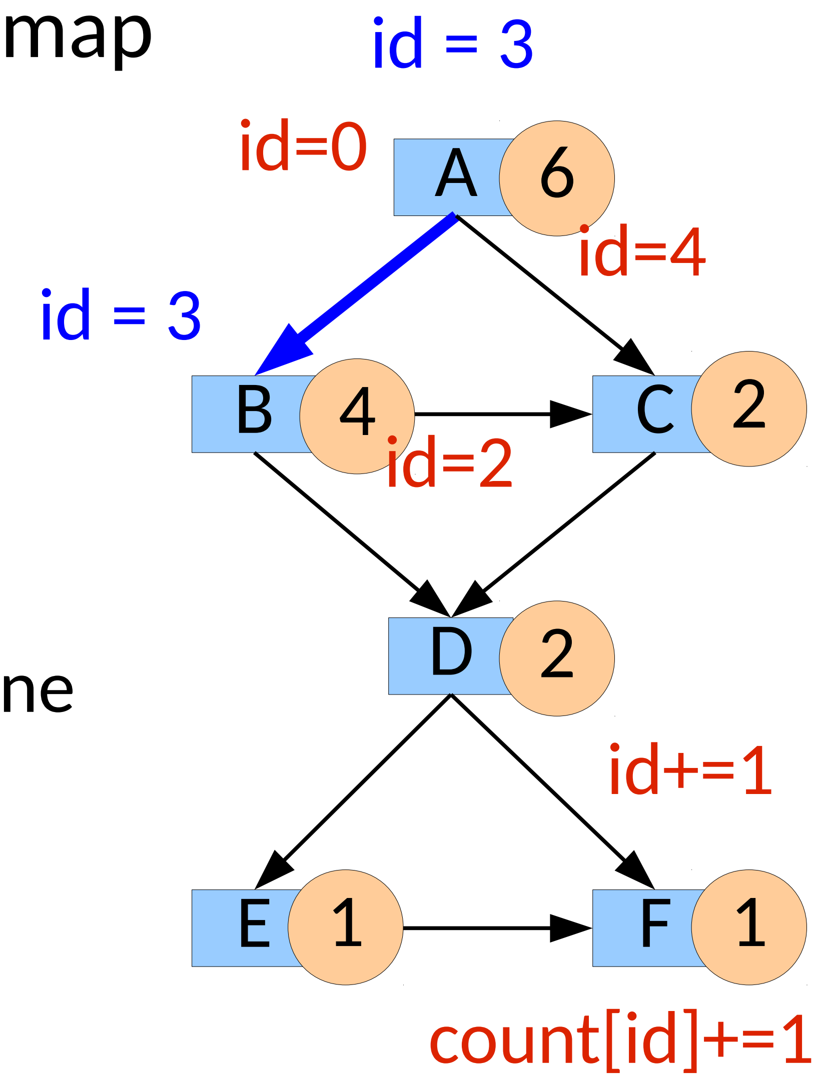
- Naive:
  - Keep a dictionary (***large***)
- Better:
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'



Path Profiling: Decoding

How do we know which IDs map to which paths?

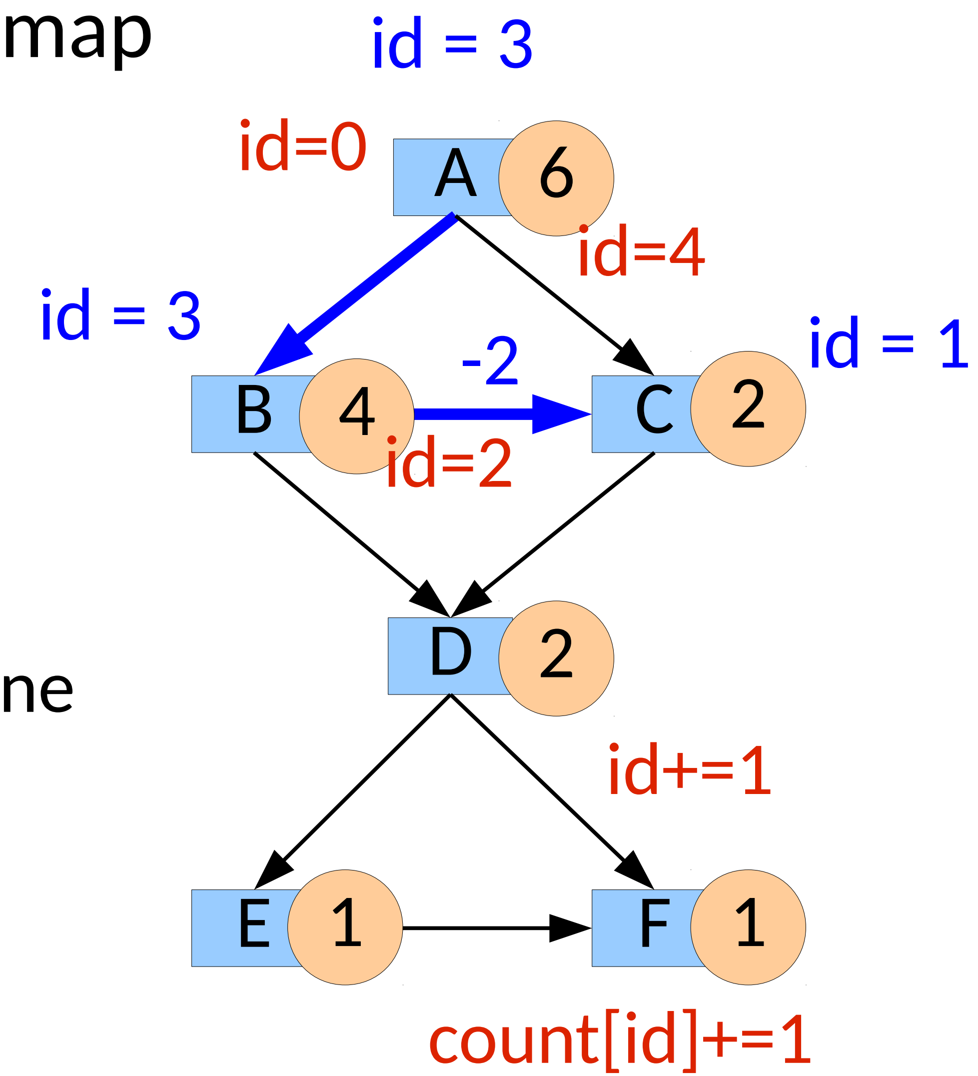
- Naive:
  - Keep a dictionary (***large***)
- Better:
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'



Path Profiling: Decoding

How do we know which IDs map to which paths?

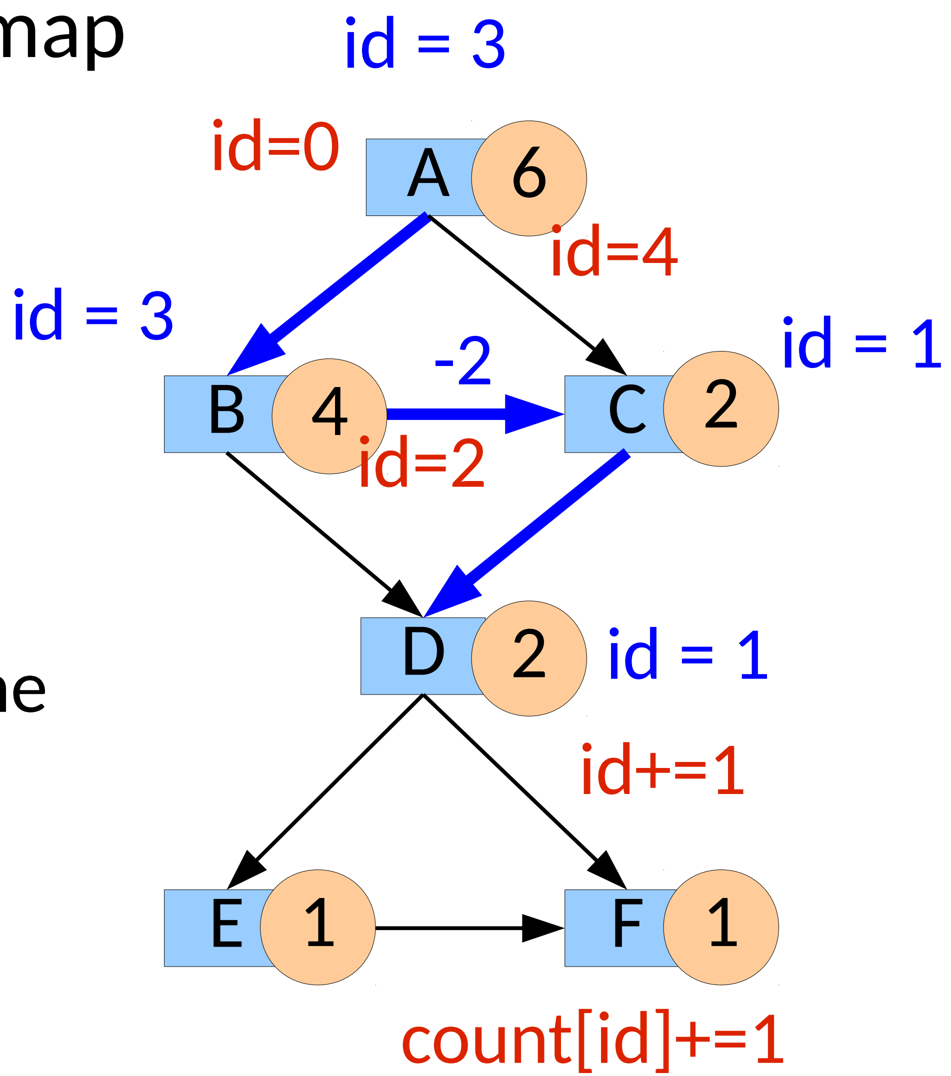
- Naive:
  - Keep a dictionary (***large***)
- Better:
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'



Path Profiling: Decoding

How do we know which IDs map to which paths?

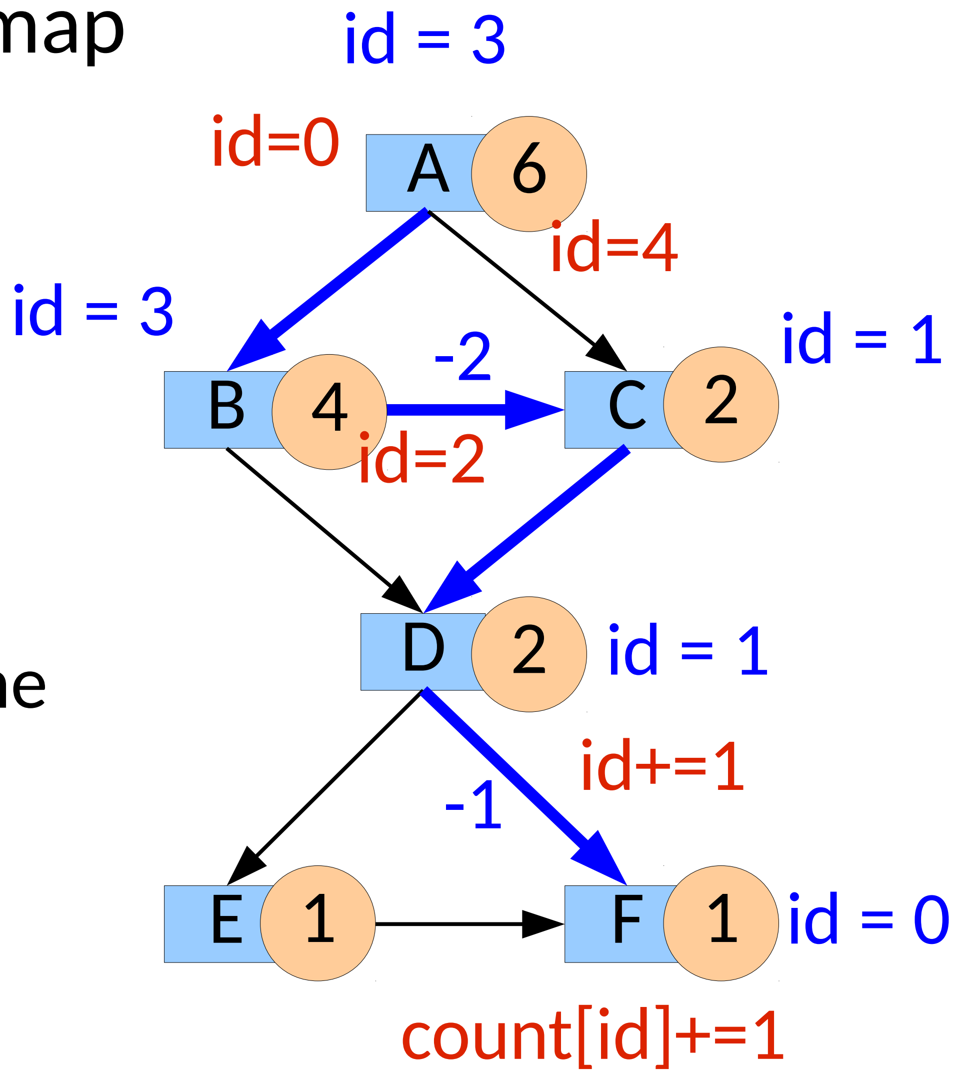
- Naive:
  - Keep a dictionary (***large***)
- Better:
  - Decode using same graph
  - Follow the CFG and only one path will 'fit'



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'



Path Profiling: Results

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

Path Profiling: Results

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

Path Profiling: Results

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

Path Profiling: Results

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

Path Profiling: Results

Benchmark	Base Time (sec)	PP Overhead %	QPT2 Overhead %	PP/ QPT	Path Inc (million)	Edge Inc (x Path)	Hashed Inc %	Inst/ Inc
099.go	885.0	53.4	24.1	2.2	1002.4	1.5	27.7	33.2
124.m88ksim	571.0	35.6	18.7	1.9	4824.9	1.2	3.9	16.2
126.gcc	322.0	96.9	52.8	1.8	9.4	1.7	16.8	15.1
129.compress	351.0	19.4	21.9	0.9	3015.7	1.5	0.0	16.6
130.li	480.0	25.4	26.7	1.0	3282.4	1.4	1.2	16.8
132.ijpeg	749.0	17.4	16.3	1.1	1164.9	1.1	1.2	31.0
134.perl	332.0	72.9	51.5	1.4	1133.0	1.9	23.4	22.2
147.vortex	684.0	37.7	34.1	1.1	3576.3	1.5	23.7	20.3
CINT95 Avg:		44.8	30.8	1.4	22251.1	1.5	12.2	21.4
101.tomcatv	503.0	19.9	2.8	7.1	574.6	1.1	95.8	93.0
102.sw	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	162.9
103.su2	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	92.8
104.hy	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	43.1
107.mgrid	872.0	6.3	3.2	2.0	1035.2	1.0	7.7	133.5
110.applu	715.0	71.0	12.0	5.9	2111.4	1.1	99.1	44.8
125.turb3d	1066.0	5.5	7.4	0.7	2952.8	1.1	0.0	56.5
141.apsi	492.0	7.7	1.8	4.2	599.3	1.1	3.5	84.0
145.fpppp	1927.0	14.6	-2.6	-5.6	395.0	1.8	42.5	636.0
146.wave5	620.0	16.9	6.1	2.8	737.3	1.3	65.0	74.1
CFP95 Avg:		19.8	4.3	4.0	1081.8	1.2	41.3	142.1
Average:		30.9	16.1	2.8	1601.5	1.3	28.4	88.4

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

Path Profiling: Results

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

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

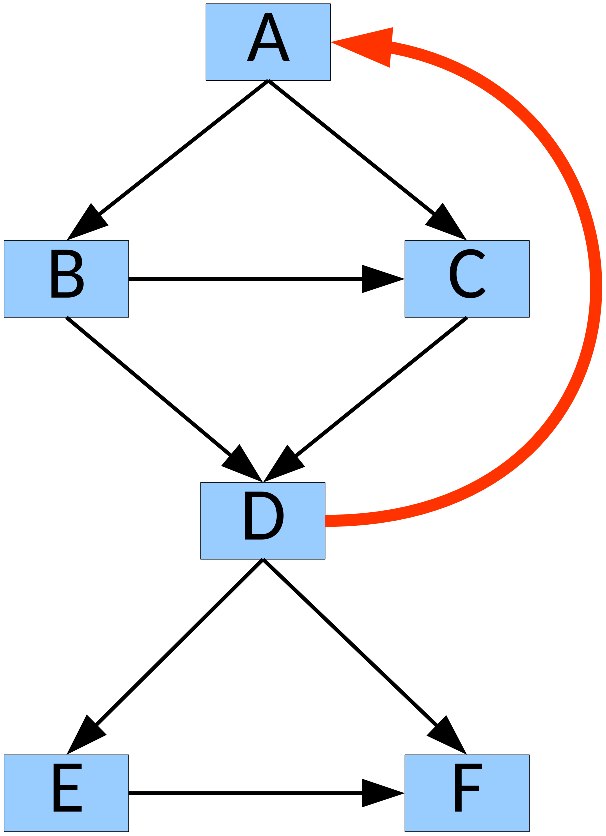
What would you add or change to the evaluation?

Path Profiling

Are there cases where this approach fails?

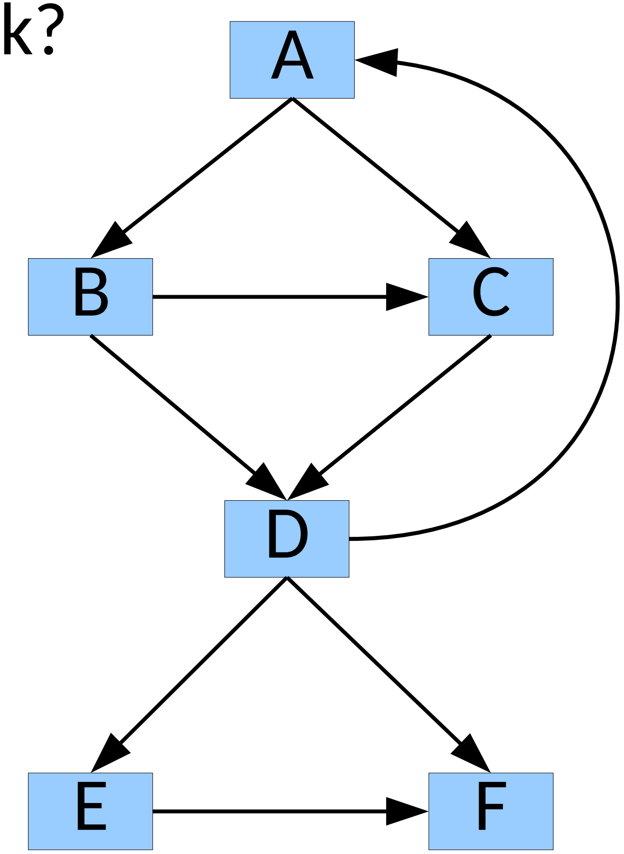
Path Profiling

- What about loops / cycles?



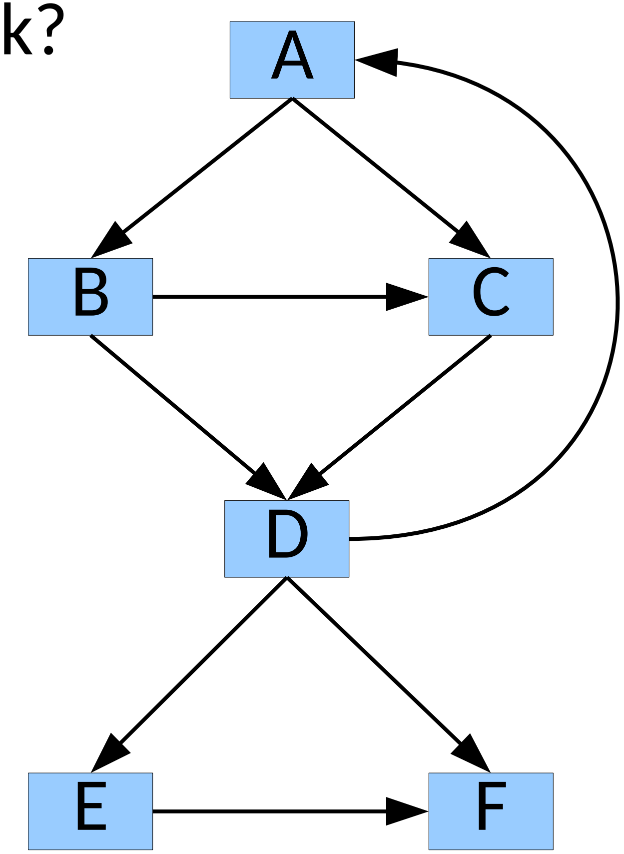
Path Profiling

- What about loops / cycles?
  - Does the existing approach work?



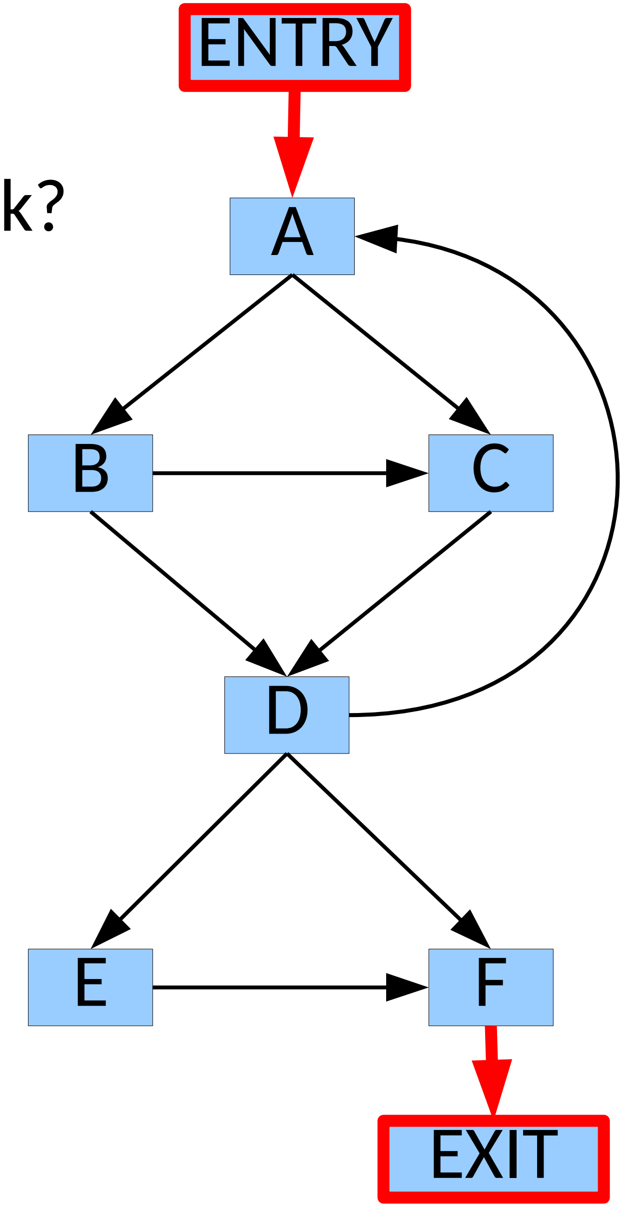
Path Profiling

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



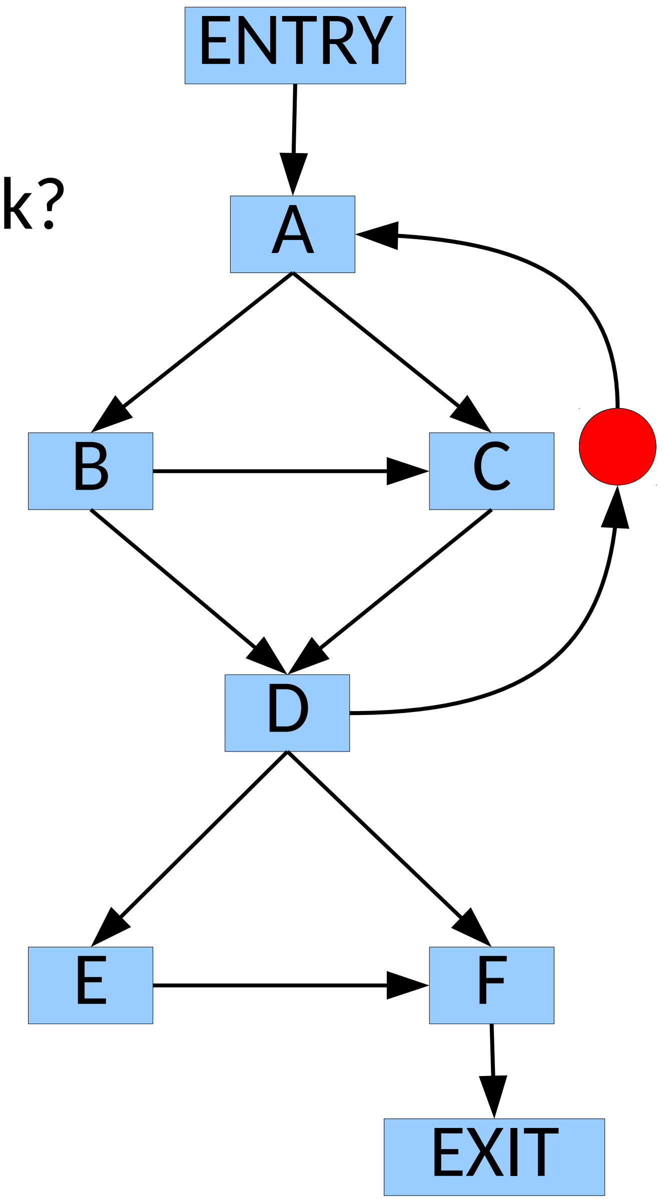
Path Profiling

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



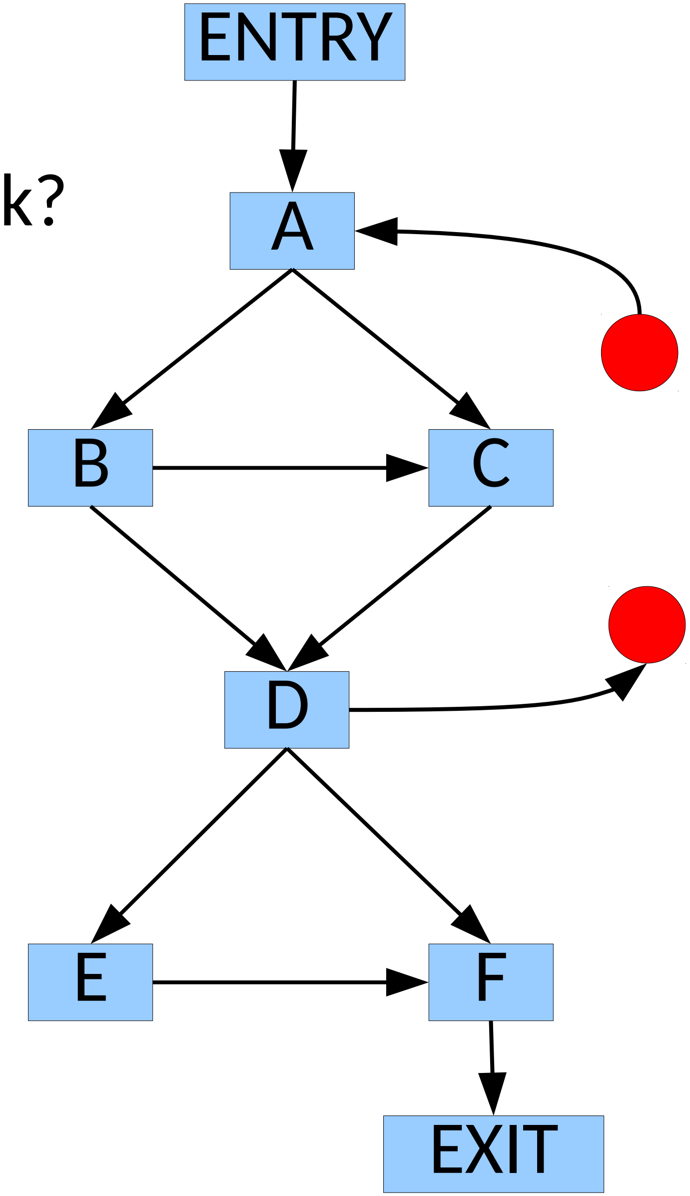
Path Profiling

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



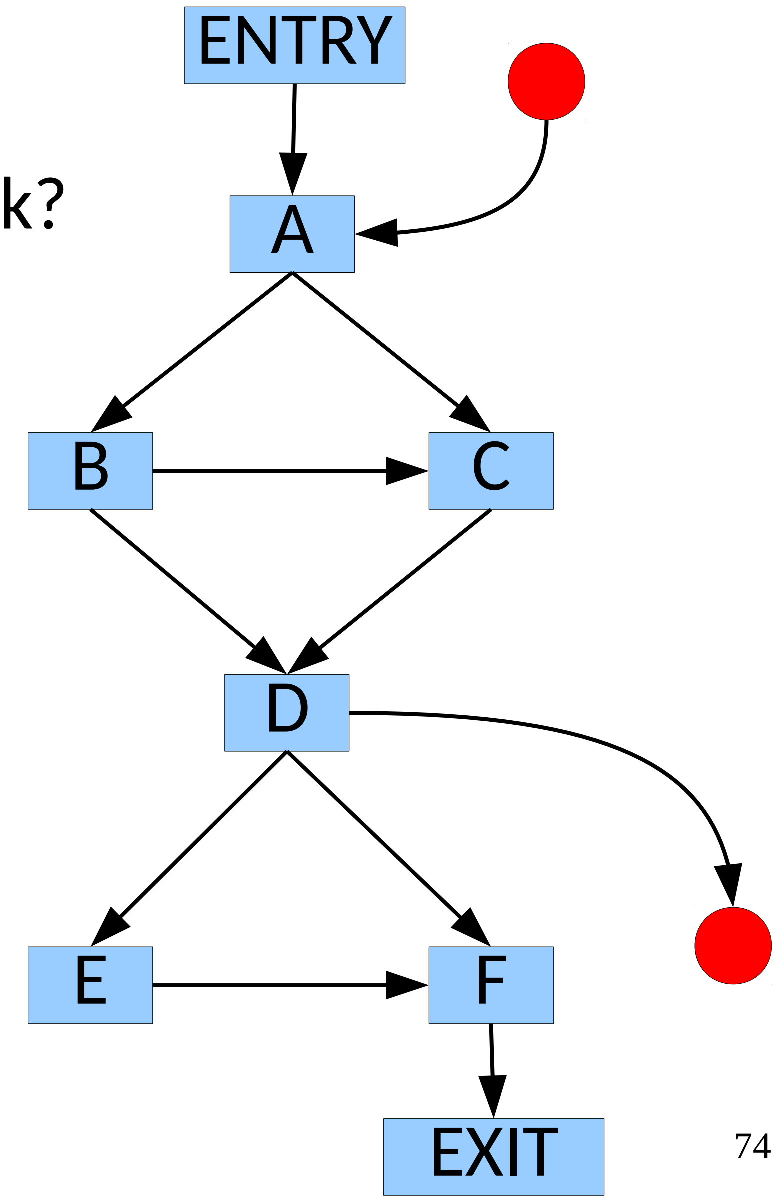
Path Profiling

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



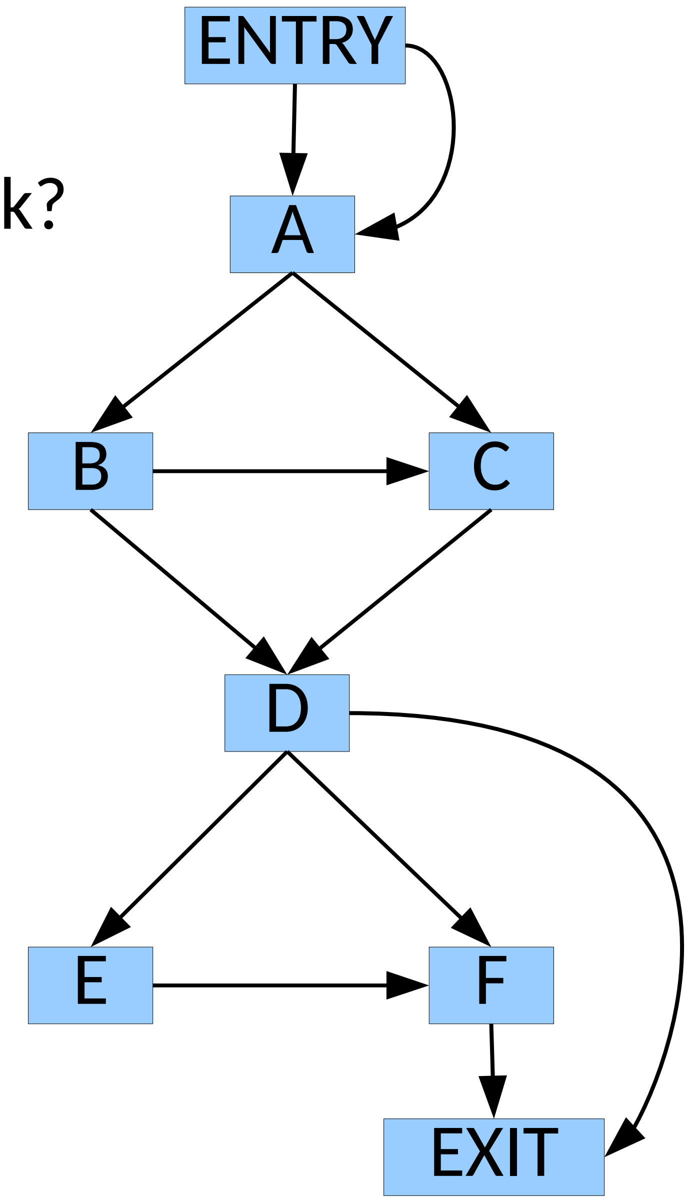
Path Profiling

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



Path Profiling

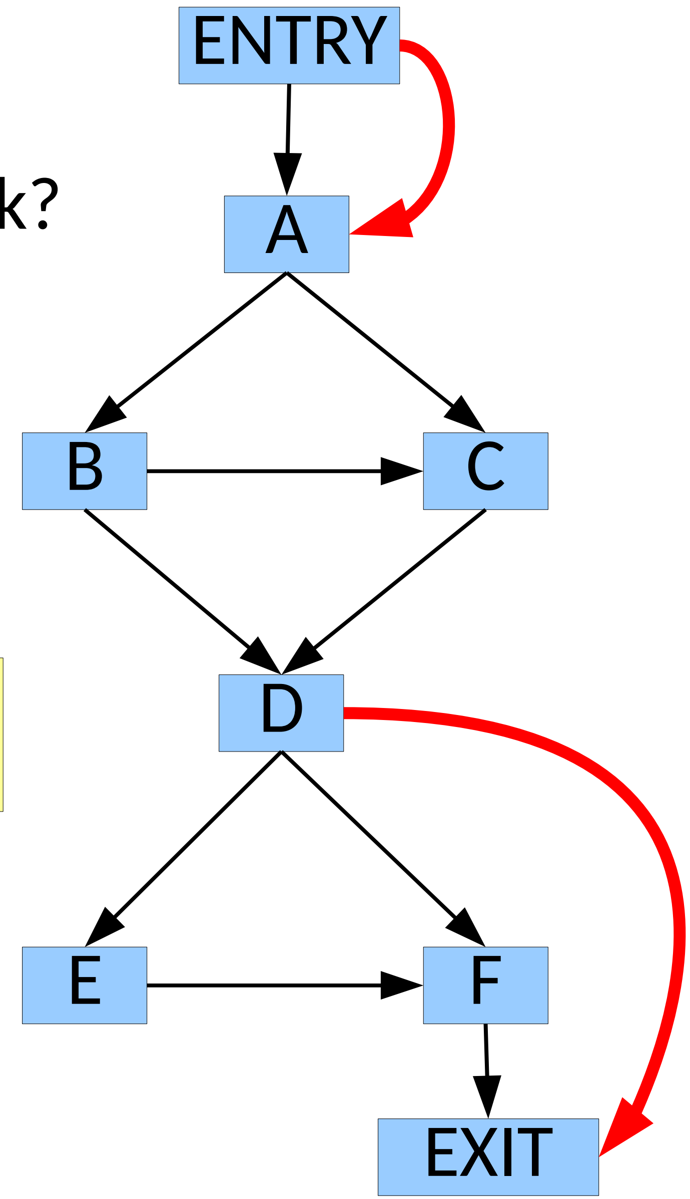
- What about loops / cycles?
  - Does the existing approach work?
  - How could we resolve it?



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

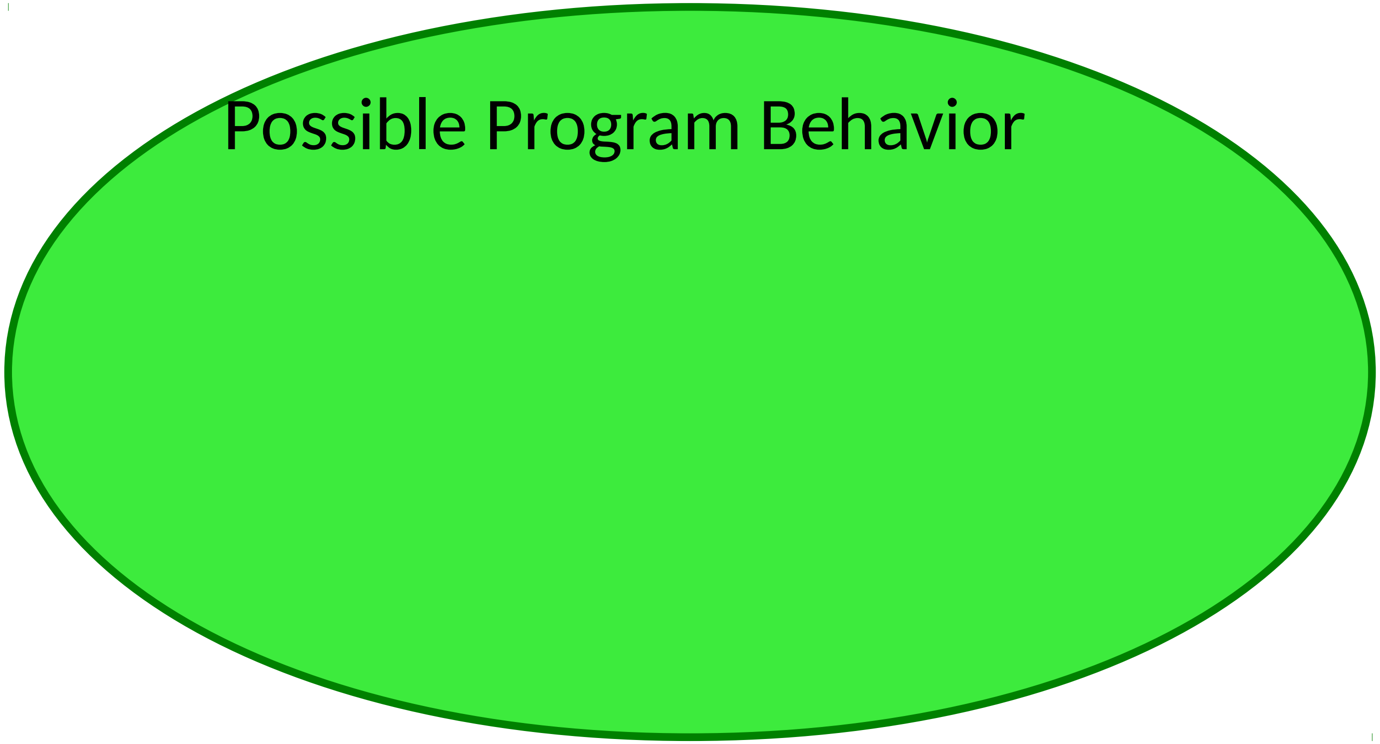
- Path profiling is a *dynamic* analysis
  - It analyzes an actual execution
  - “What were frequent paths for this input”
  - “What were frequent paths for this set of inputs”

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?

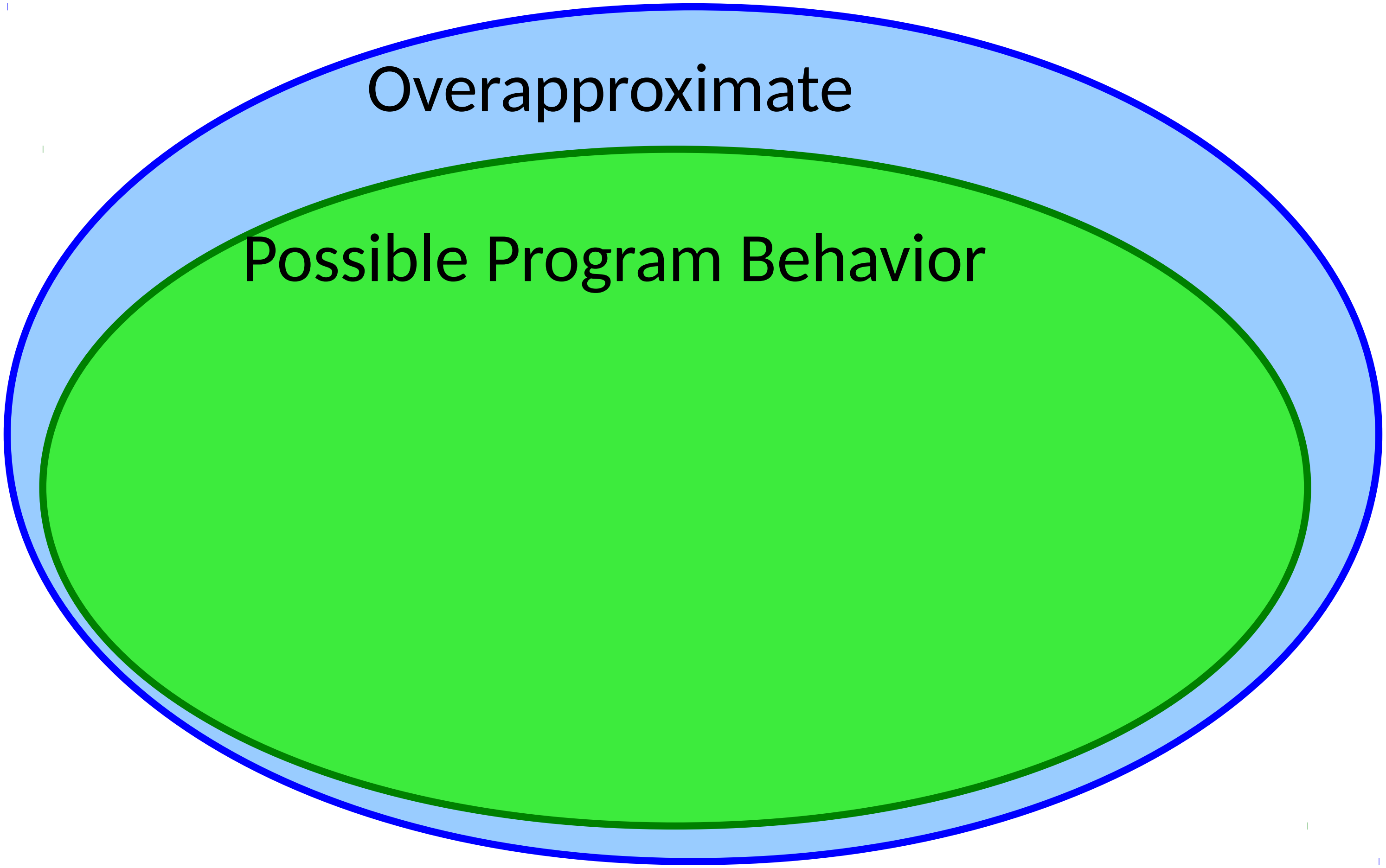
Approximation

Modeled program behaviors



Approximation

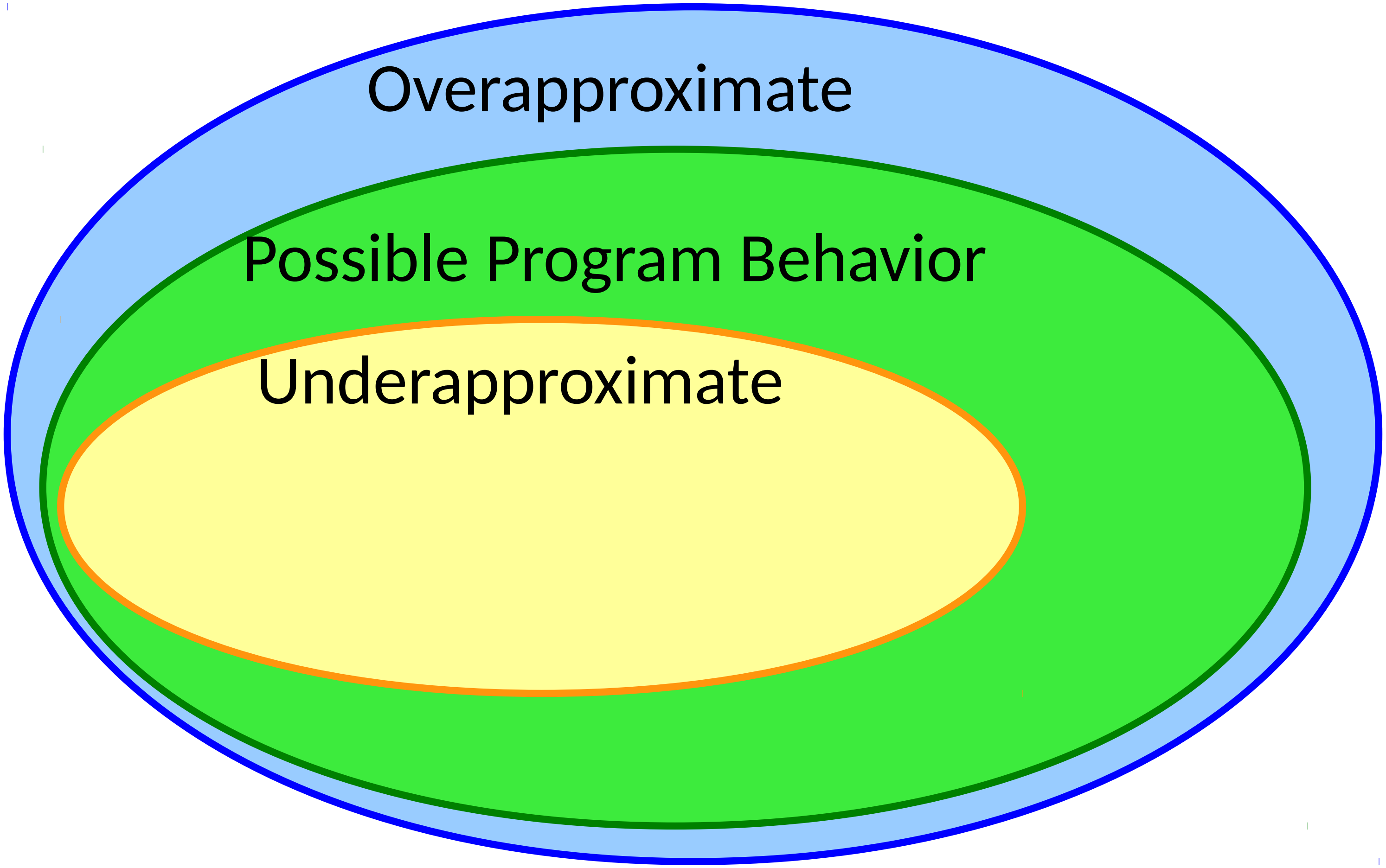
Modeled program behaviors



Consider some behaviors possible when they are not.

Approximation

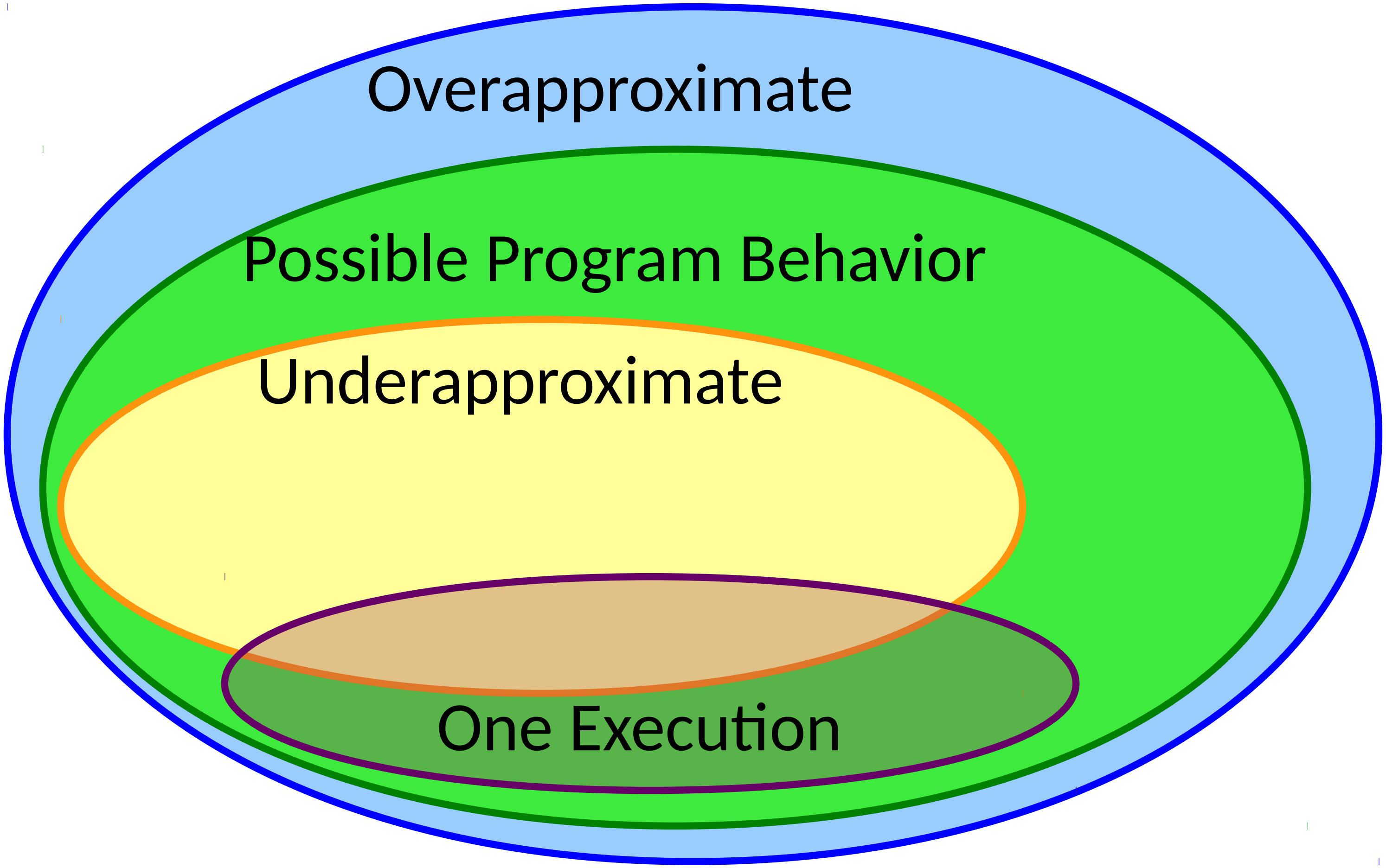
Modeled program behaviors



Ignore some behaviors that *are* possible.

Approximation

Modeled program behaviors

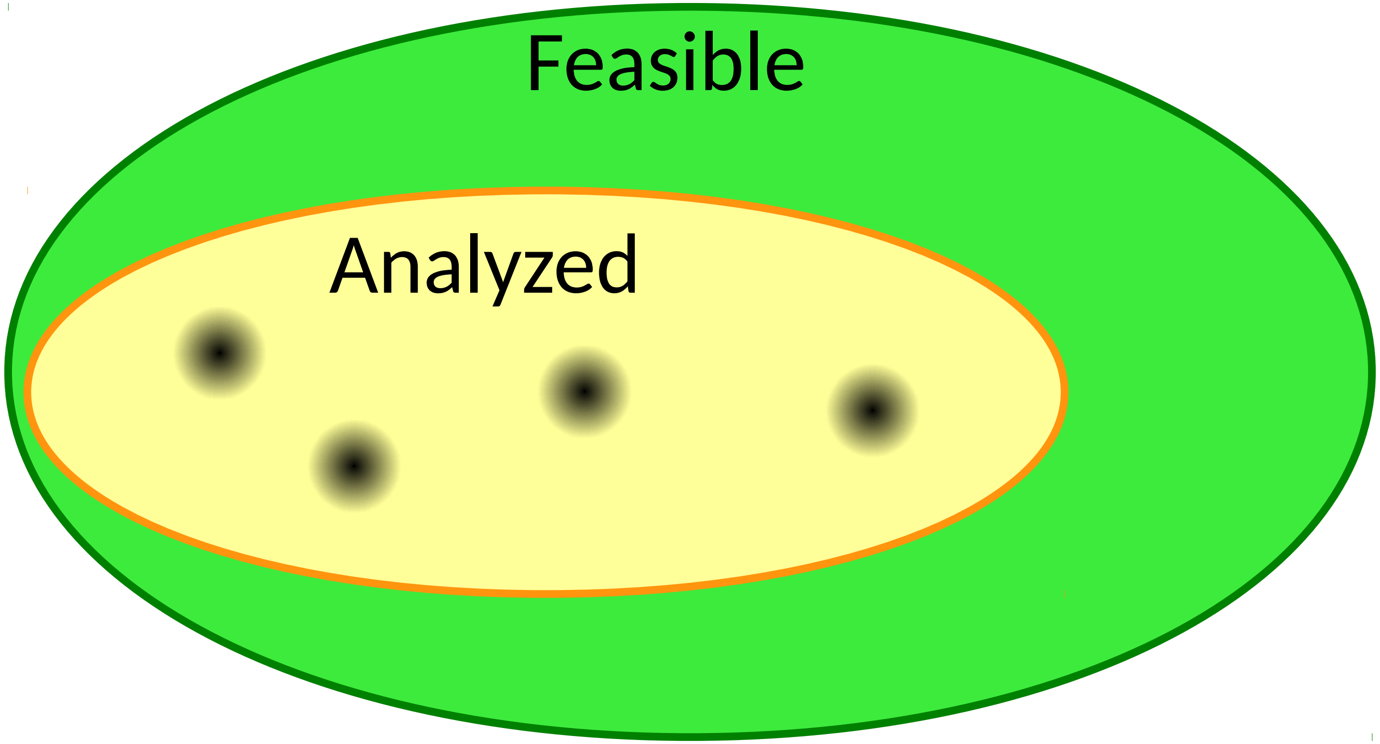


Approximation

- Dynamic Analysis
  - Analyzed ⊆ Feasible

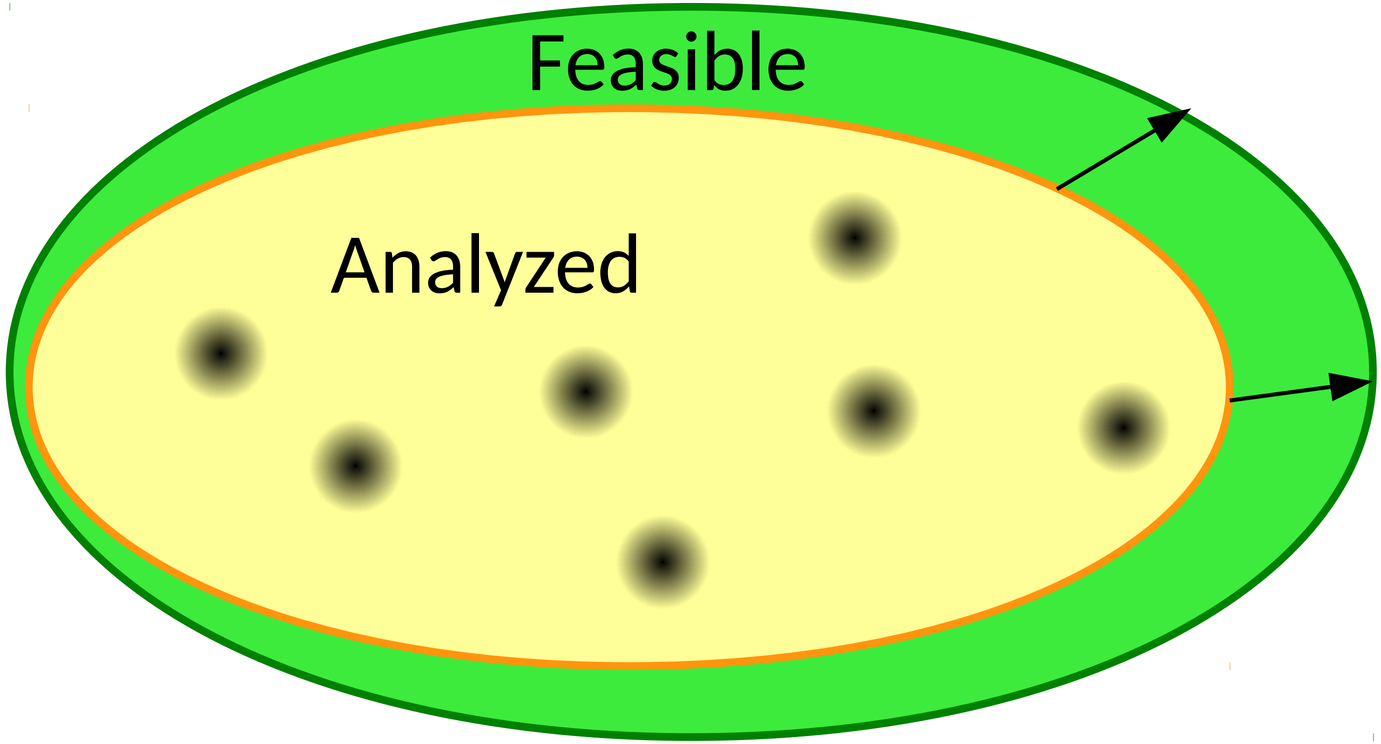
Approximation

- Dynamic Analysis
  - Analyzed ⊆ Feasible



Approximation

- Dynamic Analysis
  - Analyzed ⊆ Feasible
  - As # tests ↑, Analyzed → 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 an 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 an 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)
3) 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?

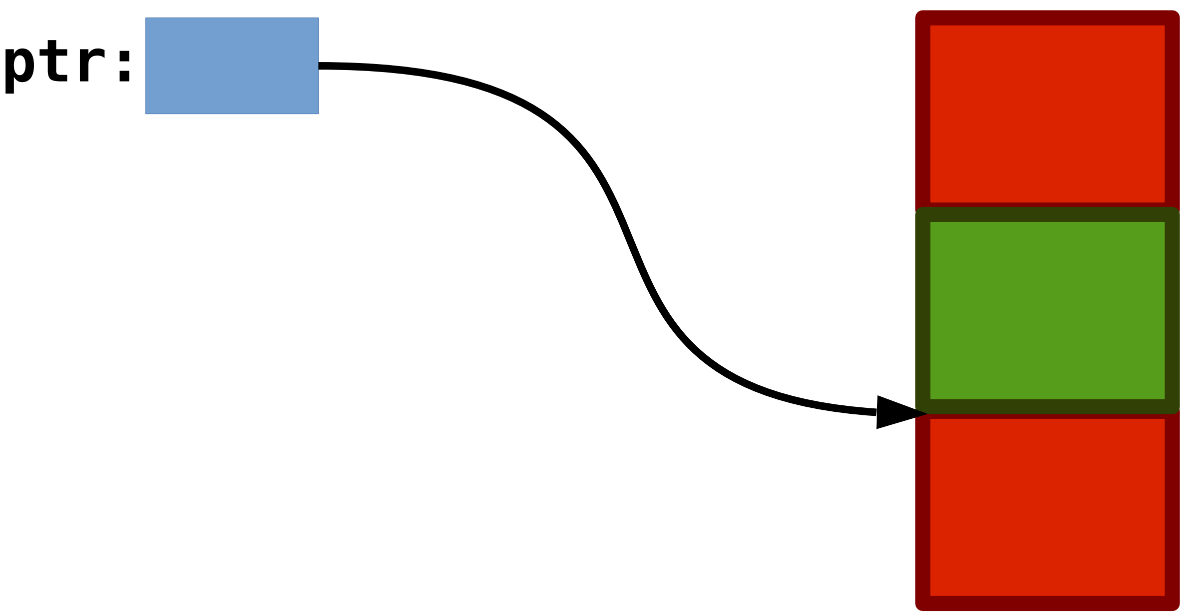
- Replaces `malloc` & `free`

Example: Address Sanitizer

How?

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

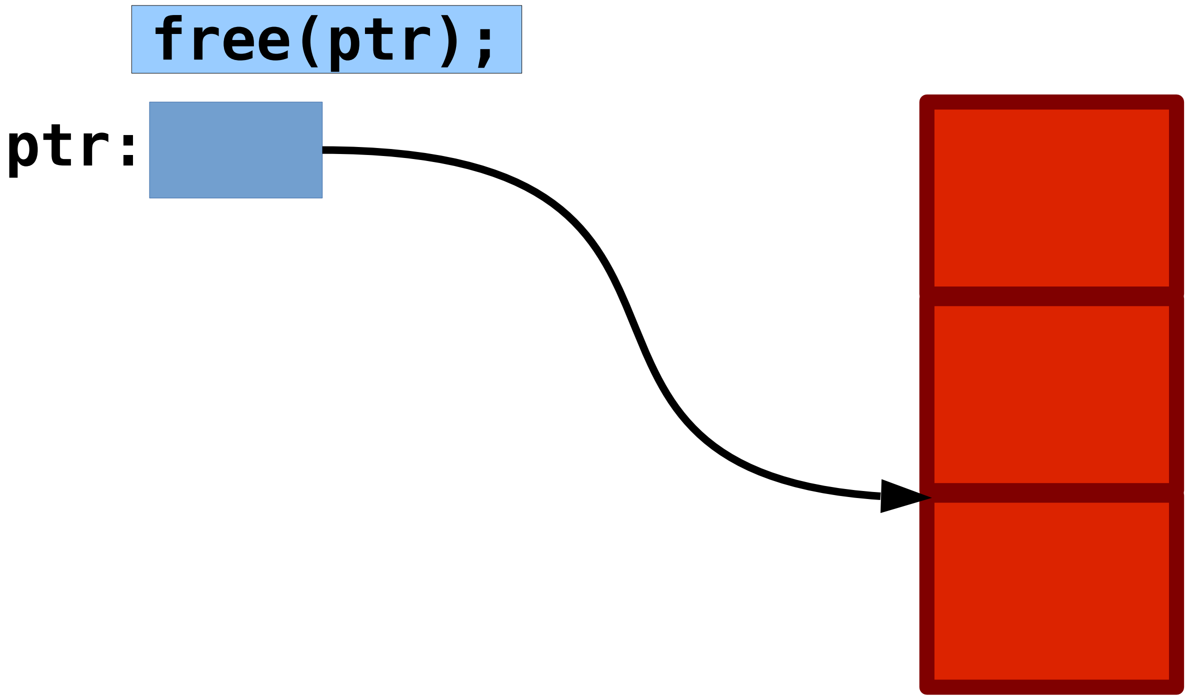
```
ptr = malloc(sizeof(MyStruct));
```



Example: Address Sanitizer

How?

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

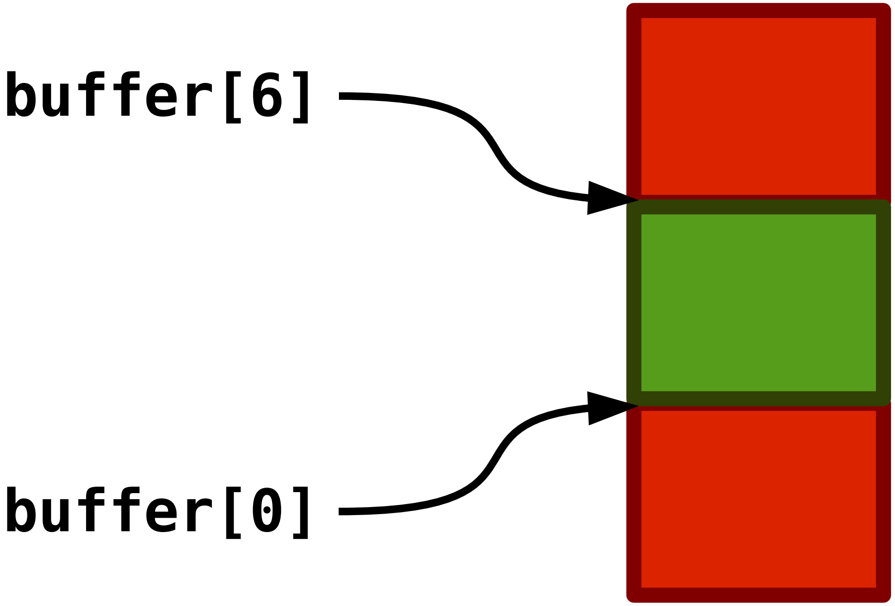


Example: Address Sanitizer

How?

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

```
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?

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

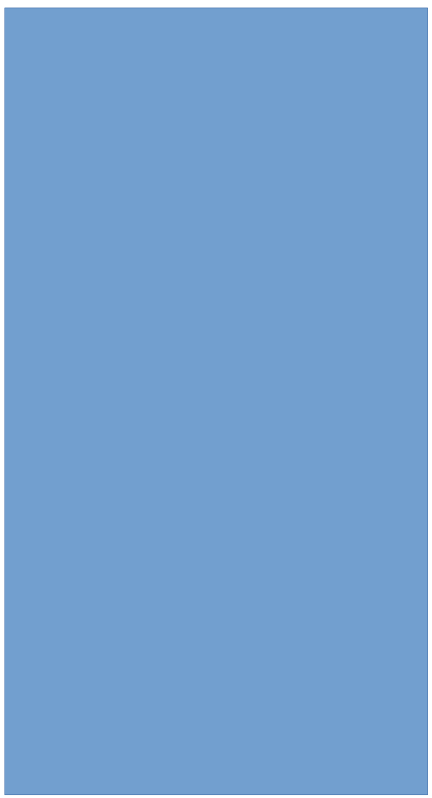
Instrumentation

```
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

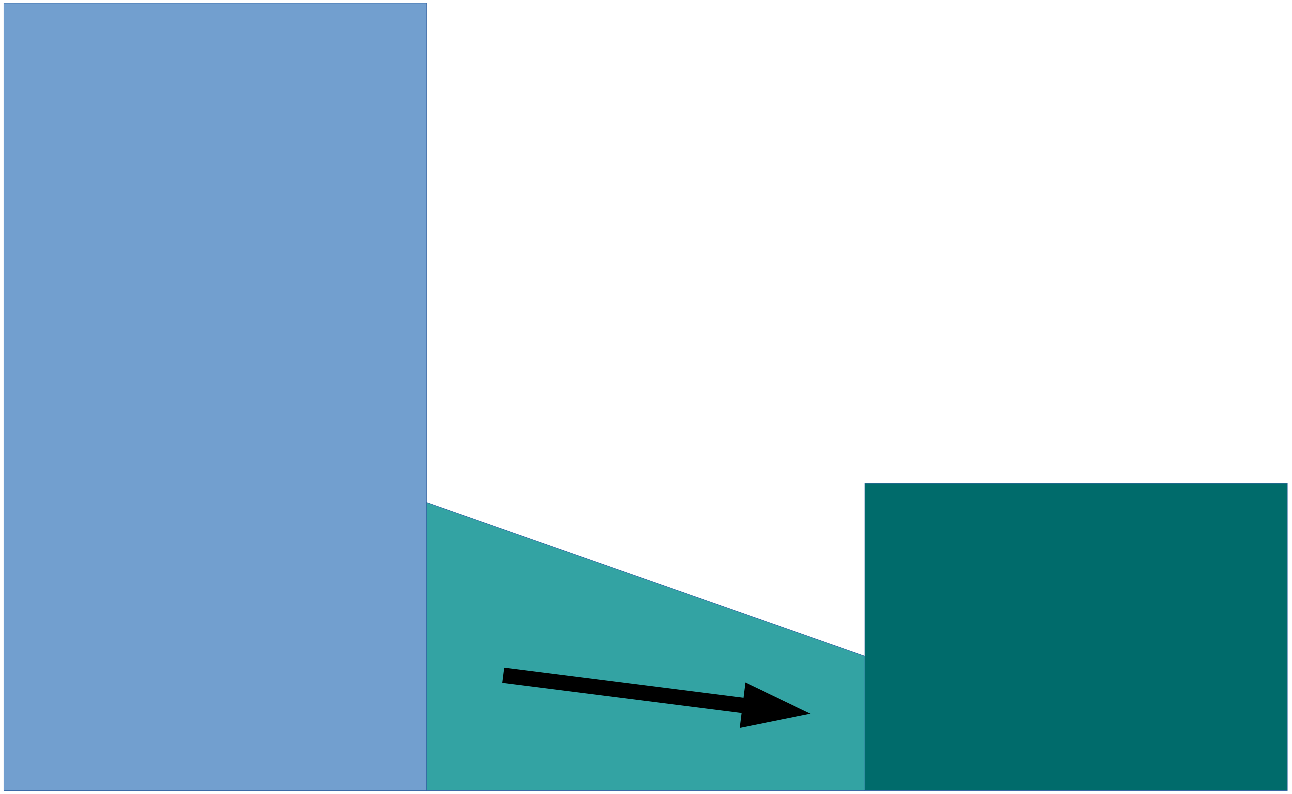


Application Memory

Need to know whether *any byte* of application memory is poisoned.

Example: Address Sanitizer

- Maintain 2 views on memory:



Application Memory *Shadow* 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

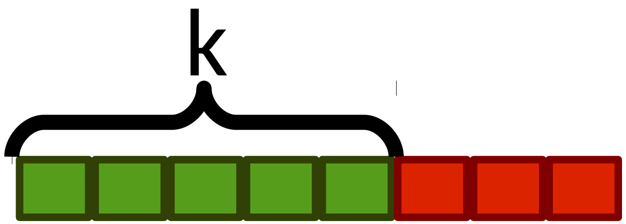
Where have you encountered this before?
(Think OS)

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:  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.

Memory:  Shadow: -1

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
  - Shadow = (Mem >> 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…