# BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-Program Path Sampling and Per-Path Abstract Interpretation

Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang

PURDUE UNIVERSITY

RENMIN UNIVERSITY OF CHINA

UNIVERSITY of VIRGINIA

# Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables

# Intro: Binary Program Dependence Analysis

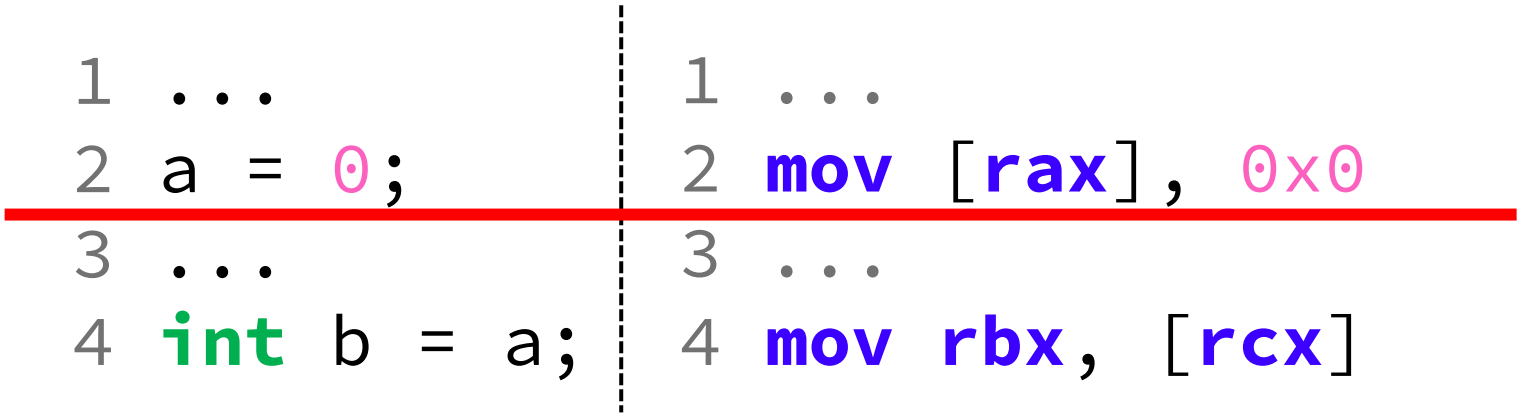
- Determine data dependence between instructions in binary executables

```
1 ...
2 a = 0;
3 ...
4 int b = a;
```

```
1 ...
2 mov [rax], 0x0
3 ...
4 mov rbx, [rcx]
```

# Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables



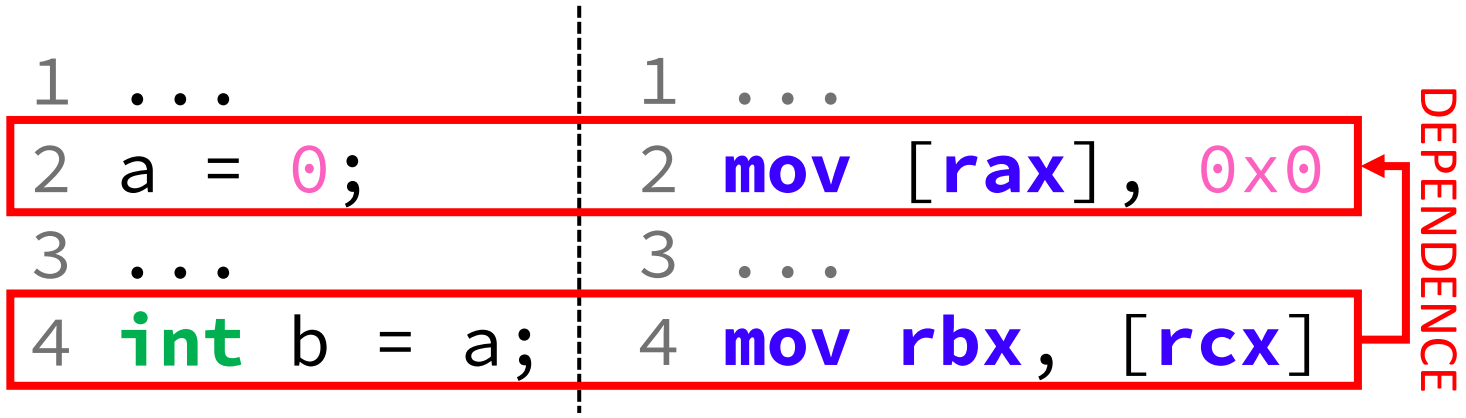
# Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables

```
1 ...                    1 ...
2 a = 0;                 2 mov [rax], 0x0
3 ...                    3 ...
4 int b = a;             4 mov rbx, [rcx]
```

# Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables



# Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables
- Have many applications
  - Precise call graph construction
  - Malware analysis to expose hidden behaviors
  - Binary rewriting
  - ……

# Intro: Binary Program Dependence Analysis

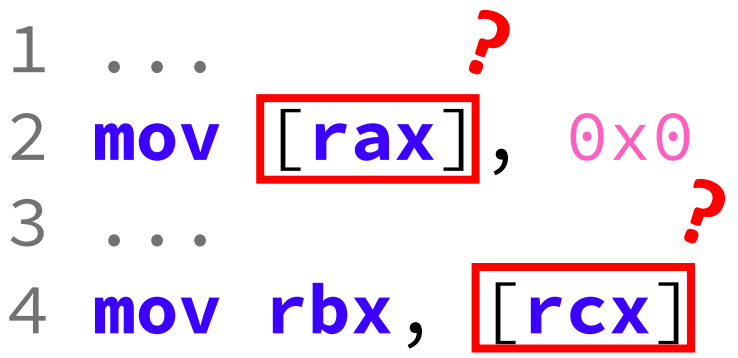
- Determine data dependence between instructions in binary executables
- Have many applications
- A key challenge is to identify if multiple memory read/write instructions access the same memory location

```
1 ...
2 a = 0;
3 ...
4 int b = a;
```

```
1 ...          ?
2 mov [rax], 0x0
3 ...                ?
4 mov rbx, [rcx]
```

# Intro: Binary Program Dependence Analysis

- Determine data dependence between instructions in binary executables
- Have many applications
- A key challenge is to identify if multiple memory read/write instructions access the same memory location



# Intro: Existing Works

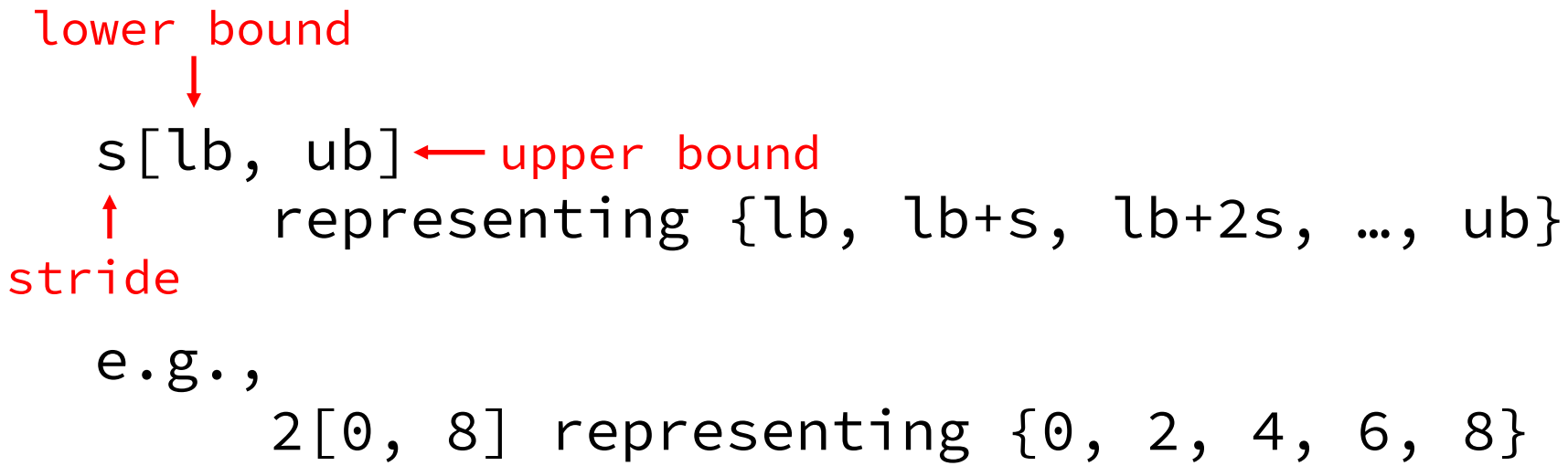
- The state-of-the-art: Value Set Analysis (VSA)
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction

# Intro: Existing Works

- The state-of-the-art: Value Set Analysis (VSA)
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a ***strided interval*** to denote the set of values

# Intro: Existing Works

- The state-of-the-art: Value Set Analysis (VSA)
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a ***strided interval*** to denote the set of values



# Intro: Existing Works

- The state-of-the-art: Value Set Analysis (VSA)
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a ***strided interval*** to denote the set of values
  - Have difficulty scaling to complex programs

```
1 int *p;          1 ...
2  if (...)        2 ...
3    p = ...;      3 mov rax, rbx
4  else            4 ...
5    p = ...;      5 mov rax, [rcx]
6 *p = 0;          6 mov [rax], 0
```

# Intro: Existing Works

- The state-of-the-art: Value Set Analysis (VSA)
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a ***strided interval*** to denote the set of values
  - Have difficulty scaling to complex programs

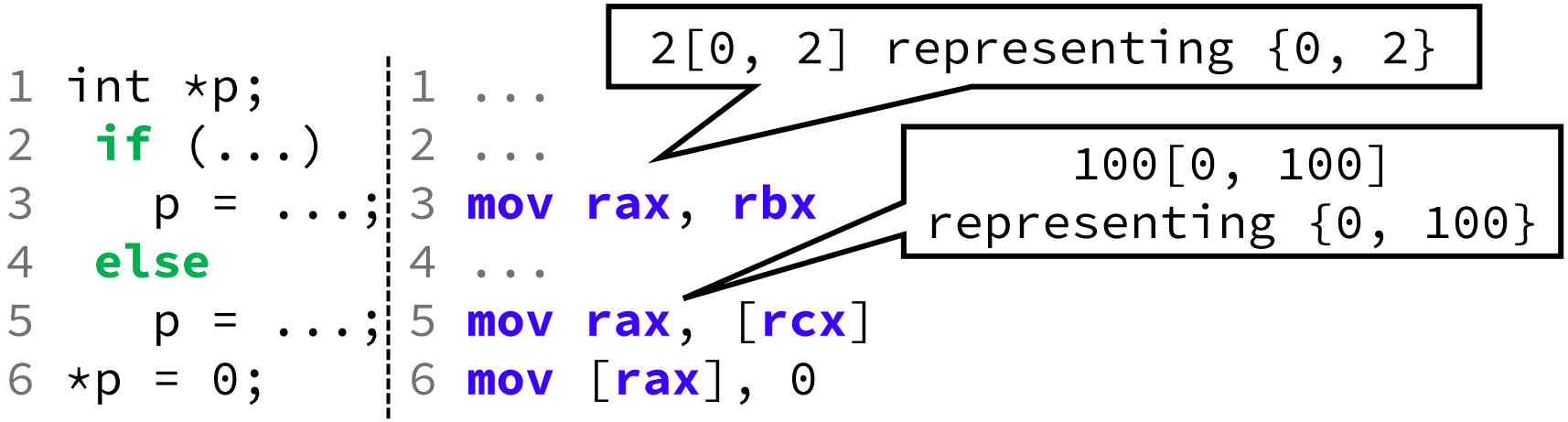
```
1 int *p;
2  if (...)
3    p = ...;
4  else
5    p = ...;
6 *p = 0;
```

```
1 ...
2 ...
3 mov rax, rbx
4 ...
5 mov rax, [rcx]
6 mov [rax], 0
```

2[0, 2] representing {0, 2}

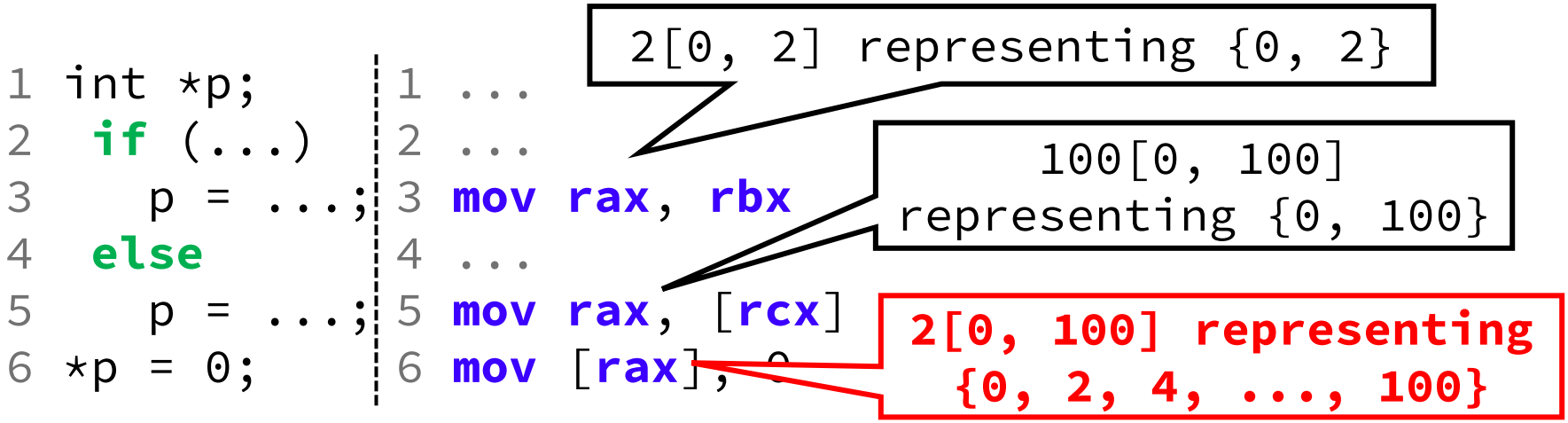
# Intro: Existing Works

- The state-of-the-art: Value Set Analysis (VSA)
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a ***strided interval*** to denote the set of values
  - Have difficulty scaling to complex programs



# Intro: Existing Works

- The state-of-the-art: Value Set Analysis (VSA)
  - Integrated into a variety of binary analysis frameworks (Angr, BAP)
  - Compute a set of possible values for each operand of an instruction
  - Use a ***strided interval*** to denote the set of values
  - Have difficulty scaling to complex programs



# Intro: Existing Works

- The state-of-the-art: Value Set Analysis (VSA)

| | ANGR-VSA | BAP-VSA |
|---|---|---|
| 164.gzip | ✗ | **TIMEOUT (12 hours)** |
| 175.vpr | ✗ | **TIMEOUT (12 hours)** |
| 176.gcc | ✗ | **TIMEOUT (12 hours)** |
| 181.mcf | ✗ | ✓ |
| 186.crafty | ✗ | **TIMEOUT (12 hours)** |
| 197.parser | ✗ | **TIMEOUT (12 hours)** |
| 252.eon | ✗ | **TIMEOUT (12 hours)** |
| 253.perlbmk | ✗ | **TIMEOUT (12 hours)** |
| 254.gap | ✗ | **TIMEOUT (12 hours)** |
| 255.vortex | ✗ | **TIMEOUT (12 hours)** |
| 256.bzip2 | ✗ | **TIMEOUT (12 hours)** |
| 300.twolf | ✗ | **TIMEOUT (12 hours)** |

# Intro: Existing Works

- The state-of-the-art: Value Set Analysis (VSA)
- More efficient but conservative technique: ALTO

# Observation

- Observation 1: probabilistic guarantees are sufficient for many practical applications

# Observation

has a very low likelihood of missing any true positive

- Observation 1: probabilistic guarantees are sufficient for many practical applications
  - E.g., indirect control-flow transfer targets can help construct precise call graphs

# Observation

has a very low likelihood of missing any true positive

- Observation 1: probabilistic guarantees are sufficient for many practical applications
  - E.g., indirect control-flow transfer targets can help construct precise call graphs

Strict Soundness

→ Never miss any true positives

→ Produce a large number of bogus call edges

Probability Guarantees

→ Discover most of the true edges

→ Have a low chance of missing some true positive edges

# Observation

- Observation 1: probabilistic guarantees are sufficient for many practical applications

- Observation 2: a dependence relation can be disclosed by many whole-program paths

# Observation

- Observation 1: probabilistic guarantees are sufficient for many practical applications

- Observation 2: a dependence relation can be disclosed by many whole-program paths
  - For a program with n statements
    - The number of dependences: $O(n^2)$
    - The number of paths is $\mathbf{O(2^n)}$

# Observation

- Observation 1: probabilistic guarantees are sufficient for many practical applications

- Observation 2: a dependence relation can be disclosed by many whole-program paths

→ BDA: a sampling-based abstract interpretation technique for dependence analysis

# Observation

- Observation 1: probabilistic guarantees are sufficient for many practical applications

- Observation 2: a dependence relation can be disclosed by many whole-program paths

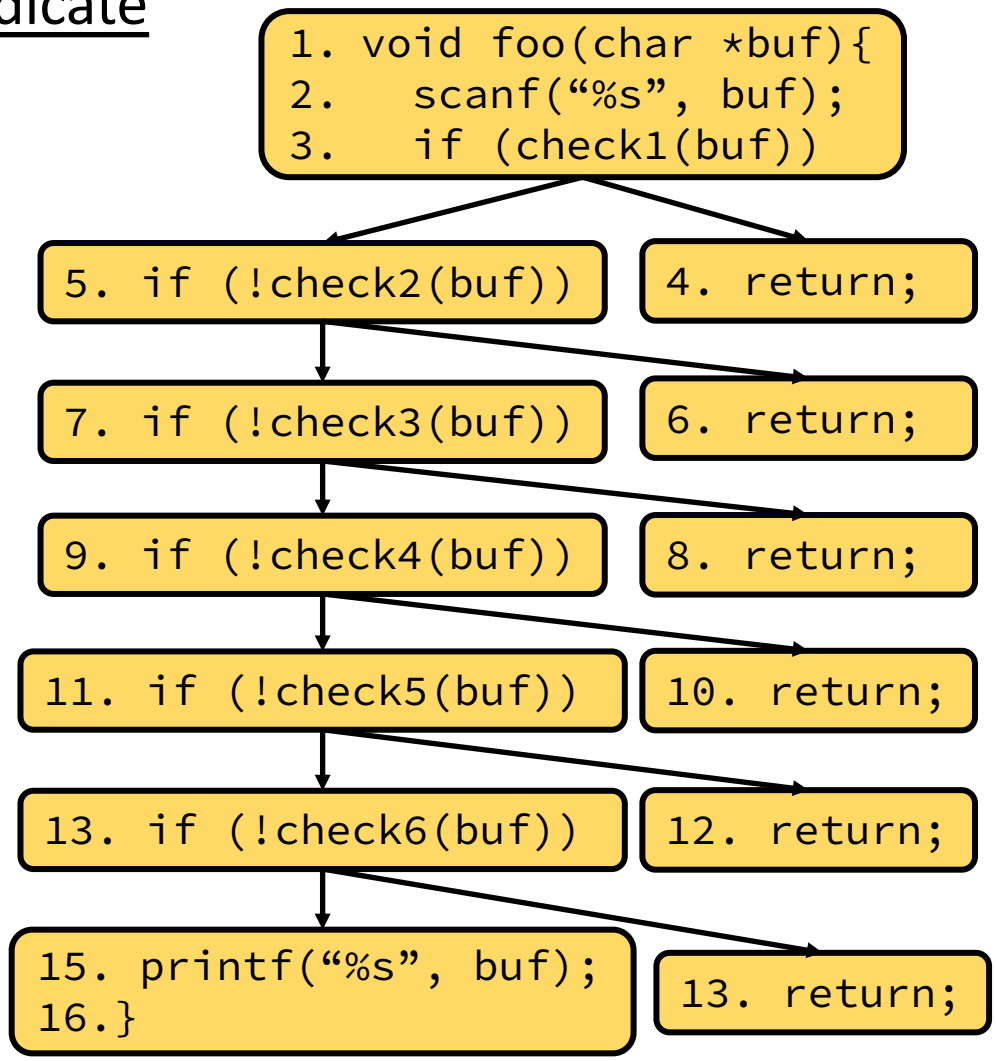
→ BDA: a sampling-based abstract interpretation technique for dependence analysis

We will use source code examples to explain our idea.
But BDA operates on stripped binary executables.

# Naïve Sampling Algorithm?

To toss a fair coin at each predicate

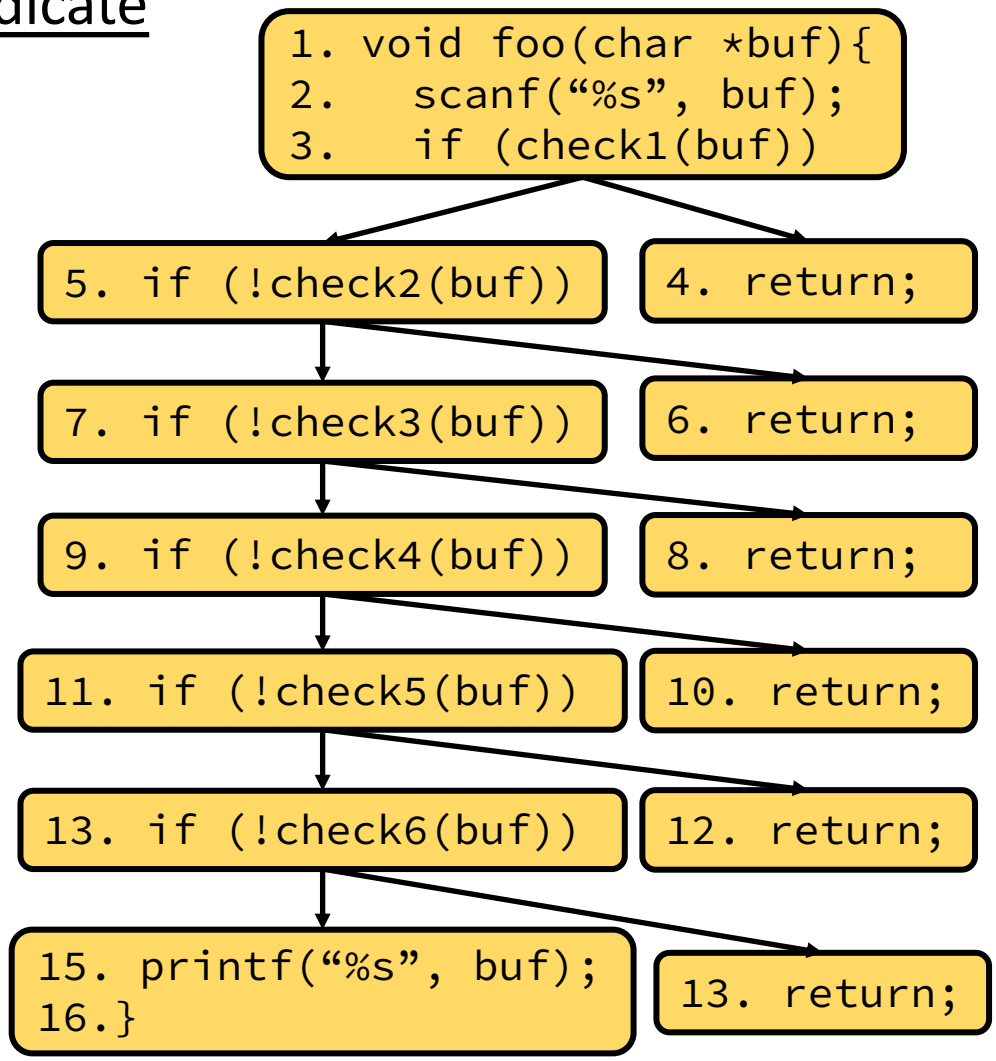
```
1.  void foo(char *buf){
2.    scanf("%s", buf);
3.    if (!check1(buf))
4.      return;
5.    if (!check2(buf))
6.      return;
7.    if (!check3(buf))
8.      return;
9.    if (!check4(buf))
10.     return;
11.   if (!check5(buf))
12.     return;
13.   if (!check6(buf))
14.     return;
15.   printf("%s", buf);
16.}
```



# Naïve Sampling Algorithm?

To toss a fair coin at each predicate

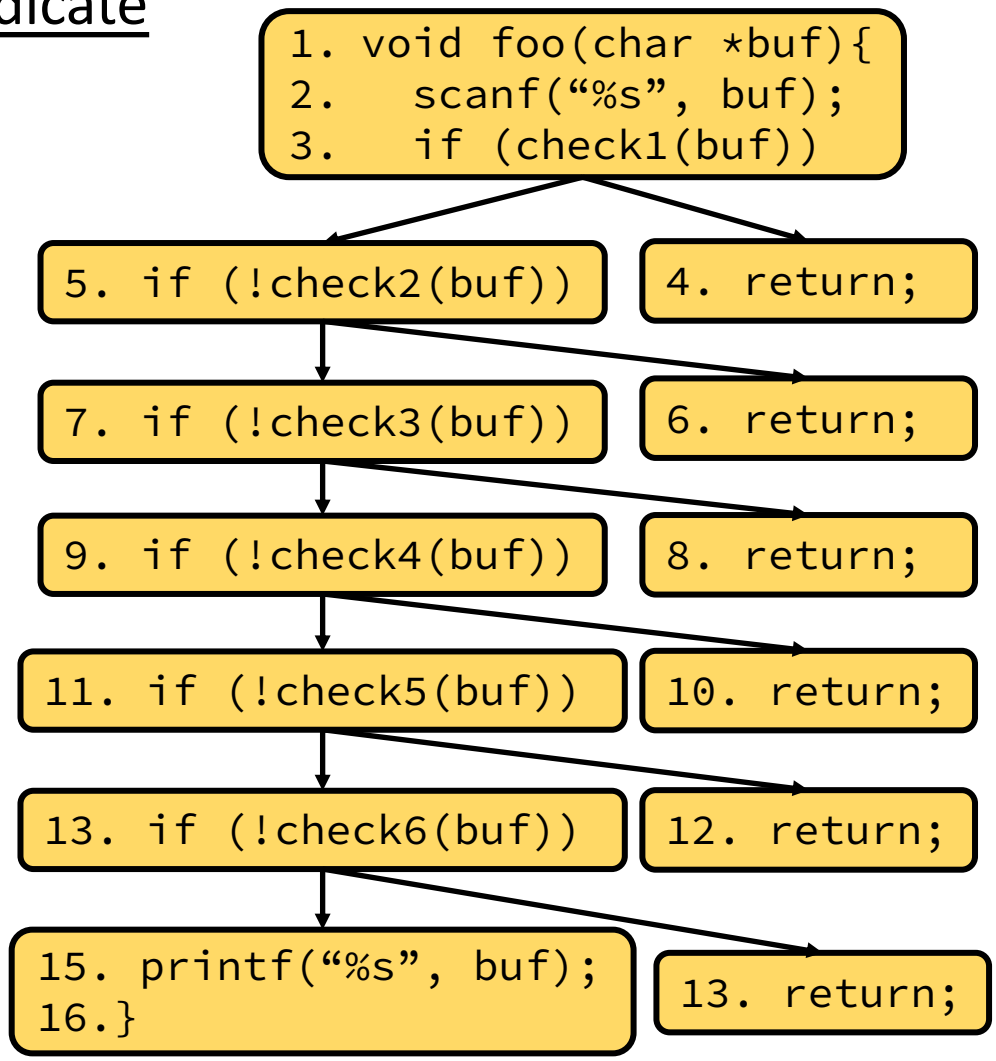
```
1.  void foo(char *buf){
2.    scanf("%s", buf);
3.    if (!check1(buf))
4.      return;
5.    if (!check2(buf))
6.      return;
7.    if (!check3(buf))
8.      return;
9.    if (!check4(buf))
10.     return;
11.   if (!check5(buf))
12.     return;
13.   if (!check6(buf))
14.     return;
15.   printf("%s", buf);
16.}
```



# Naïve Sampling Algorithm?

To toss a fair coin at each predicate

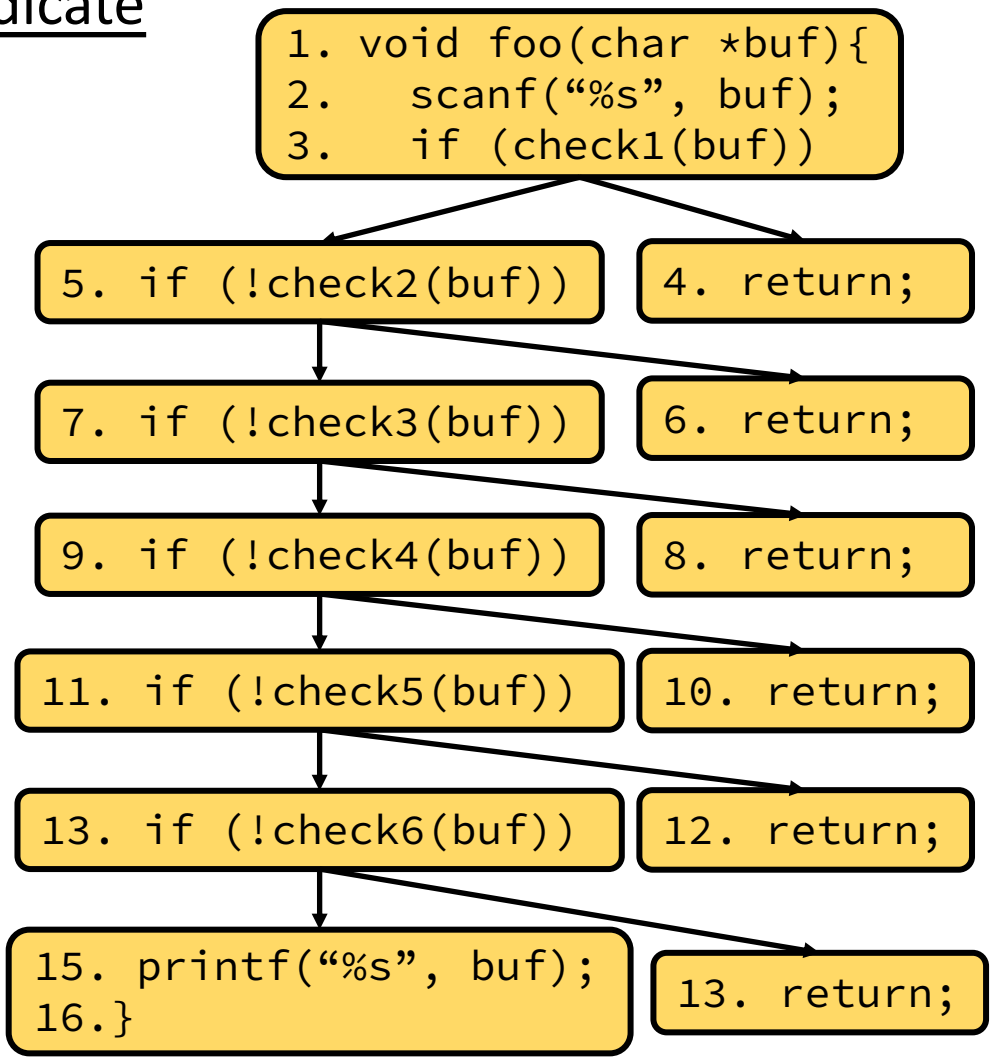
```
1. void foo(char *buf){
2.   scanf("%s", buf);
3.   if (!check1(buf))
4.     return;
5.   if (!check2(buf))
6.     return;
7.   if (!check3(buf))
8.     return;
9.   if (!check4(buf))
10.    return;
11.  if (!check5(buf))
12.    return;
13.  if (!check6(buf))
14.    return;
15.  printf("%s", buf);
16.}
```



# Naïve Sampling Algorithm?

To toss a fair coin at each predicate

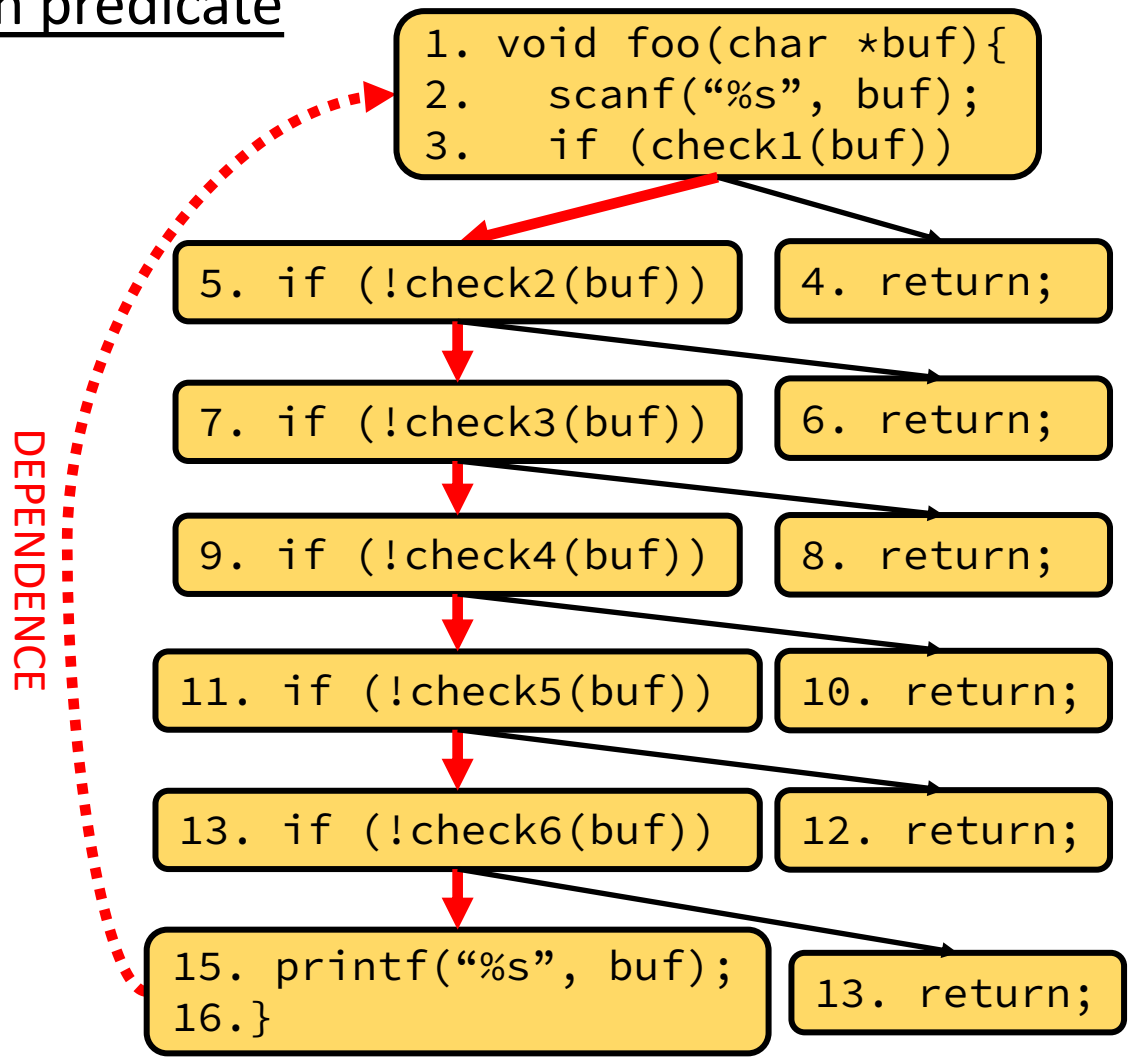
```
1.  void foo(char *buf){
2.    scanf("%s", buf);
3.    if (!check1(buf))
4.      return;
5.    if (!check2(buf))
6.      return;
7.    if (!check3(buf))
8.      return;
9.    if (!check4(buf))
10.     return;
11.   if (!check5(buf))
12.     return;
13.   if (!check6(buf))
14.     return;
15.   printf("%s", buf);
16.}
```



# Naïve Sampling Algorithm?

To toss a fair coin at each predicate

```
1.  void foo(char *buf){
2.    scanf("%s", buf);
3.    if (!check1(buf))
4.      return;
5.    if (!check2(buf))
6.      return;
7.    if (!check3(buf))
8.      return;
9.    if (!check4(buf))
10.     return;
11.   if (!check5(buf))
12.     return;
13.   if (!check6(buf))
14.     return;
15.   printf("%s", buf);
16.}
```



# Naïve Sampling Algorithm? *NO*

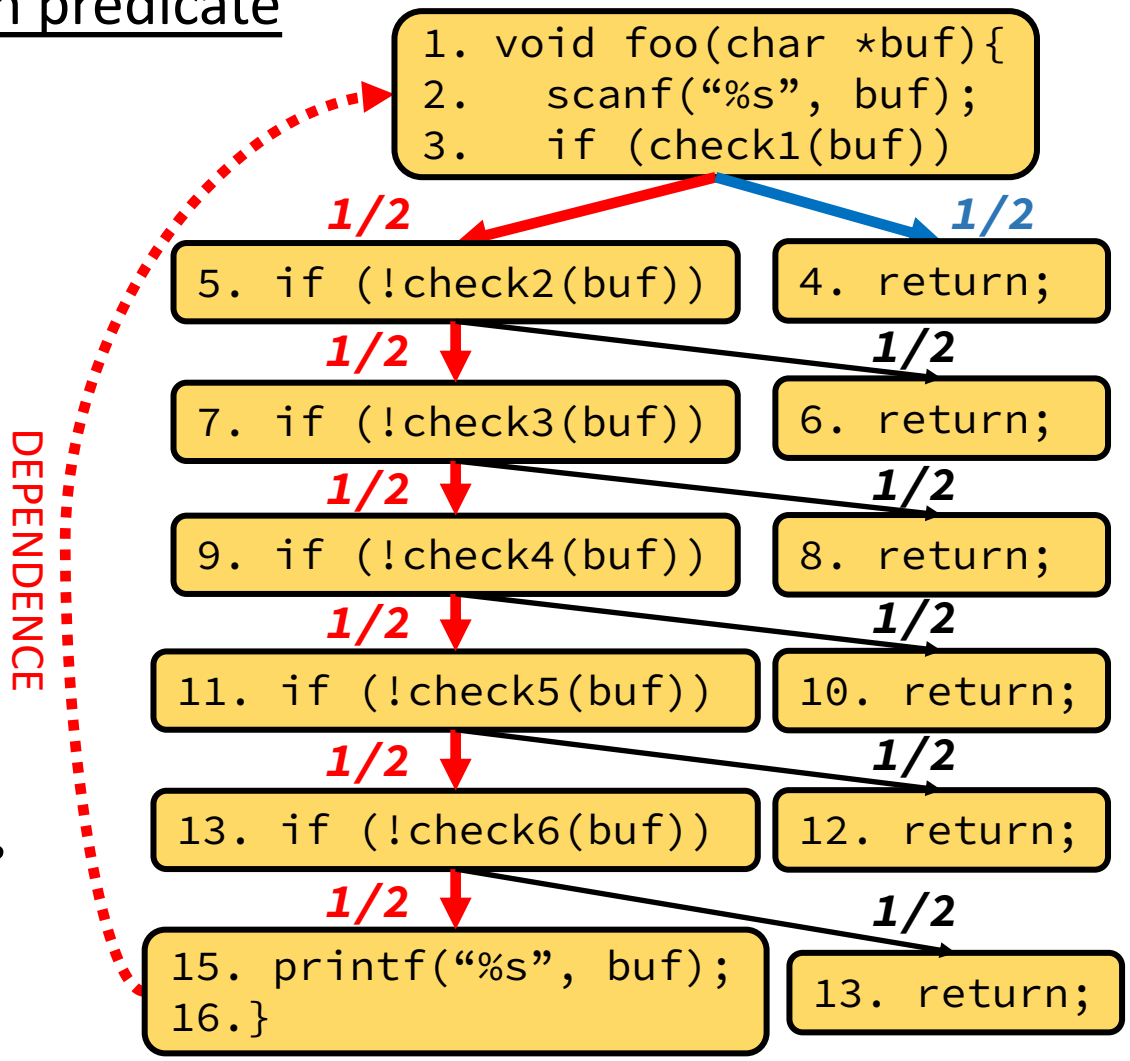
To toss a fair coin at each predicate

P(**Red Path**): ***1/64***

*Find Dependence*

P(**Blue Path**): ***1/2***

*Cannot Find Dependence*



# Naïve Sampling Algorithm? *NO*

To toss a fair coin at each predicate

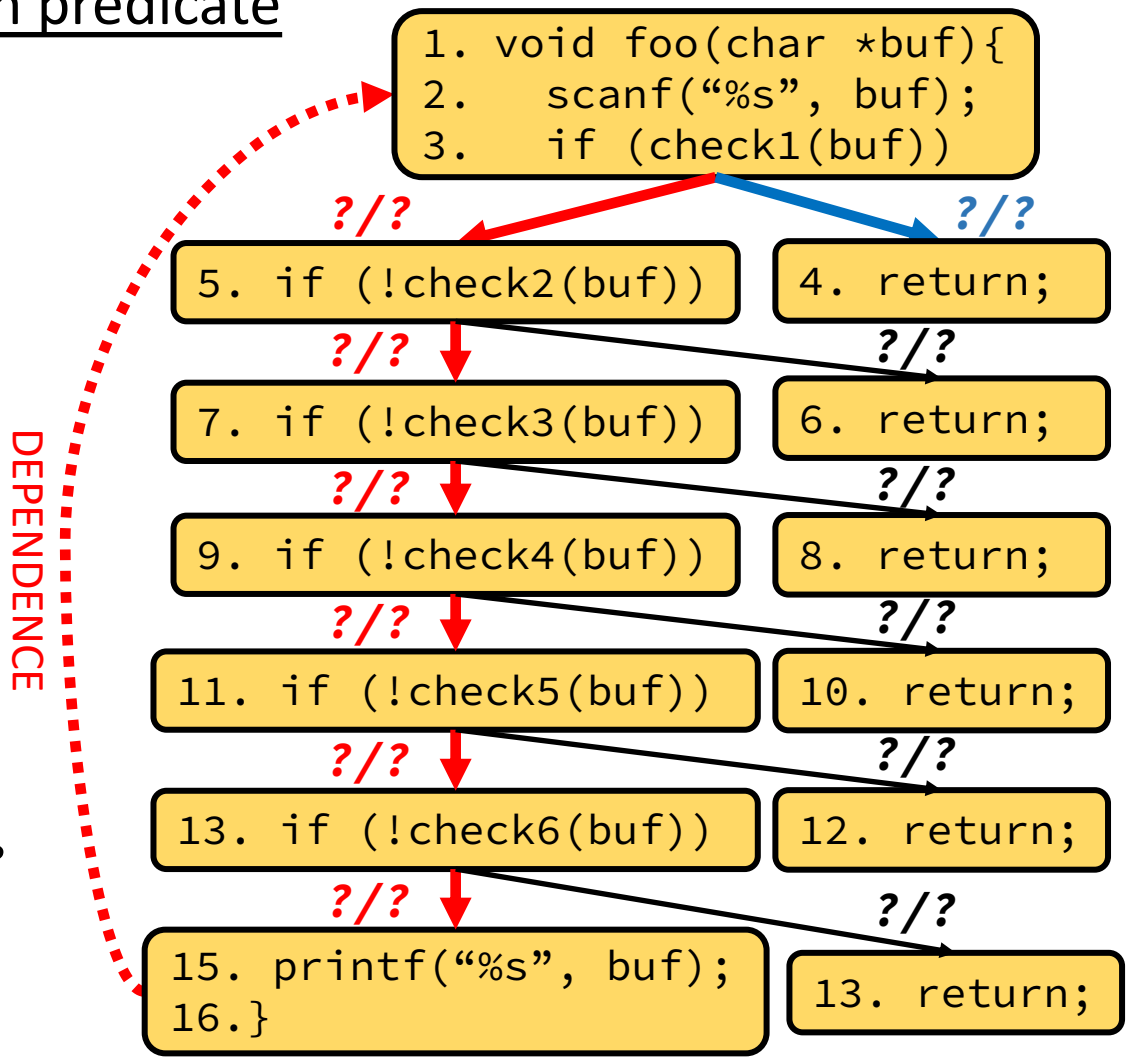
**7 paths in total**

P(**Red Path**): ***1/7***

*Find Dependence*

P(**Blue Path**): ***1/7***

*Cannot Find Dependence*



# Workflow of BDA

- Phase 1: Path Sampling

  Sample whole-program paths under a ***uniform distribution***

- Phase 2: Per-path Abstract Interpretation

  Compute the possible values for individual instructions, ***following the given sample path***

- Phase 3: Posterior Analysis

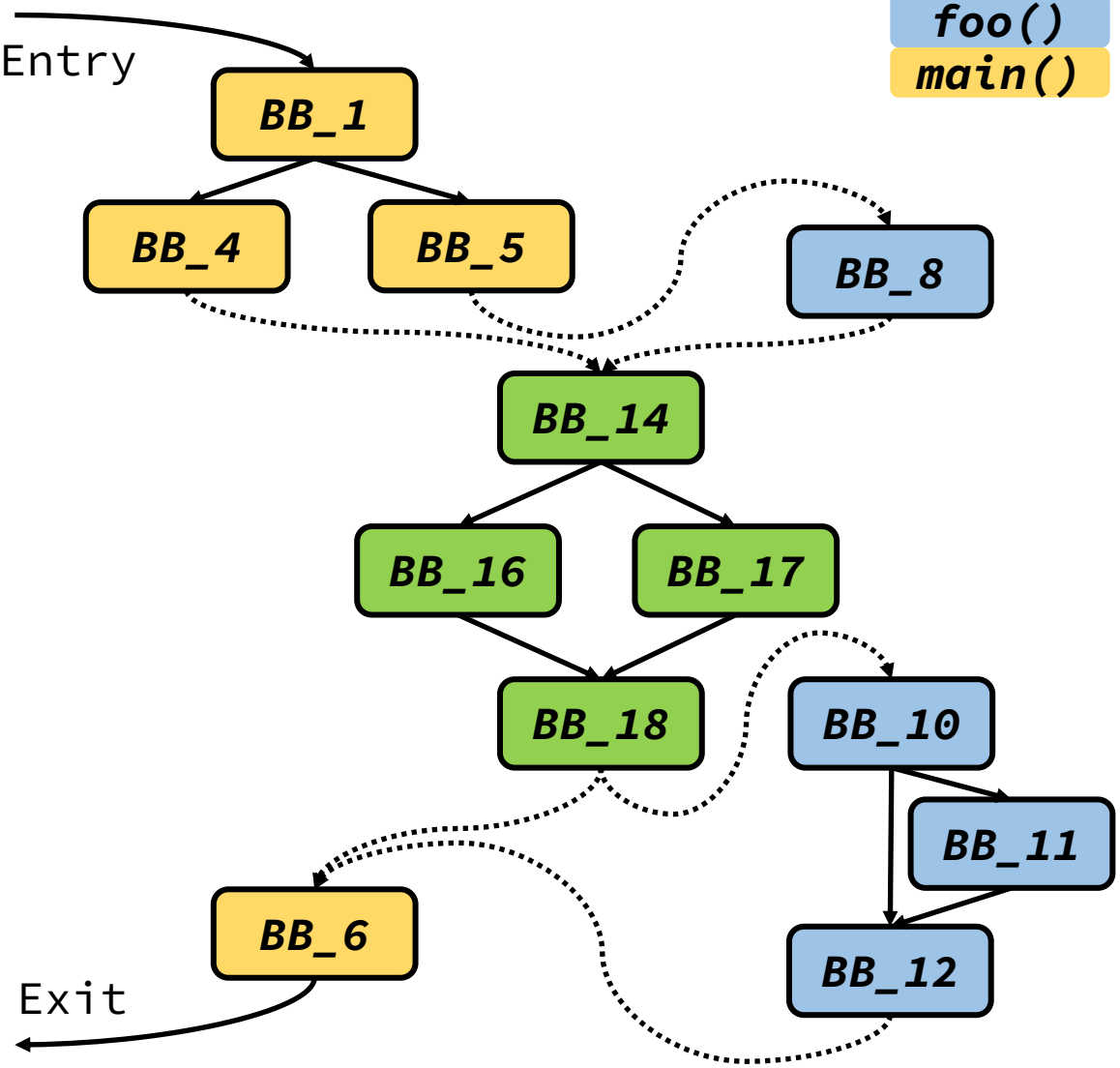
  Mitigate the possible ***incomplete path coverage*** during sampling

# Phase 1: Path Sampling

- Input: Binary executable and its inter-procedural control flow graph
- Output: A number of whole-program path samples

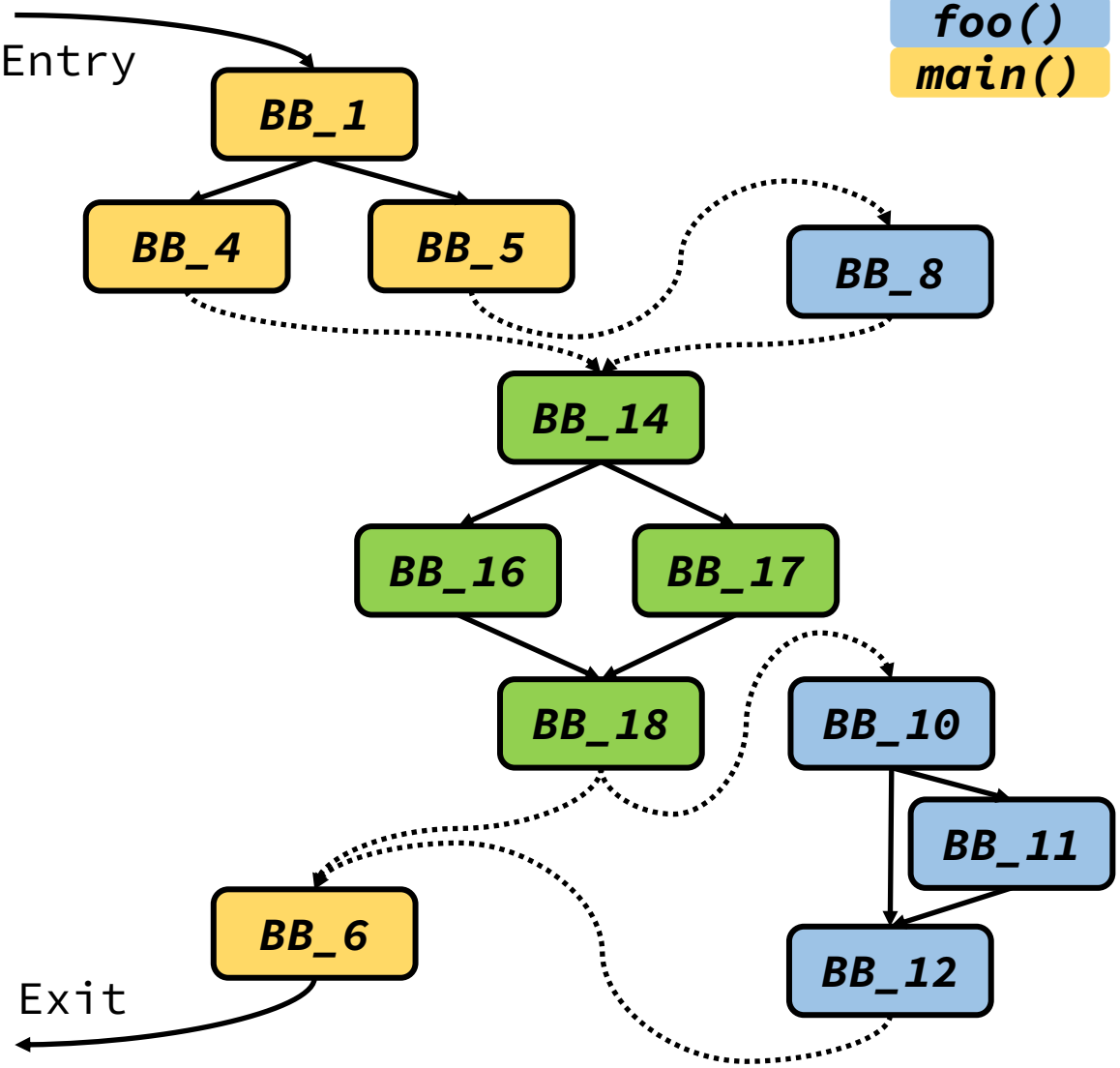
# Phase 1: Inter-procedural Control Flow Graph

```
1.  int main(){
2.    int a;
3.    if (rand())
4.      gee(&a);
5.    else foo(&a) ;
6.  }
7.
8.  void foo(int *a){
9.    gee(a) ;
10.   if (rand())
11.     *a+=1;
12. }
13.
14. void gee(int *a){
15.   if (rand())
16.     *a=0;
17.   else *a=2;
18. }
```



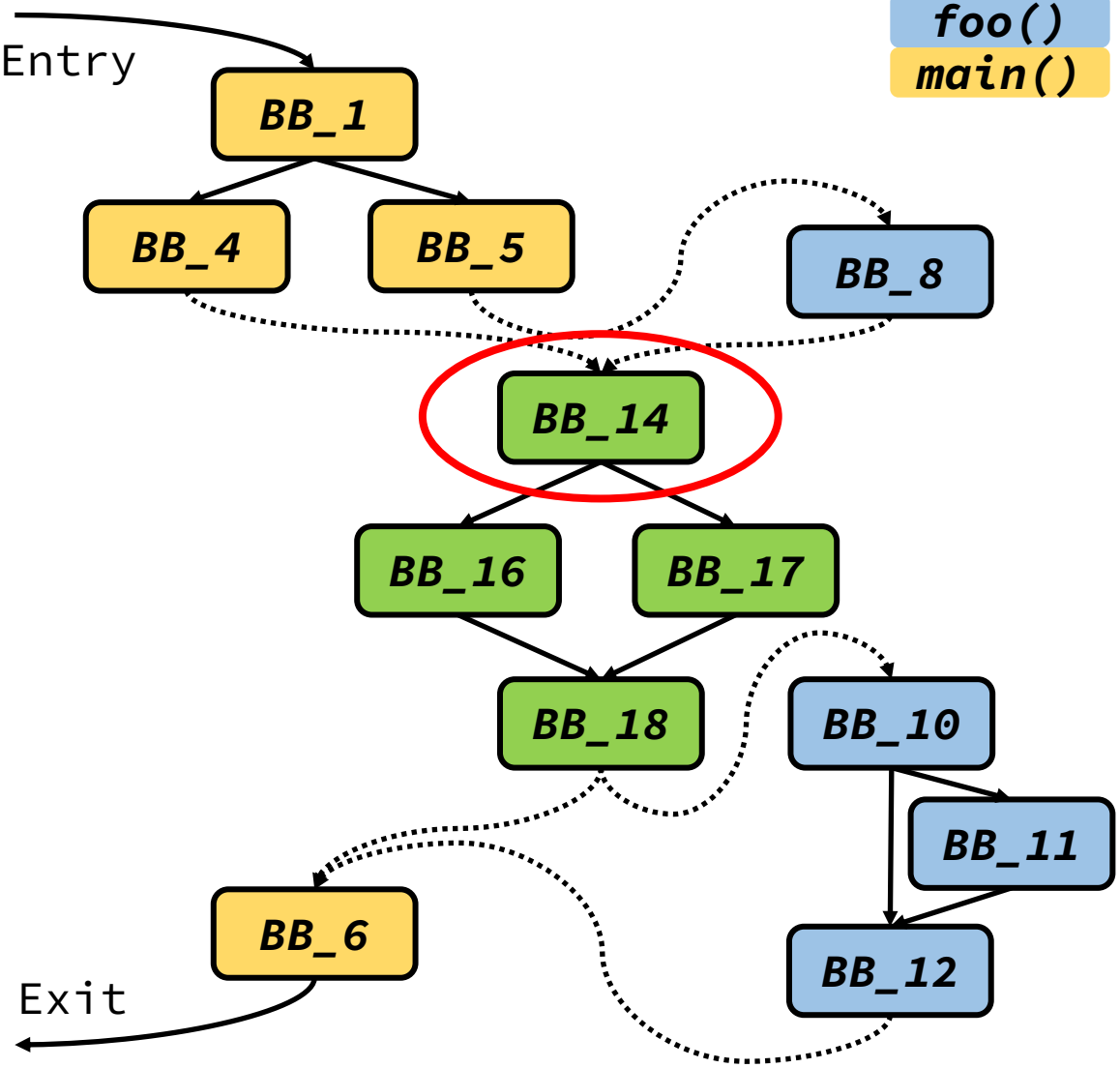
# Phase 1: Inter-procedural Control Flow Graph

```
1.  int main(){
2.    int a;
3.    if (rand())
4.      gee(&a);
5.    else foo(&a) ;
6.  }
7.
8.  void foo(int *a){
9.    gee(a) ;
10.   if (rand())
11.     *a+=1;
12. }
13.
14. void gee(int *a){
15.   if (rand())
16.     *a=0;
17.   else *a=2;
18. }
```



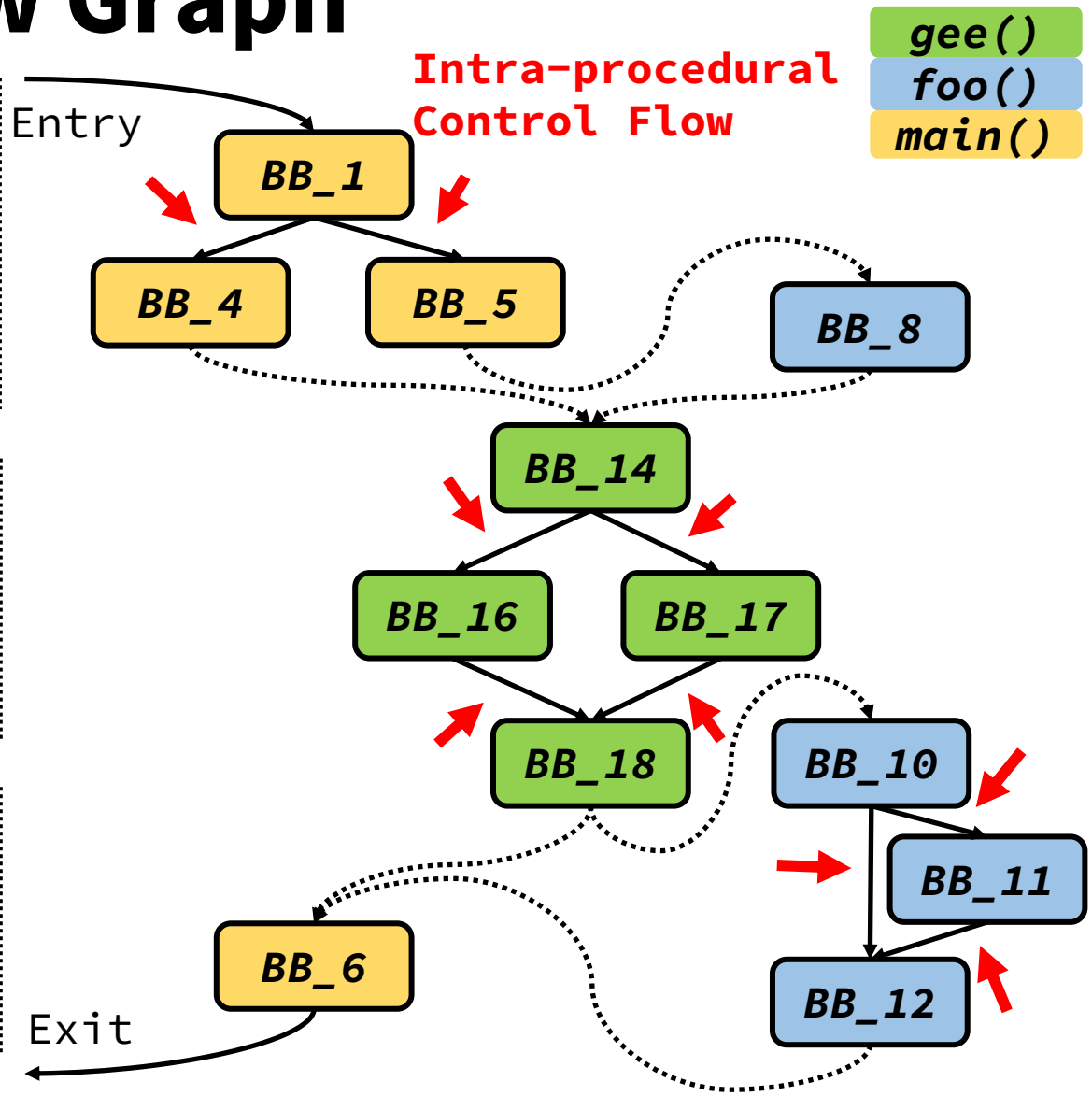
# Phase 1: Inter-procedural Control Flow Graph

```
1.  int main(){
2.    int a;
3.    if (rand())
4.      gee(&a);
5.    else foo(&a) ;
6.  }
7.
8.  void foo(int *a){
9.    gee(a) ;
10.   if (rand())
11.     *a+=1;
12. }
13.
14. void gee(int *a){
15.   if (rand())
16.     *a=0;
17.   else *a=2;
18. }
```



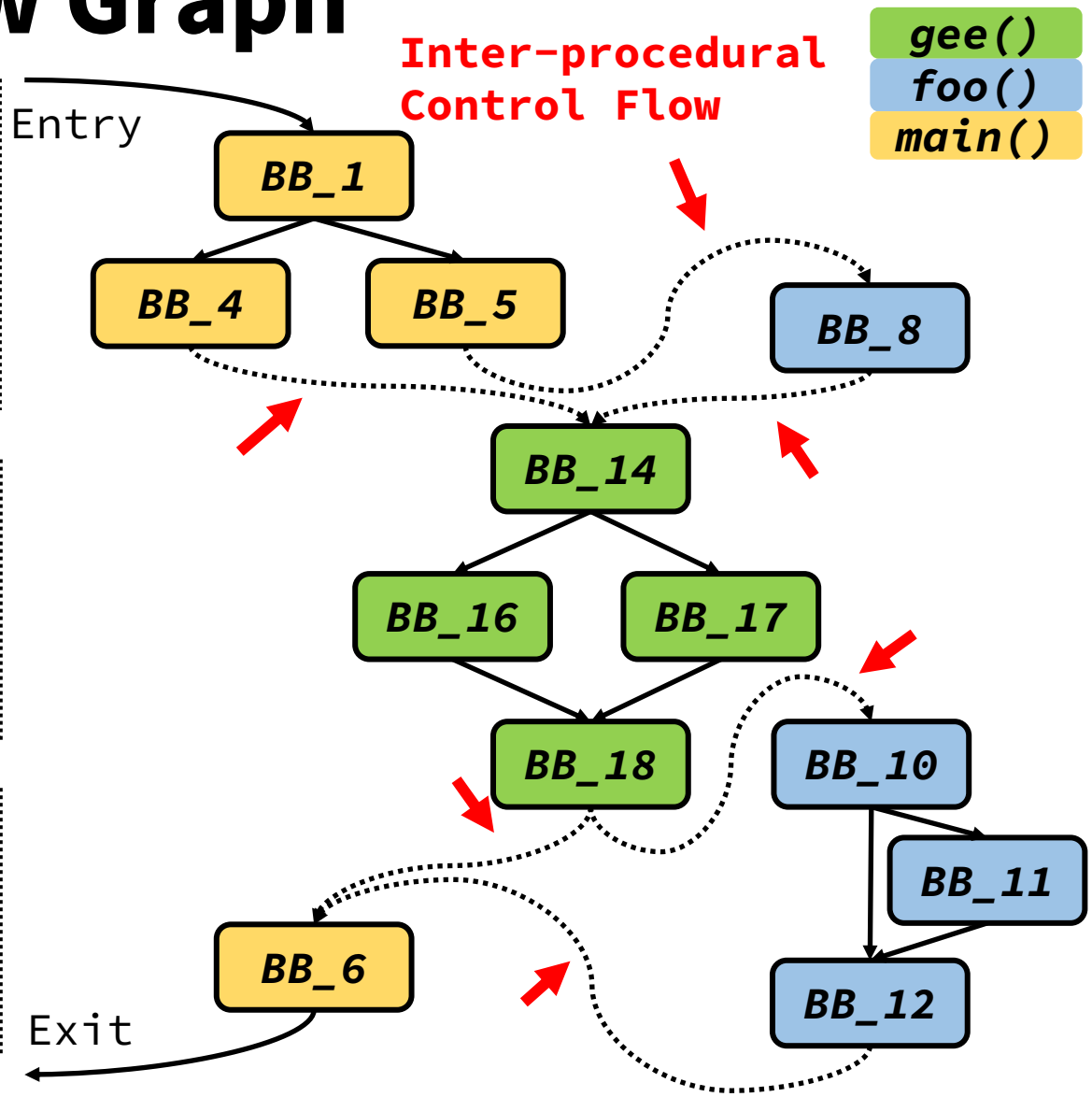
# Phase 1: Inter-procedural Control Flow Graph

```
1.  int main(){
2.    int a;
3.    if (rand())
4.      gee(&a);
5.    else foo(&a) ;
6.  }
7.
8.  void foo(int *a){
9.    gee(a) ;
10.   if (rand())
11.     *a+=1;
12. }
13.
14. void gee(int *a){
15.   if (rand())
16.     *a=0;
17.   else *a=2;
18. }
```



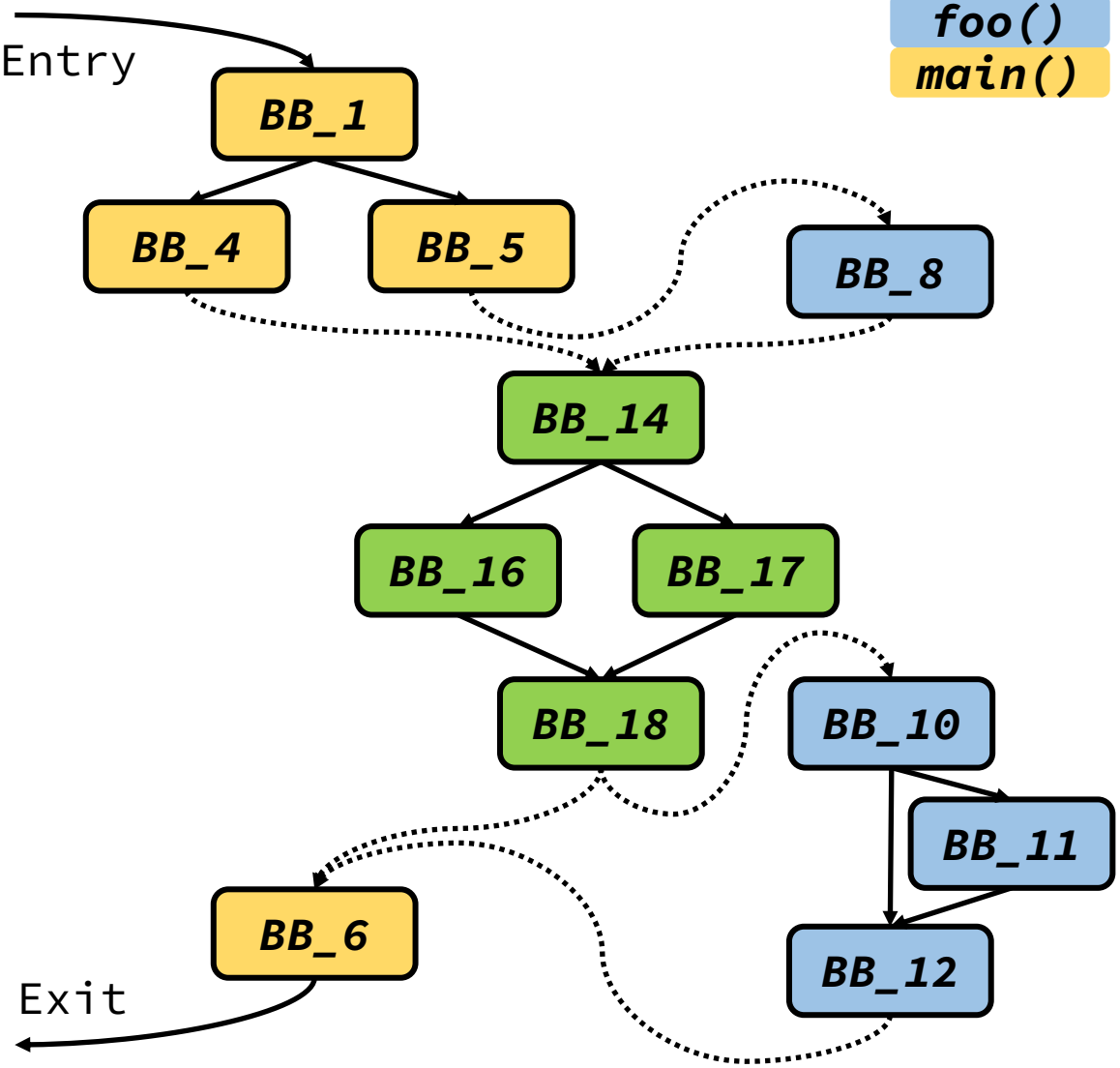
# Phase 1: Inter-procedural Control Flow Graph

```
1.  int main(){
2.    int a;
3.    if (rand())
4.      gee(&a);
5.    else foo(&a) ;
6.  }
7.
8.  void foo(int *a){
9.    gee(a) ;
10.   if (rand())
11.     *a+=1;
12. }
13.
14. void gee(int *a){
15.   if (rand())
16.     *a=0;
17.   else *a=2;
18. }
```



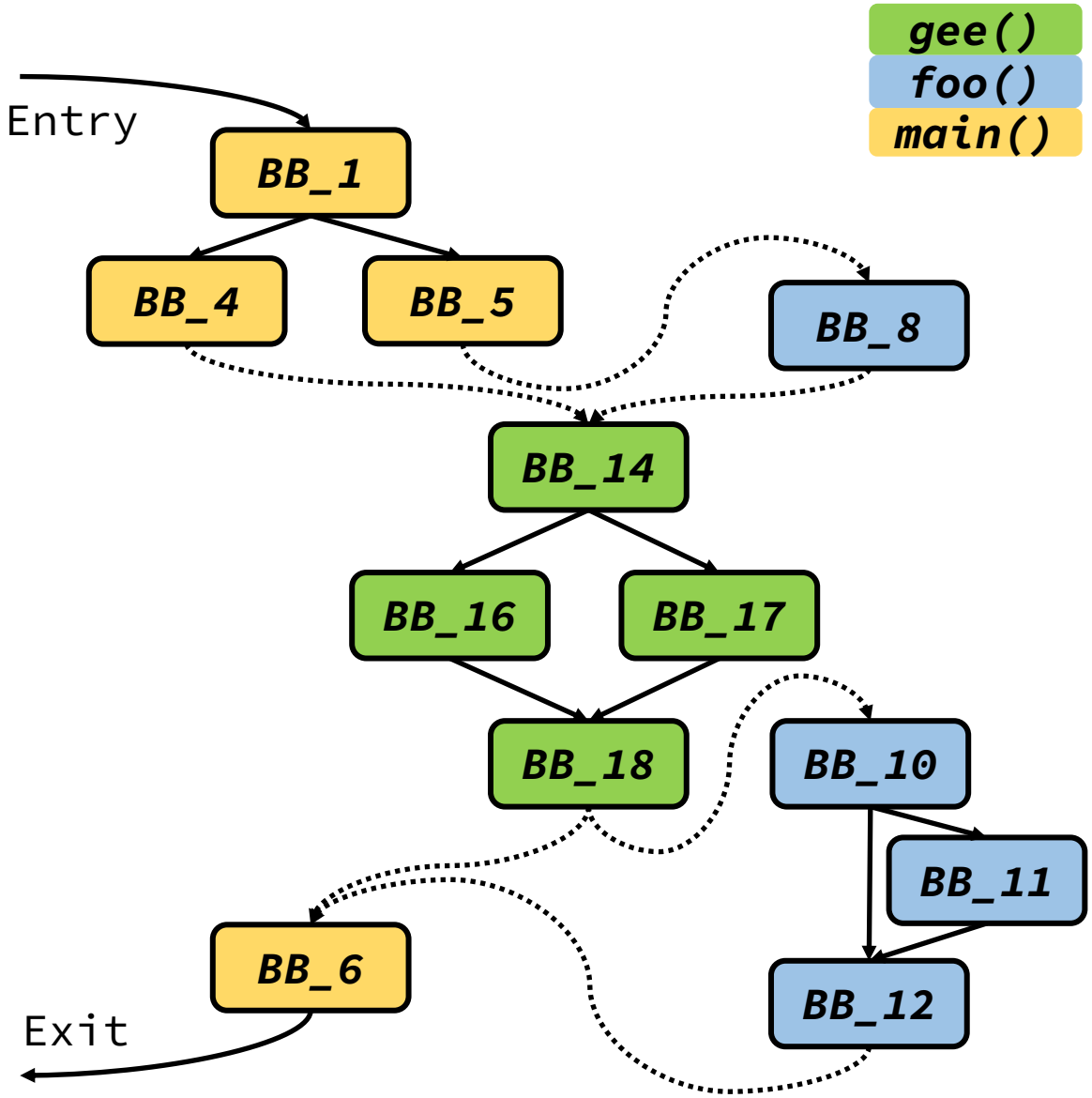
# Phase 1: Inter-procedural Control Flow Graph

```
1.  int main(){
2.    int a;
3.    if (rand())
4.      gee(&a);
5.    else foo(&a) ;
6.  }
7.
8.  void foo(int *a){
9.    gee(a) ;
10.   if (rand())
11.     *a+=1;
12. }
13.
14. void gee(int *a){
15.   if (rand())
16.     *a=0;
17.   else *a=2;
18. }
```



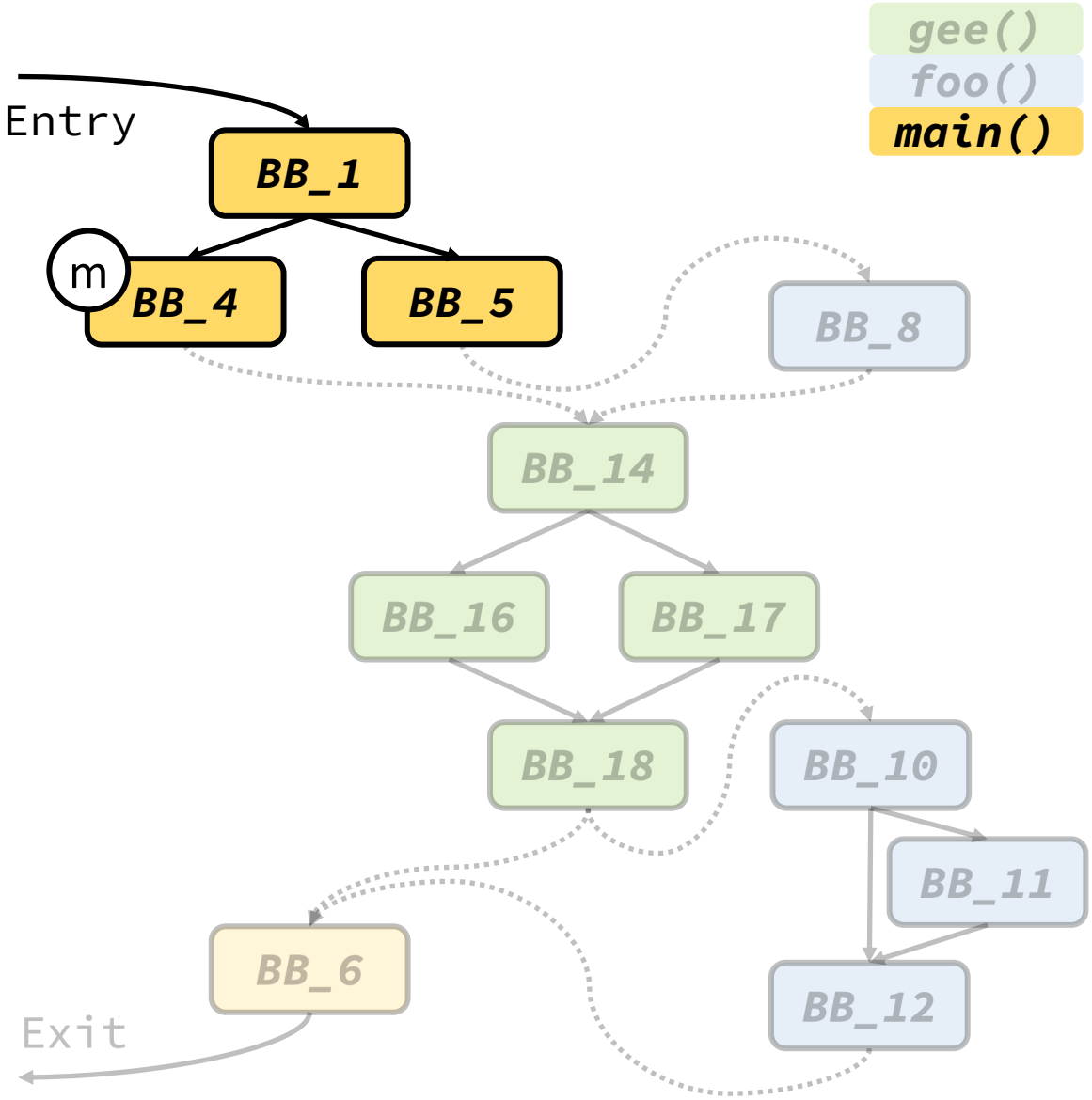
# Phase 1: Path Counting

Toss a ***biased*** coin at predicates to sample each path uniformly



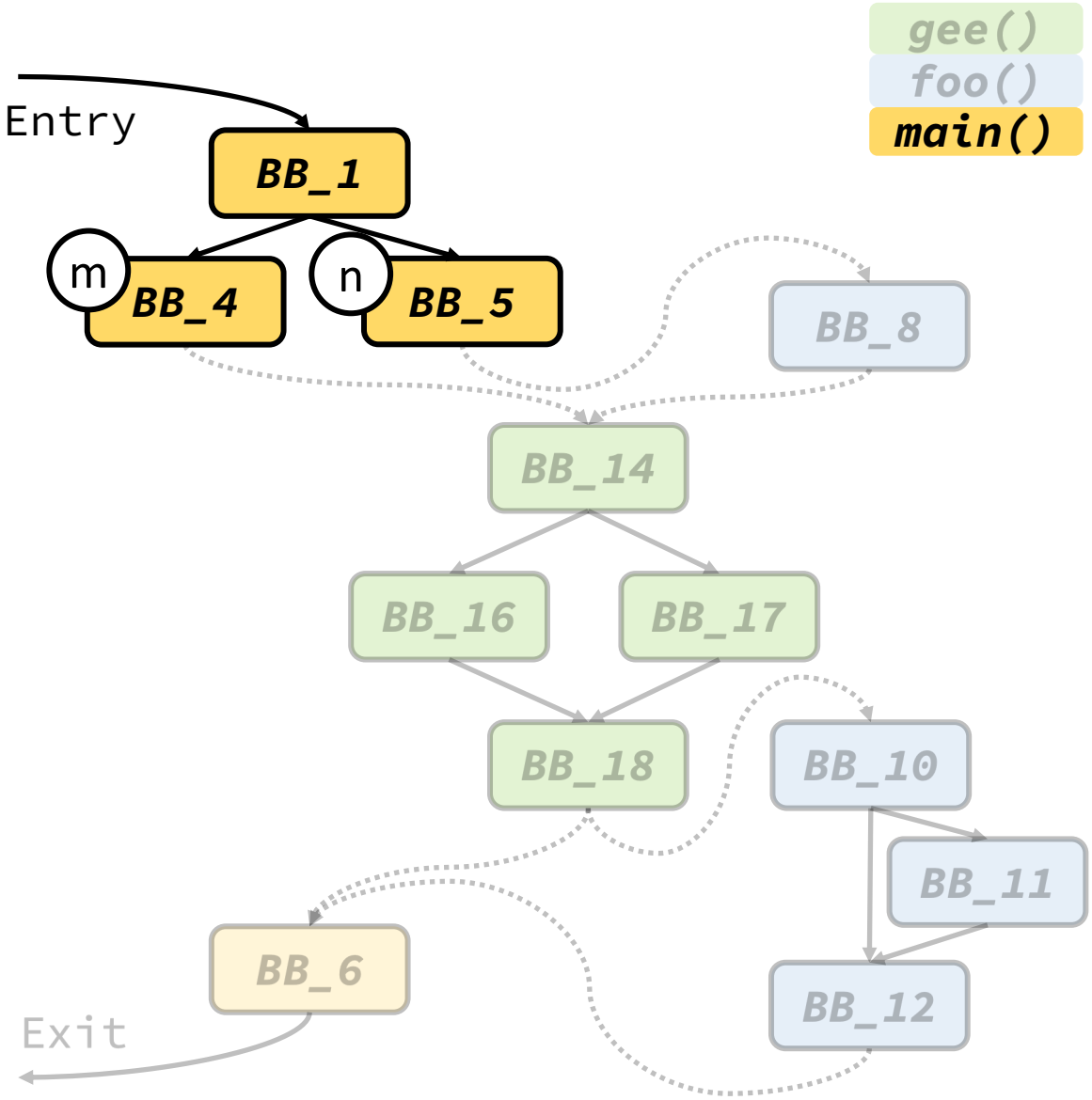
# Phase 1: Path Counting

- ***m*** inter-procedural paths from BB_4 to BB_6



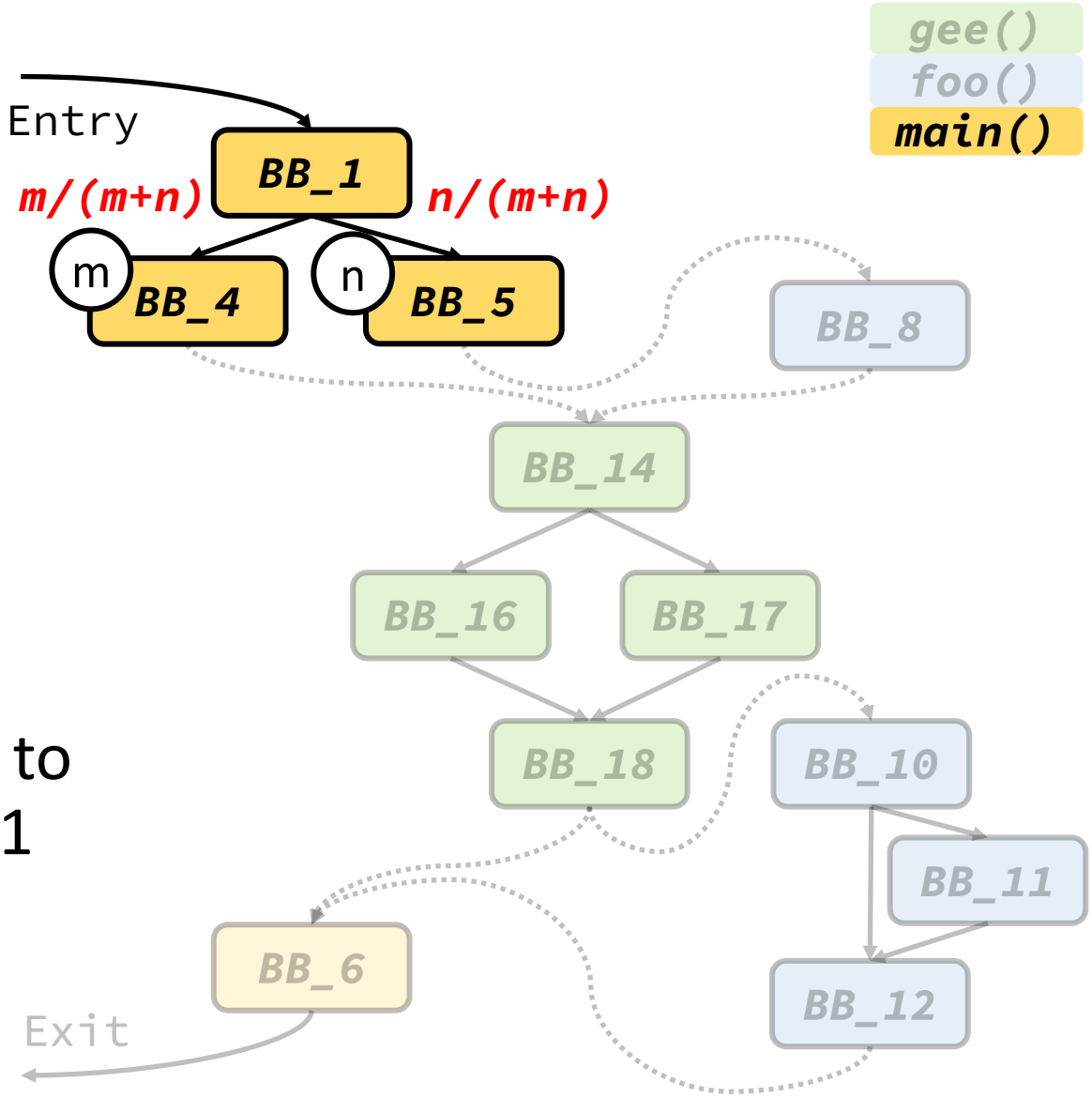
# Phase 1: Path Counting

- ***m*** inter-procedural paths from BB_4 to BB_6
- ***n*** inter-procedural paths from BB_5 to BB_6



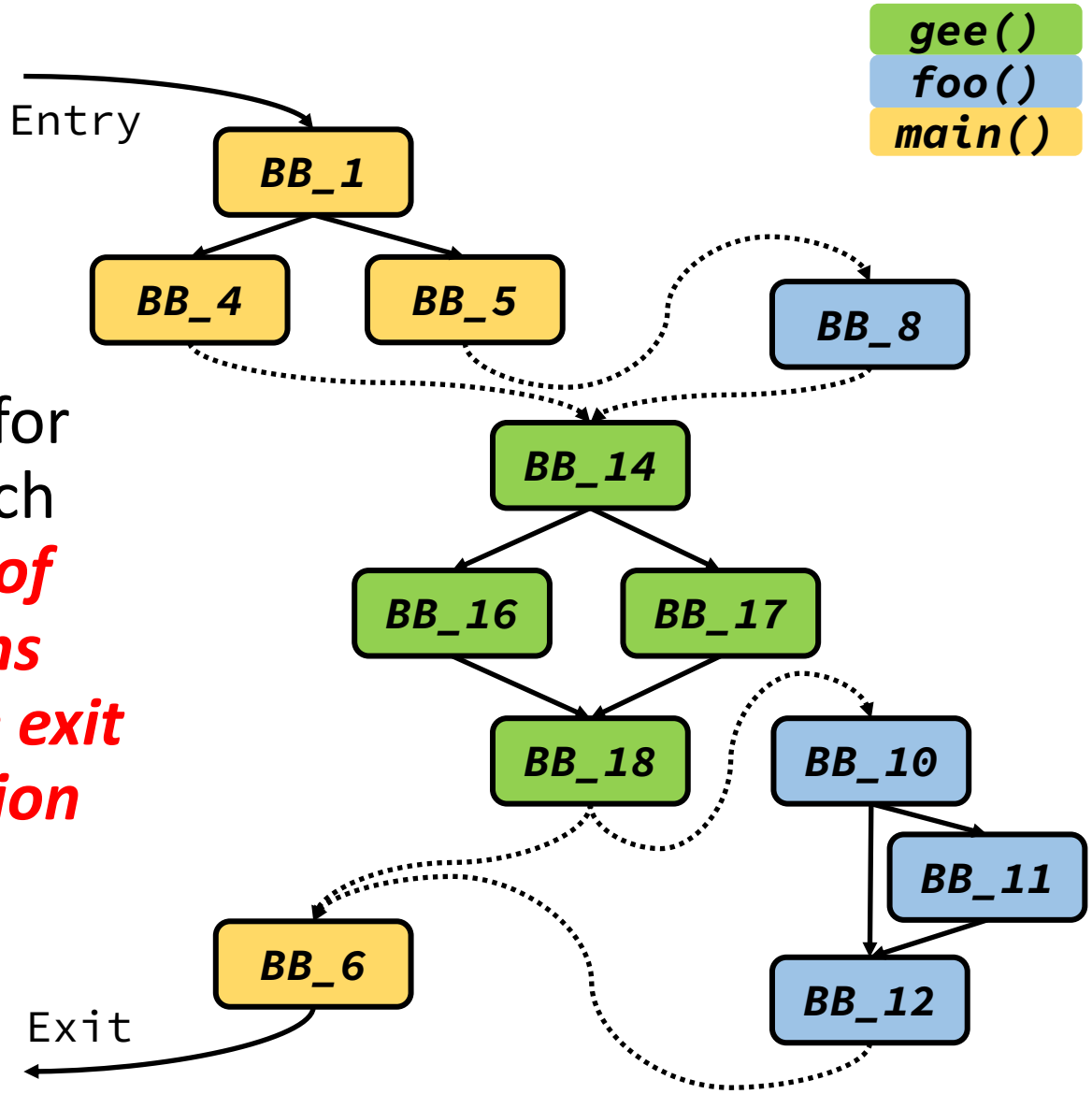
# Phase 1: Path Counting

- ***m*** inter-procedural paths from BB_4 to BB_6
- ***n*** inter-procedural paths from BB_5 to BB_6
- ***n/(m+n)*** probability to take BB_5 from BB_1



# Phase 1: Path Counting

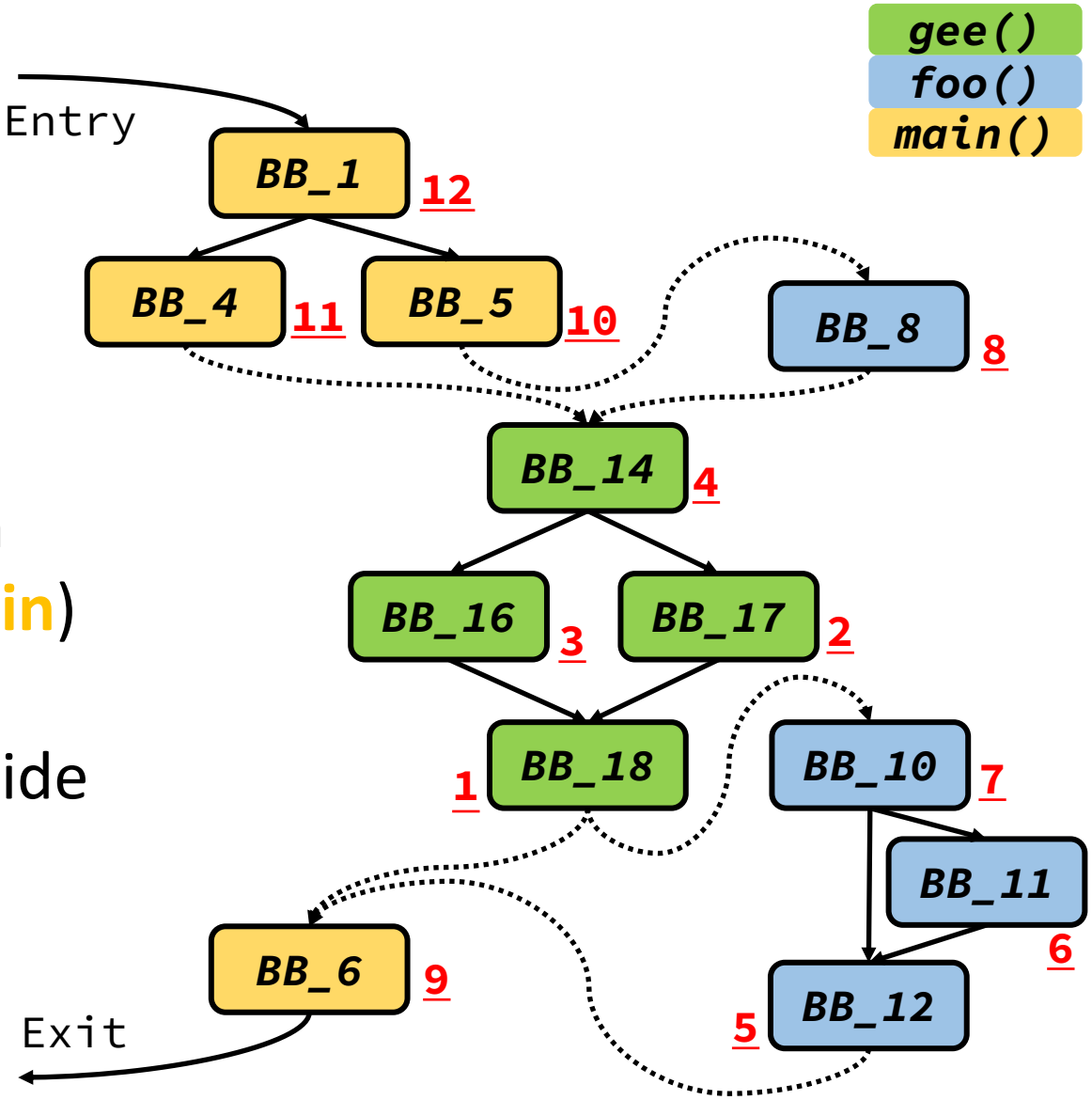
Compute the weight for each basic block, which denotes ***the number of inter-procedural paths from the block to the exit of its enclosing function***



# Phase 1: Path Counting

The path counting is performed in ***reverse topological order***

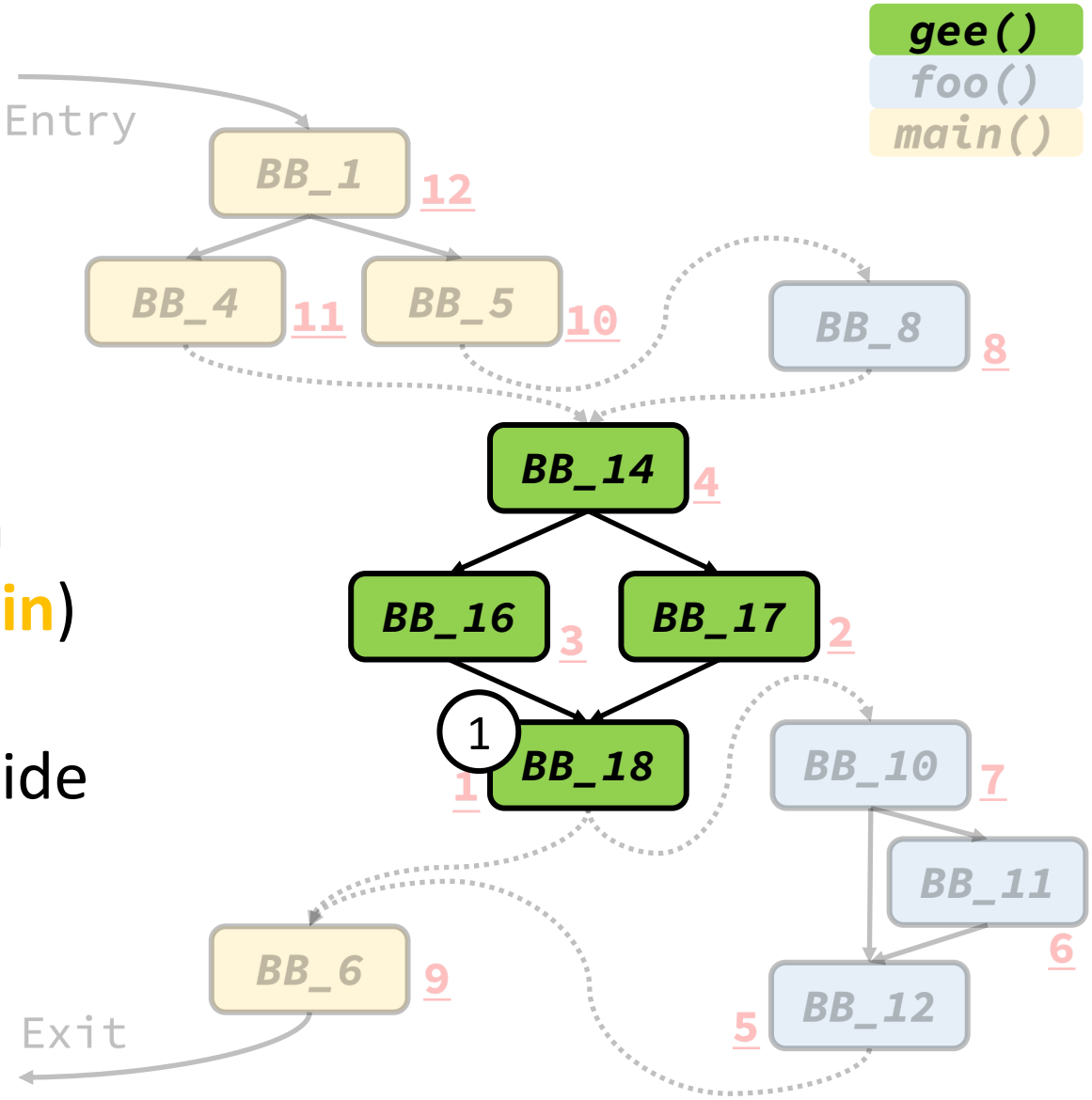
1. Sort the call graph (gee → foo → main)
2. Sort the nodes inside each function



# Phase 1: Path Counting

The path counting is performed in ***reverse topological order***

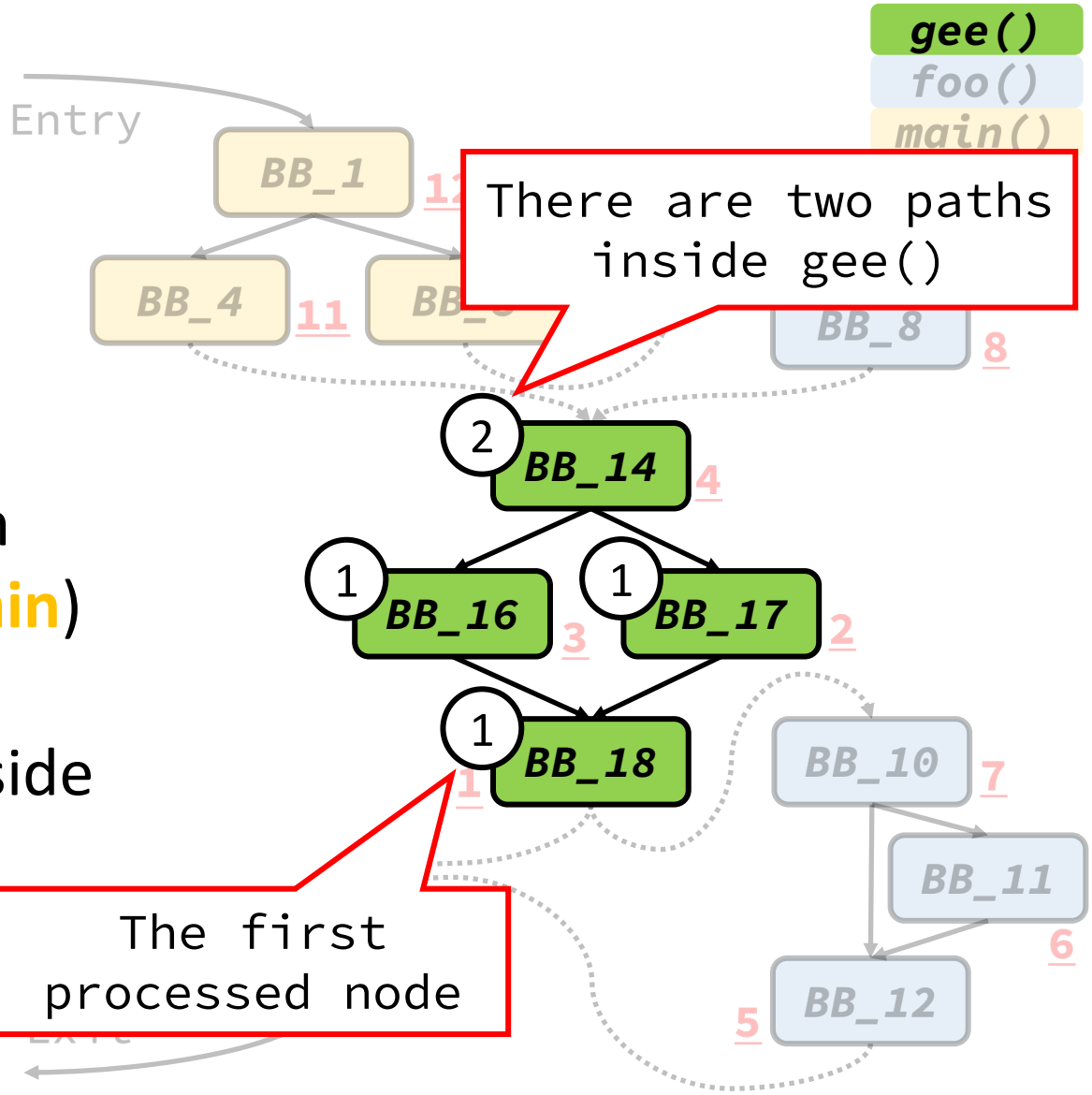
1. Sort the call graph (**gee** → **foo** → **main**)
2. Sort the nodes inside each function



# Phase 1: Path Counting

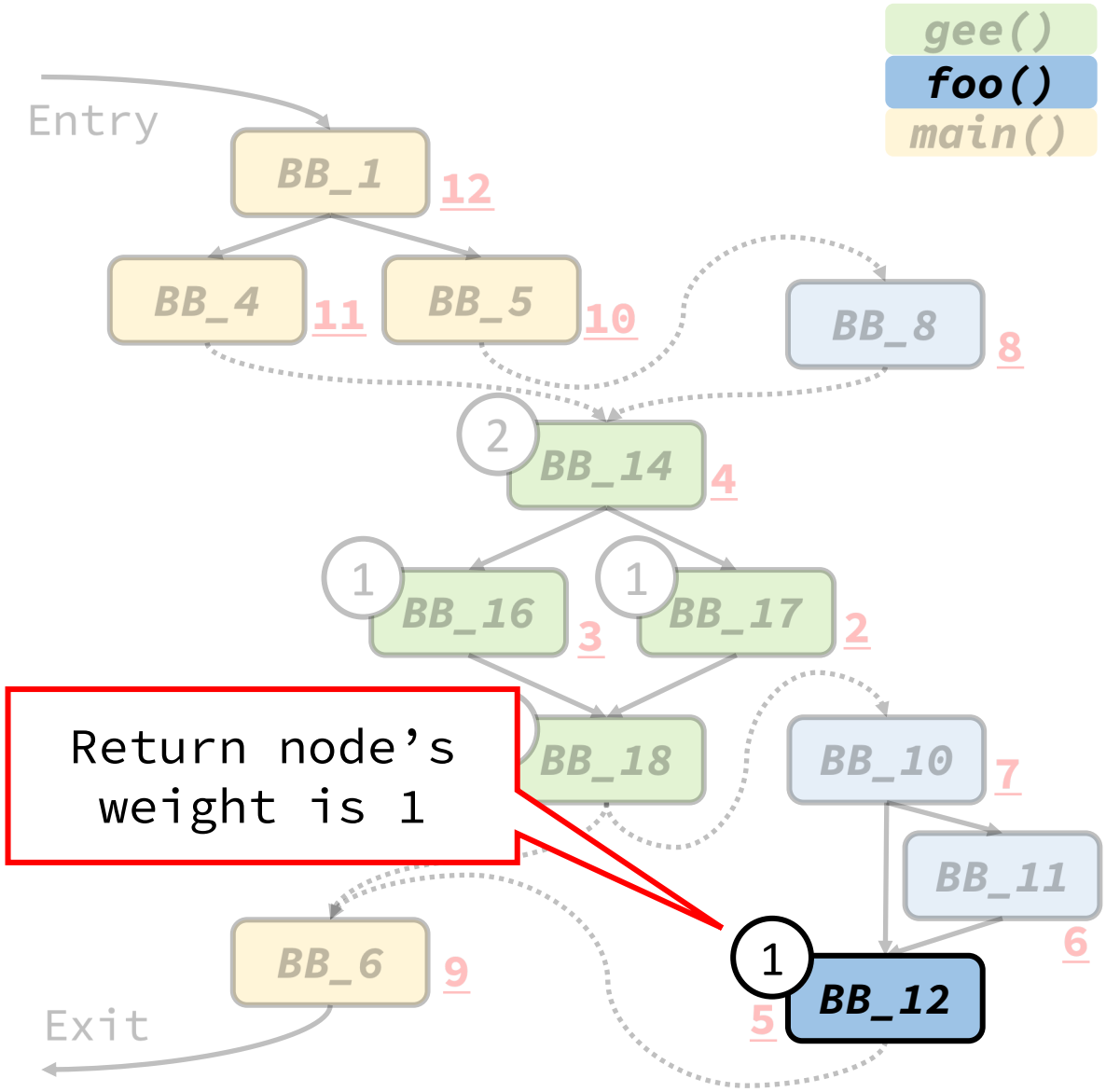
The path counting is performed in ***reverse topological order***

1. Sort the call graph (gee → foo → main)
2. Sort the nodes inside each function



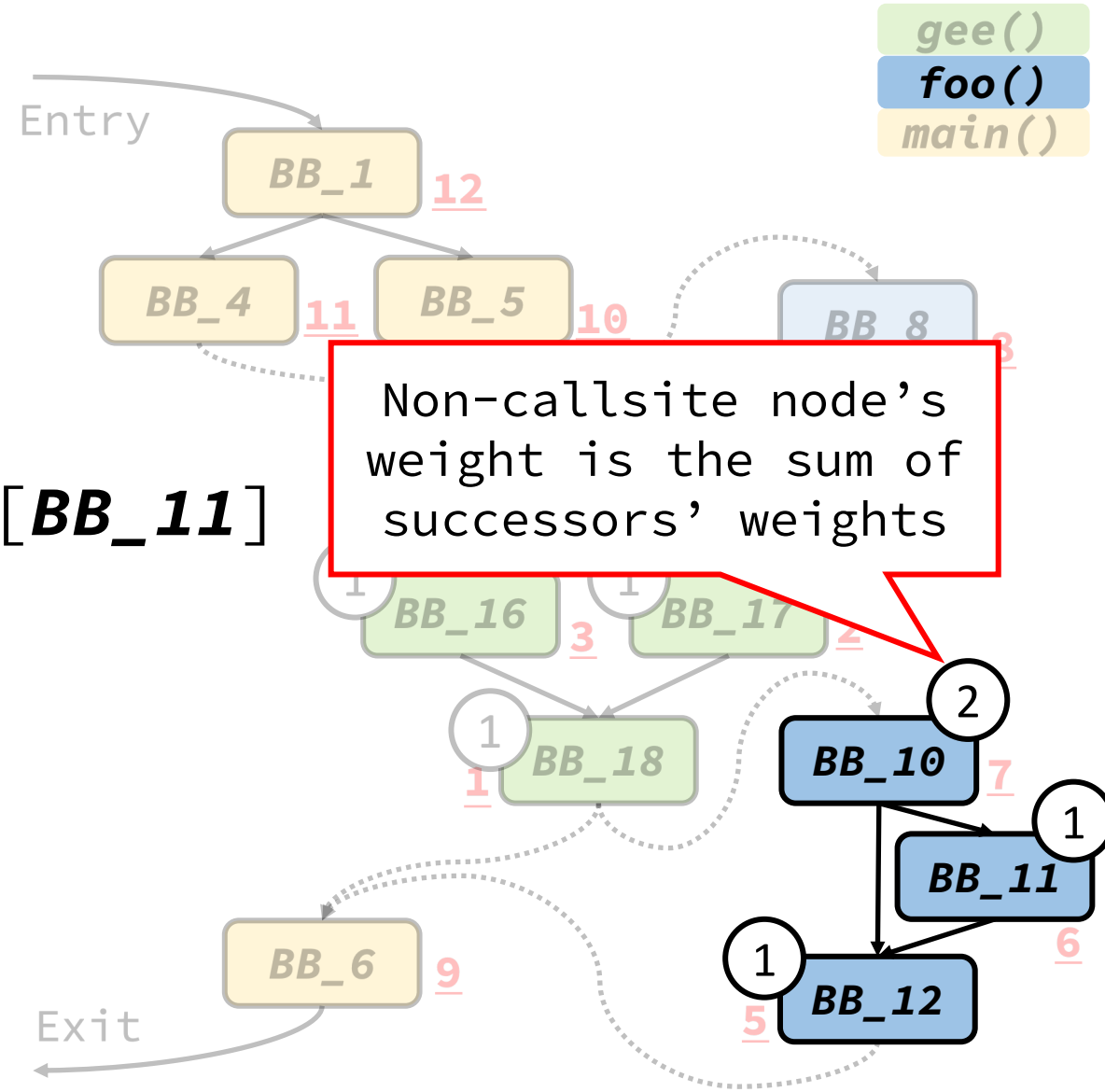
# Phase 1: Path Counting

$W[BB\_12] = 1$



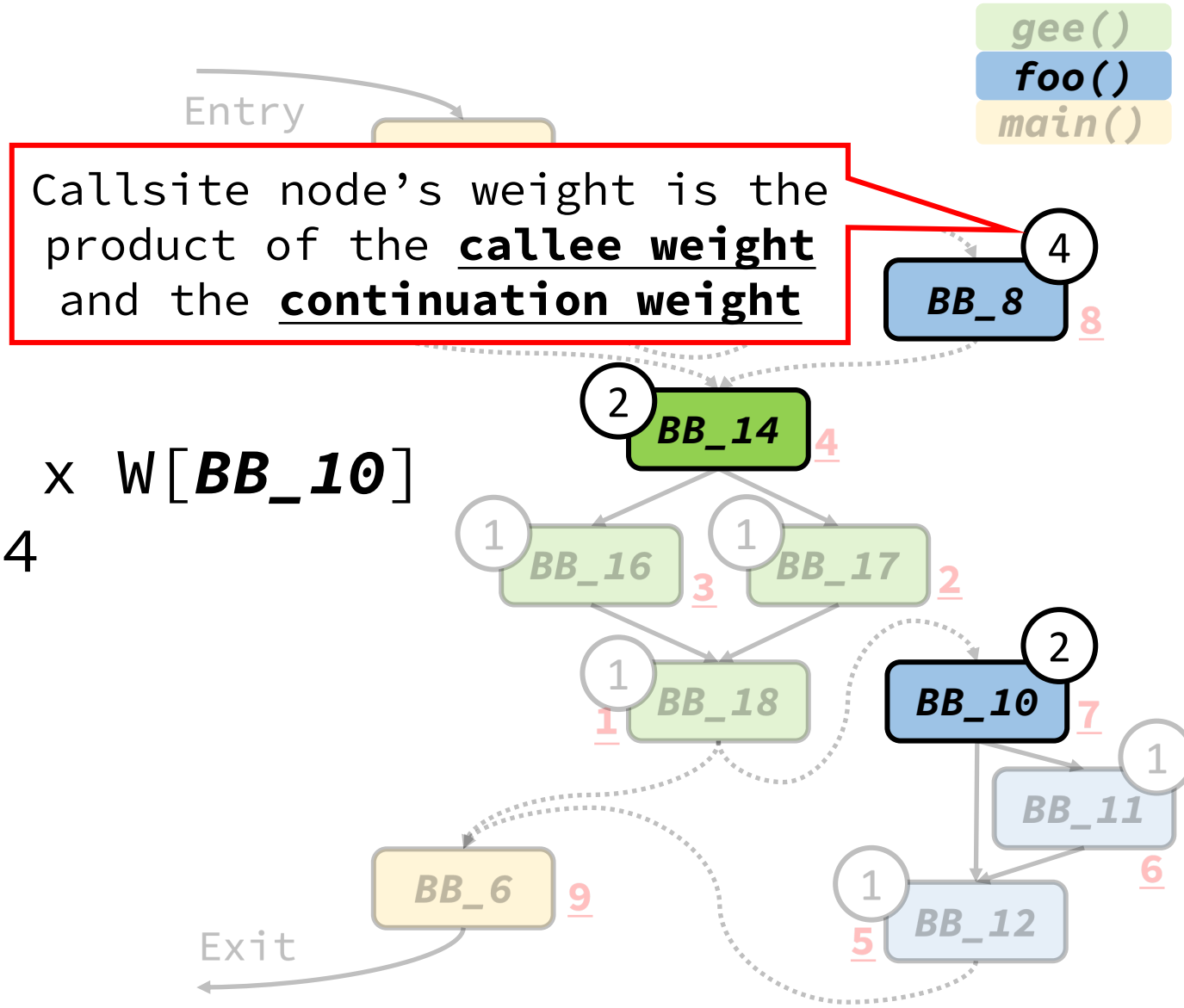
# Phase 1: Path Counting

$$W[BB\_10] = W[BB\_12] + W[BB\_11] = 1 + 1 = 2$$



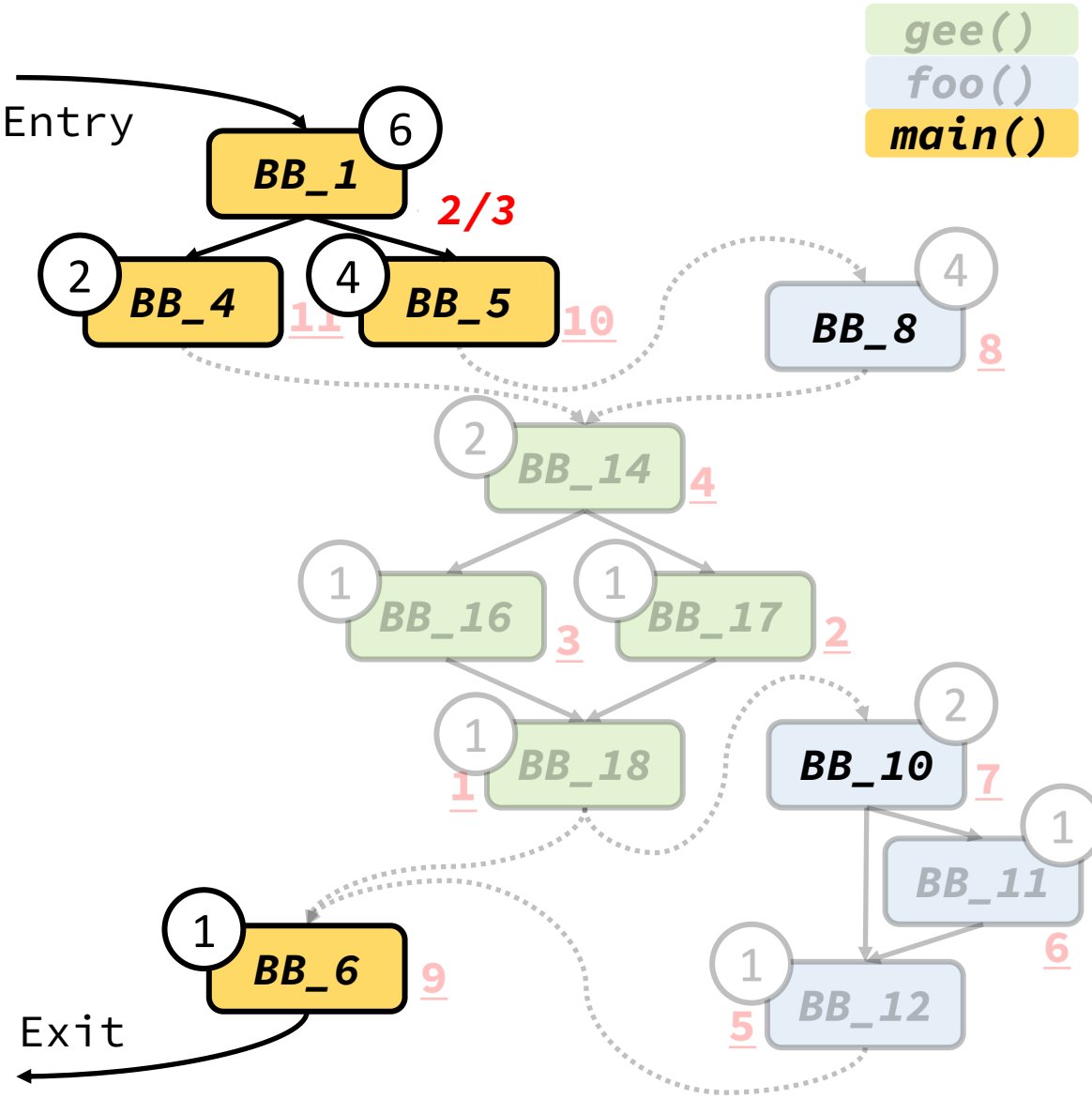
# Phase 1: Path Counting

W[*BB_8*]
= W[*BB_14*] x W[*BB_10*]
= 2 x 2 = 4



# Phase 1: Path Counting

- ***2*** inter-procedural paths from BB_4 to BB_6
- ***4*** inter-procedural paths from BB_5 to BB_6
- ***2/3*** probability to take BB_5 from BB_1



# Phase 1: Path Sampling

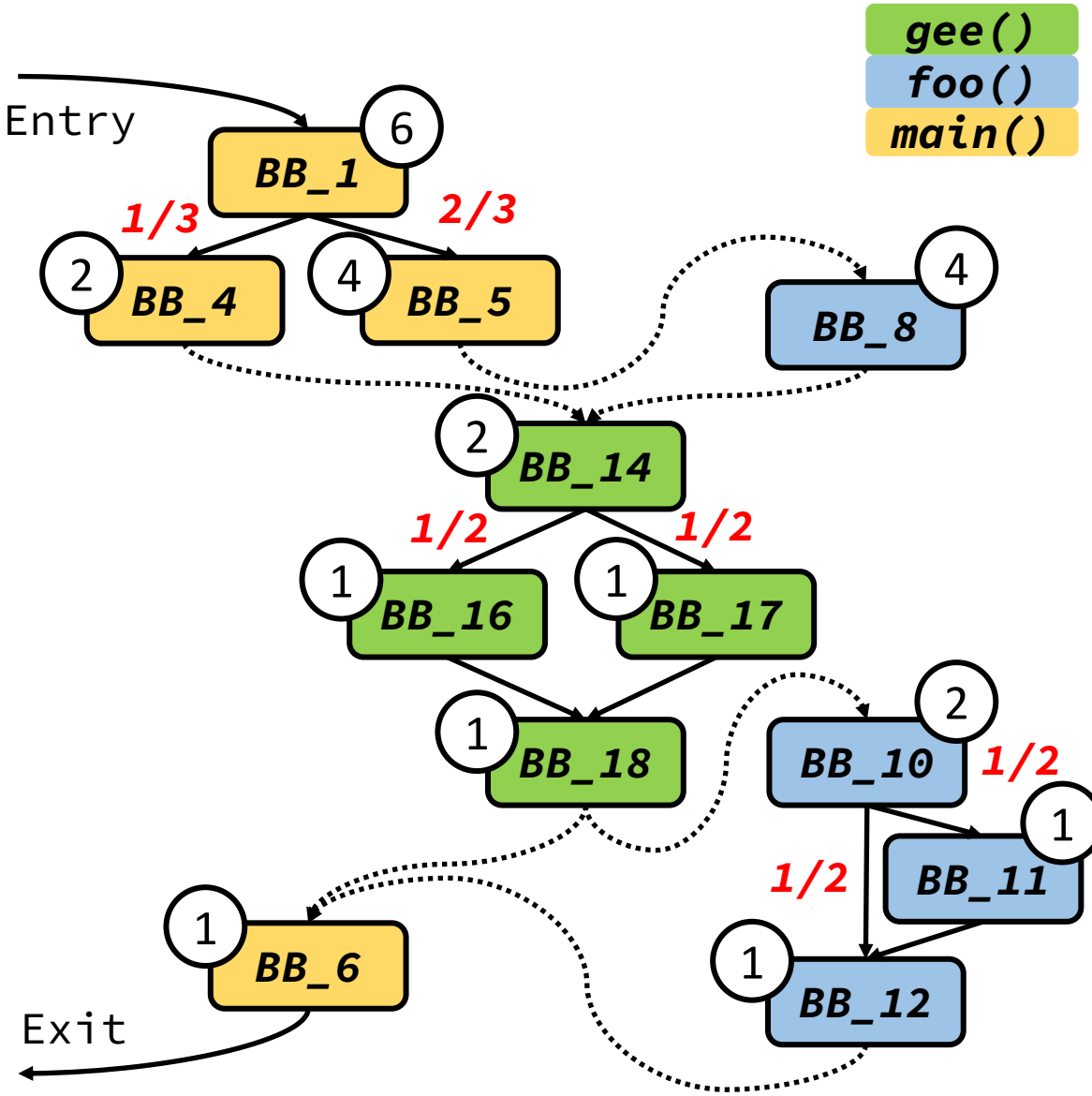
Path 1:

BB_1 → (1/3) → BB_4 → BB_14 → (1/2) → BB_16 → BB_18 → BB_6

Path 2:

BB_1 → (2/3) → BB_5 → BB_8 → BB_14 → (1/2) → BB_17 → BB_18 → BB_10 → (1/2) → BB_12 → BB_6

$P_1 = 1/6$ | $P_2 = 1/6$



# Phase 1: Practical Challenges

- The weight of each block is extremely large
  - The number of whole-program paths: ***O($2^n$)***

# Phase 1: Practical Challenges

- The weight of each block is extremely large
  - The number of whole-program paths: ***O($2^n$)***
  - How to handle biased distribution (e.g., ***1 : $10^{1000}$***)

A simple random number generator will introduce substantial error on such a biased odds.

# Phase 1: Practical Challenges

- The weight of each block is extremely large
  - The number of whole-program paths: ***O($2^n$)***
  - How to handle biased distribution (e.g., ***1 : $10^{1000}$***)
  - We develop a novel algorithm to simulate the biased distribution

# Phase 1: Practical Challenges

- The weight of each block is extremely large
- Loops and recursion [***Bounded unrolling***]
- Multi-exit [***Two different kinds of weights***]

# Phase 2: Per-Path Abstract Interpretation

- *Follow the given sampled path*

# Phase 2: Per-Path Abstract Interpretation

- ***Follow the given sampled path***
- Use singleton value, instead of strided interval

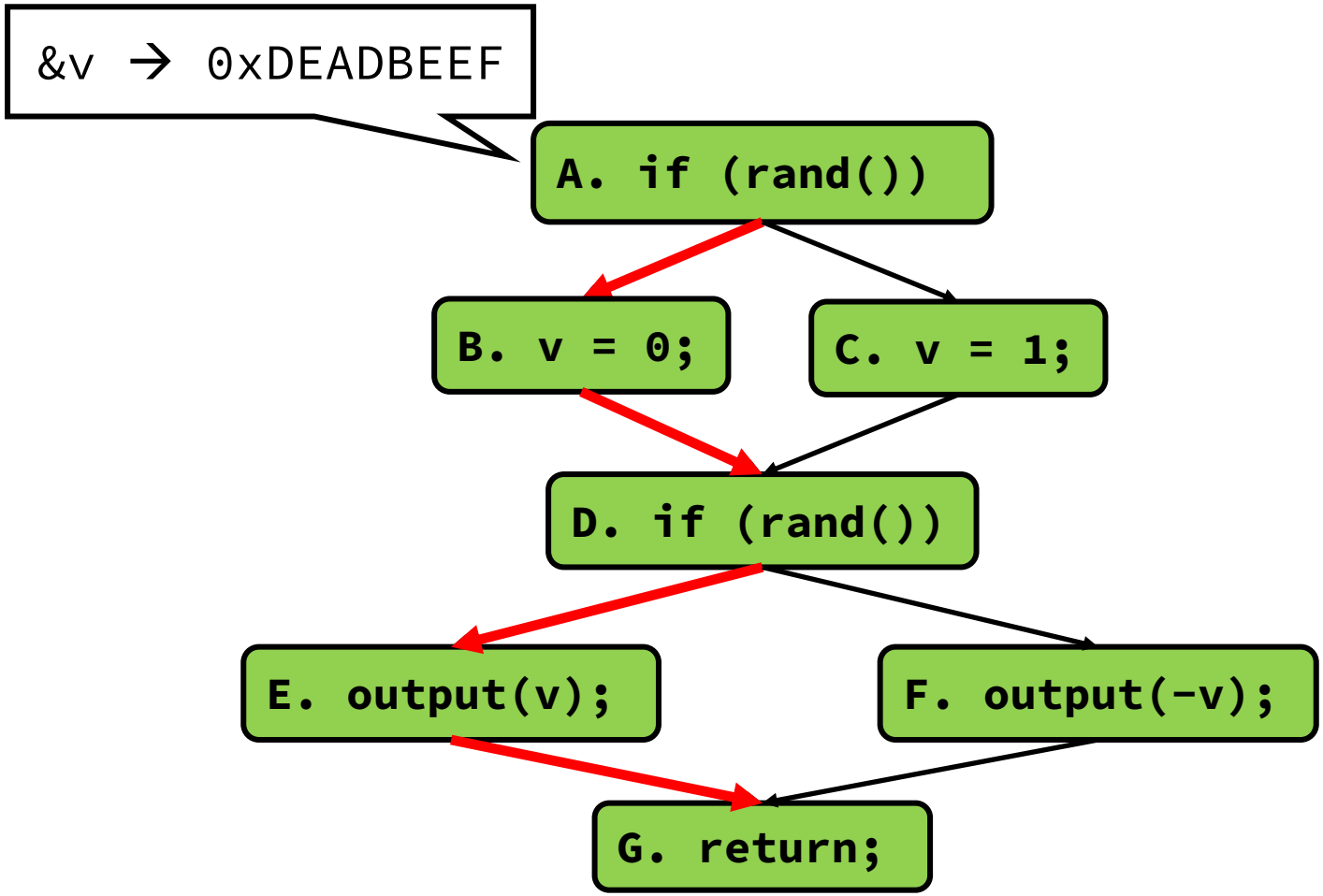
# Phase 2: Per-Path Abstract Interpretation

- ***Follow the given sampled path***
- Use singleton value, instead of strided interval
- Is to-some-extent similar to concrete execution

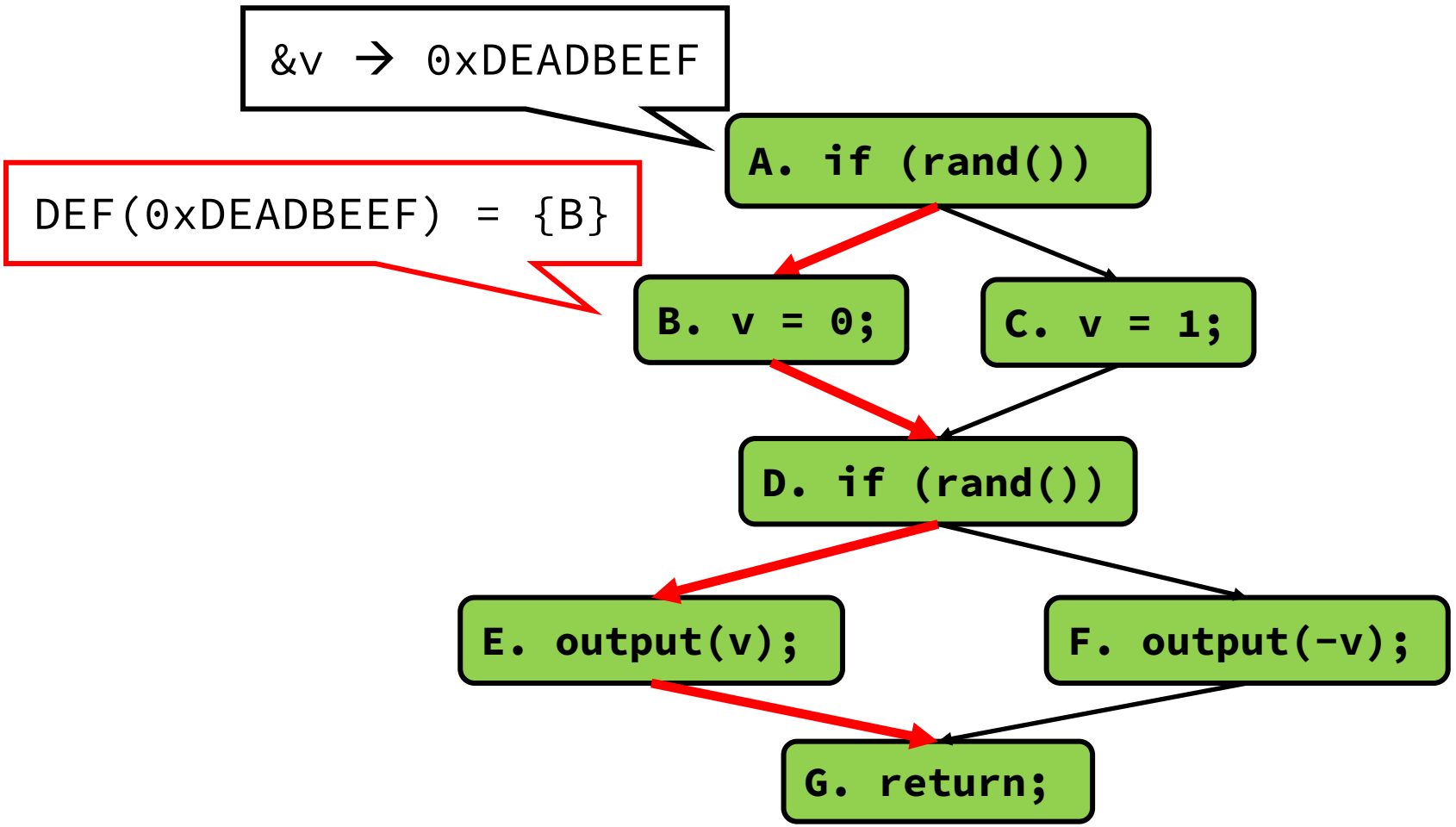
# Phase 2: Per-Path Abstract Interpretation

- ***Follow the given sampled path***
- Use singleton value, instead of strided interval
- Is to-some-extent similar to concrete execution
- Compute the possible values for individual instructions, which will be used to collect the ***definition and use information*** about memory

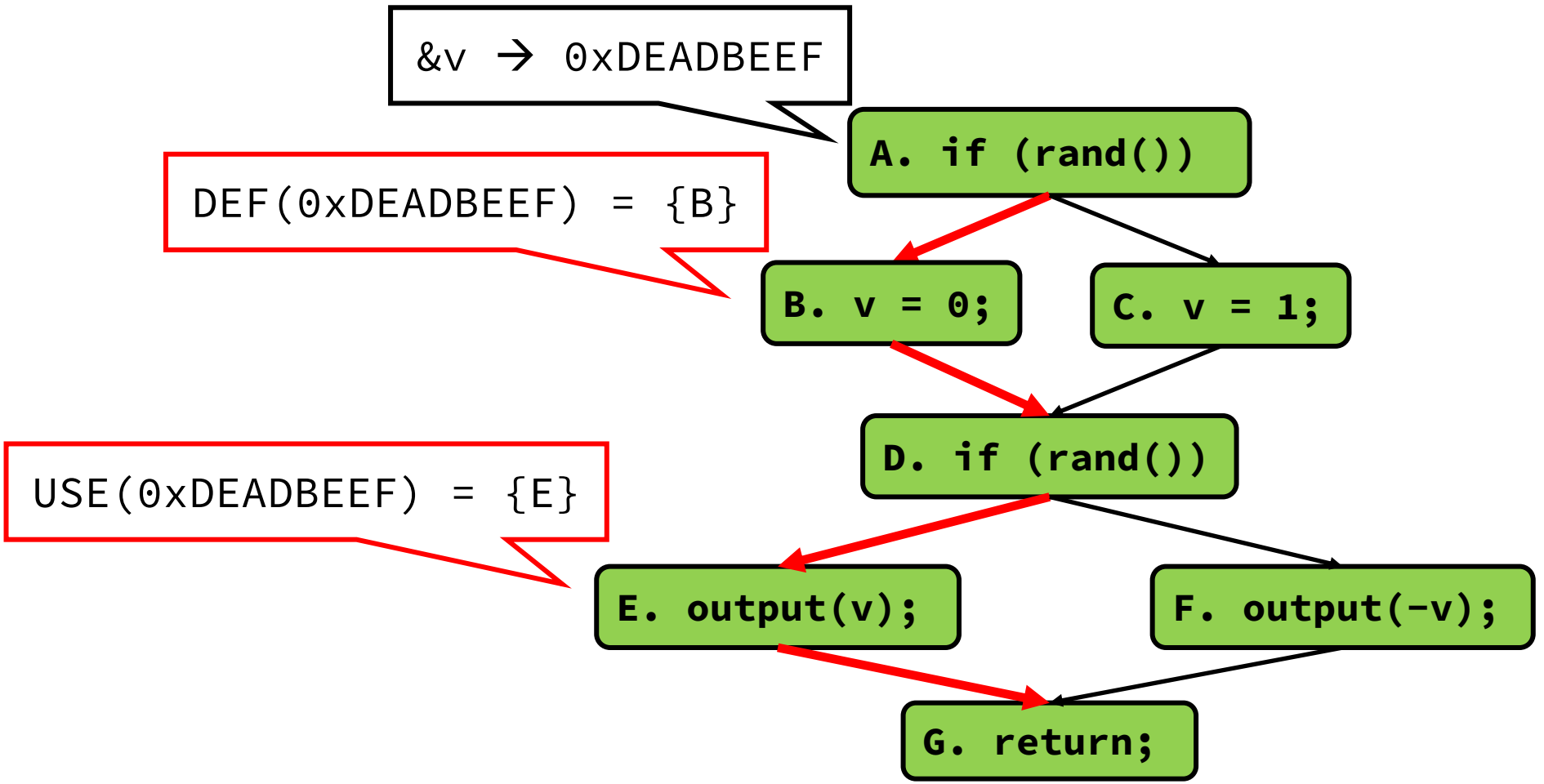
# Phase 2: Per-Path Abstract Interpretation



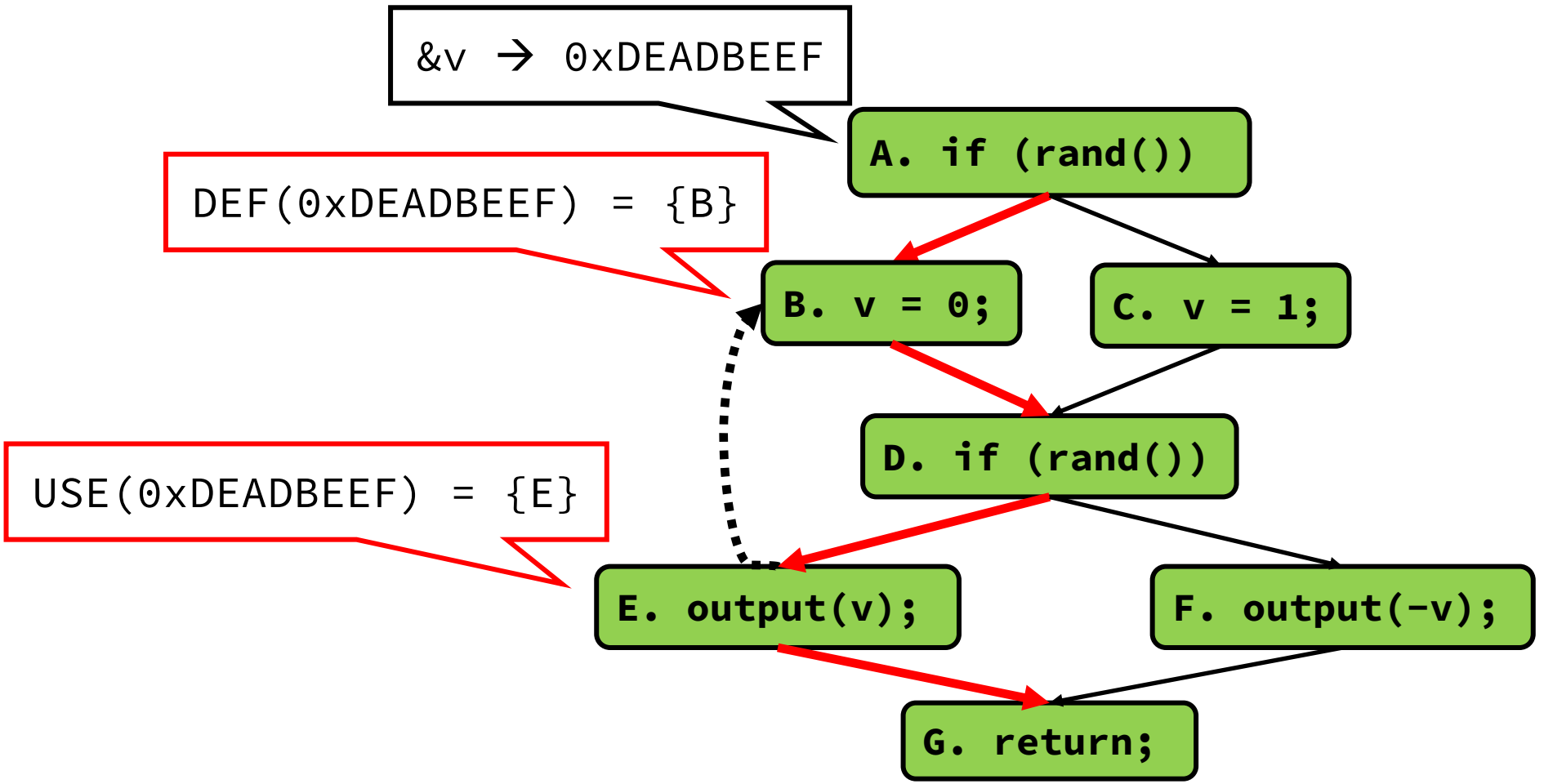
# Phase 2: Per-Path Abstract Interpretation



# Phase 2: Per-Path Abstract Interpretation



# Phase 2: Per-Path Abstract Interpretation

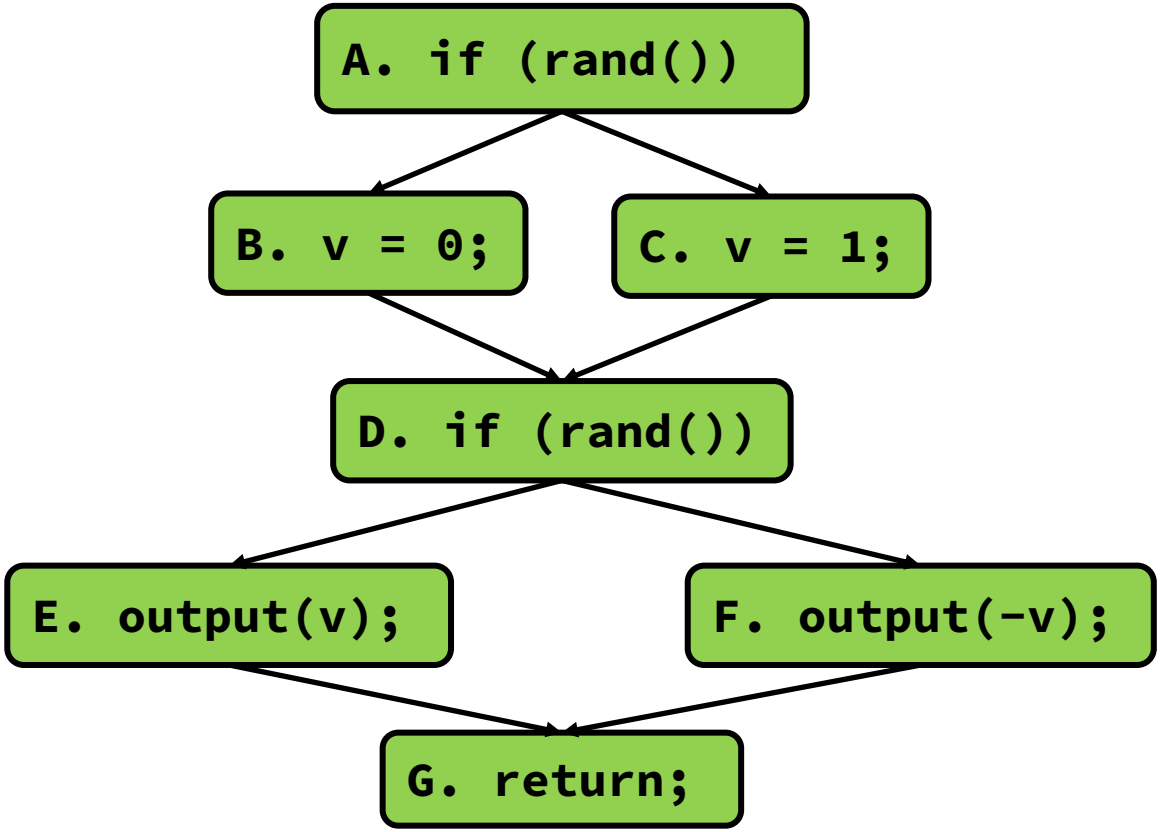


# Phase 3: Posterior Analysis

- Cannot sample all the whole-program paths
- Miss some dependence belonging to uncovered paths

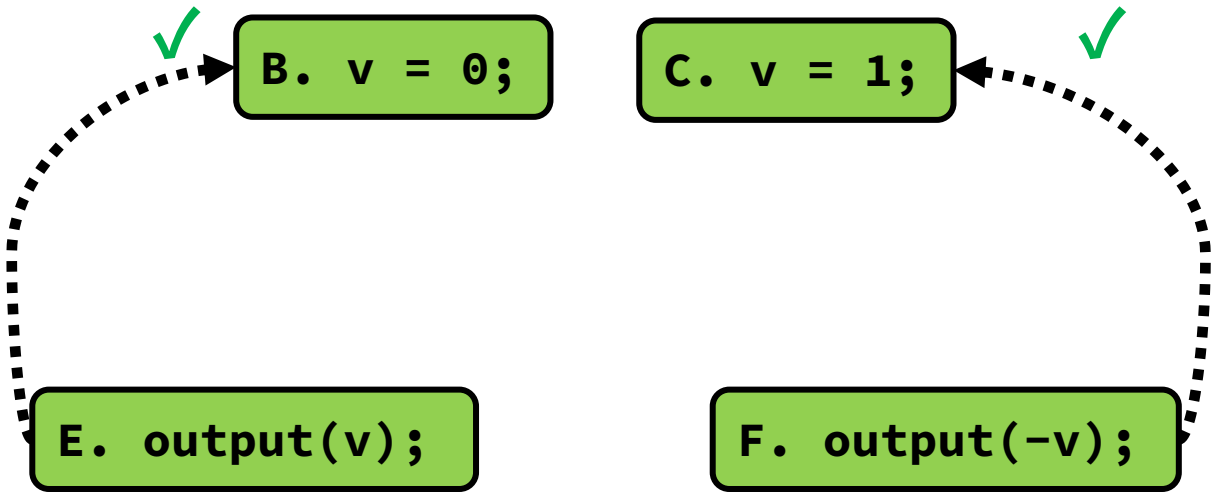
# Phase 3: Posterior Analysis

- Cannot sample all the whole-program paths
- Miss some dependence belonging to uncovered paths



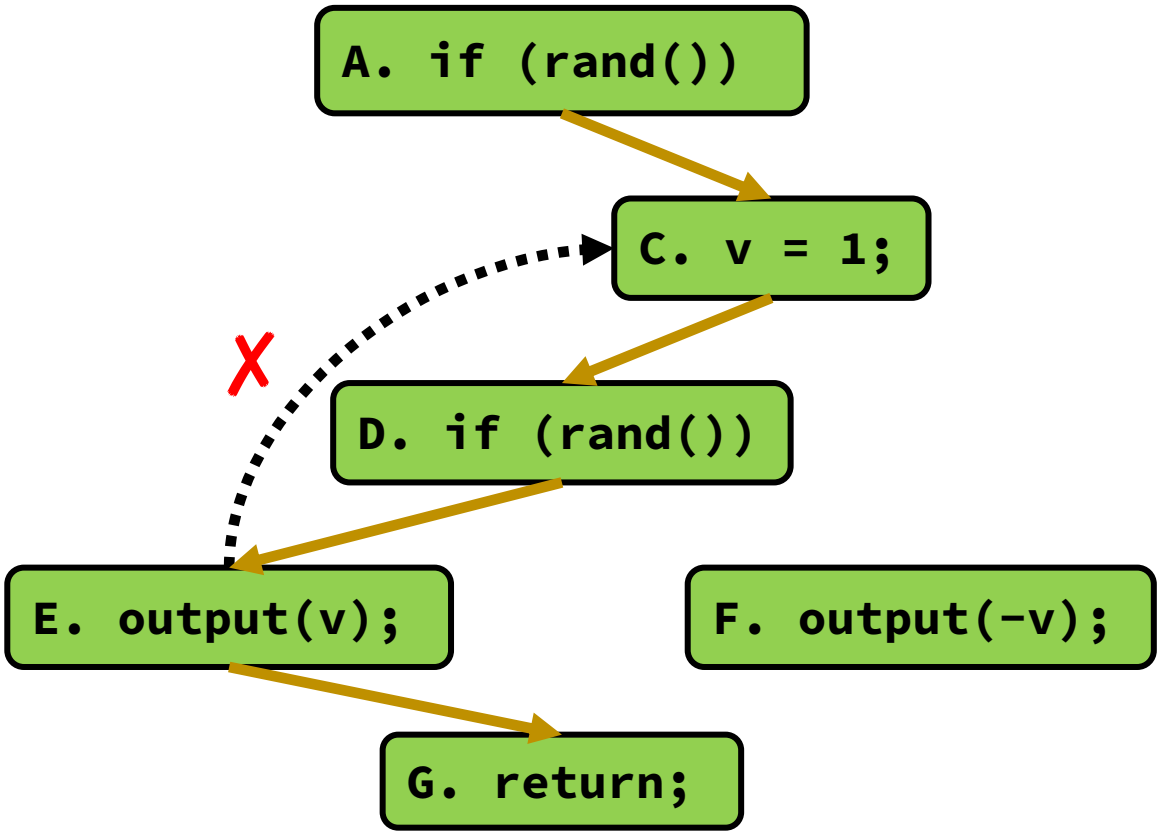
# Phase 3: Posterior Analysis

- Cannot sample all the whole-program paths
- Miss some dependence belonging to uncovered paths



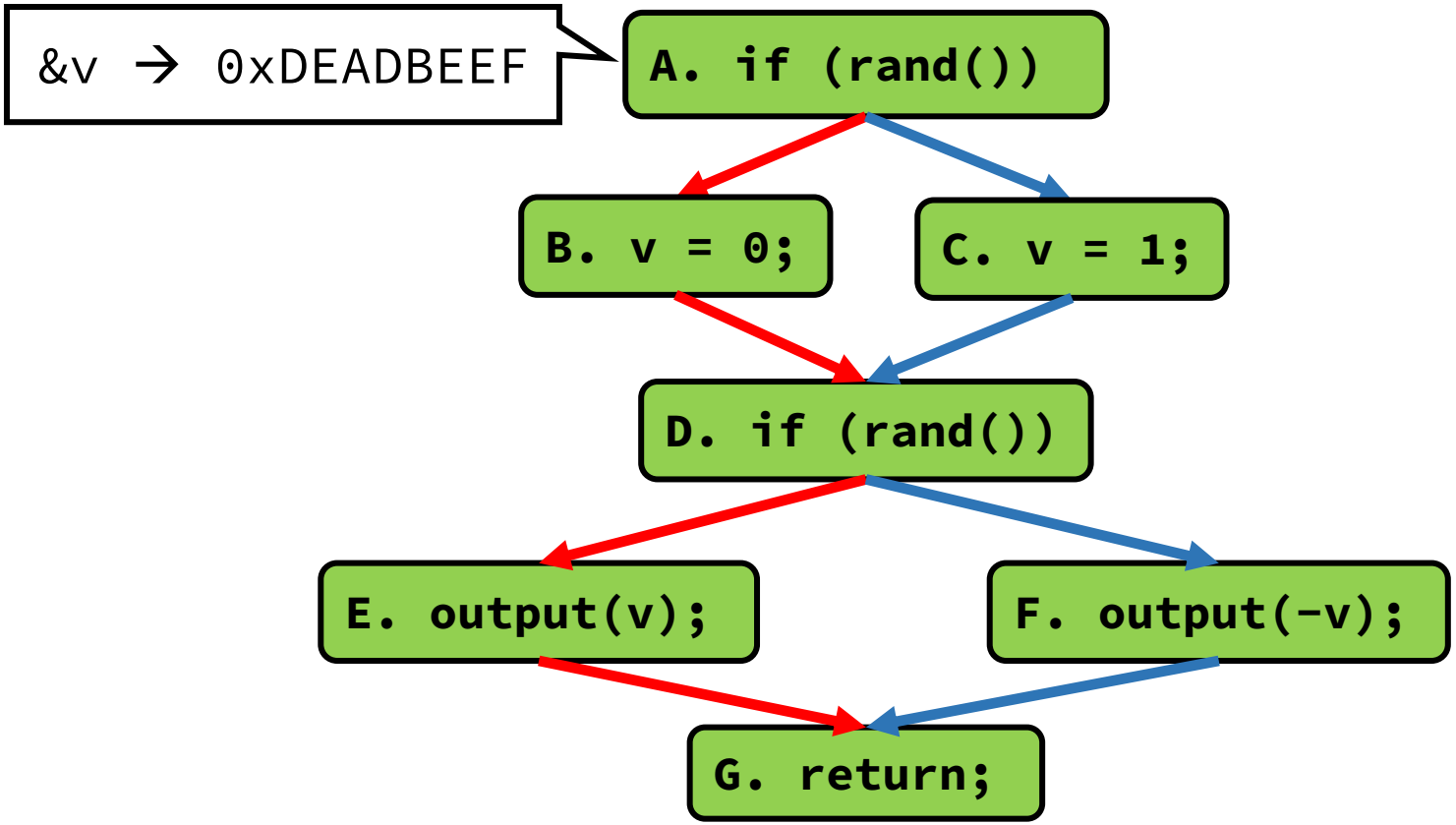
# Phase 3: Posterior Analysis

- Cannot sample all the whole-program paths
- Miss some dependence belonging to uncovered paths



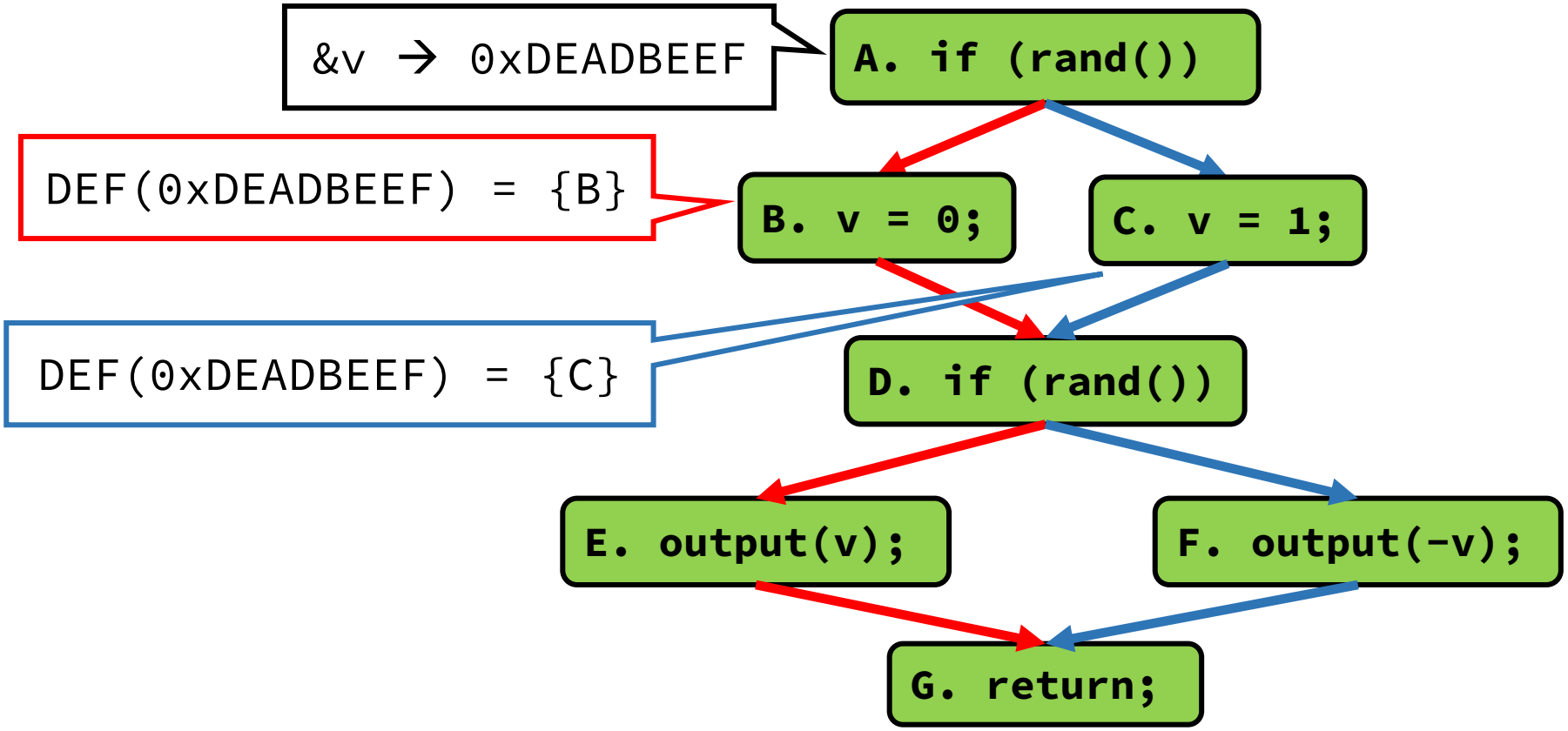
# Phase 3: Posterior Analysis

- Merge per-path memory write information at each control-flow joint point



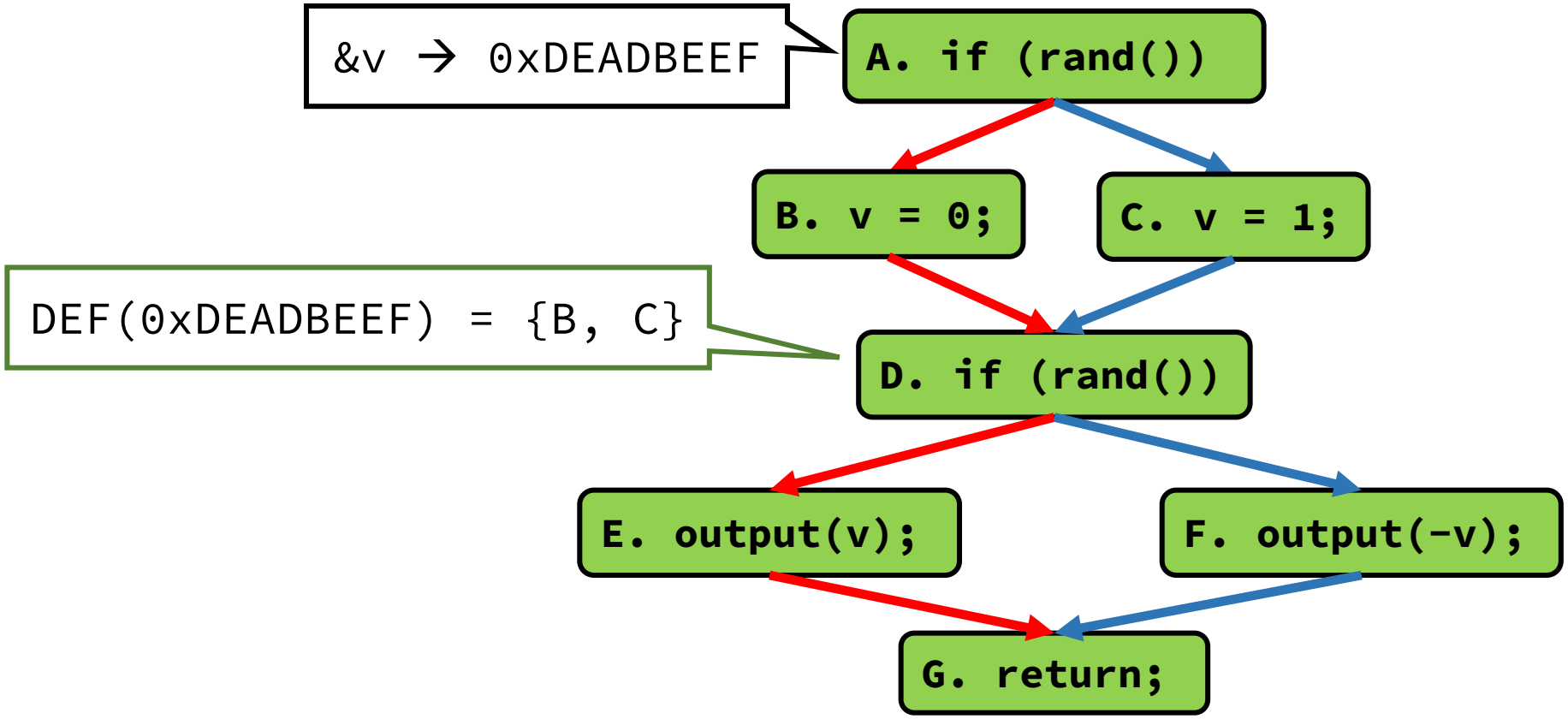
# Phase 3: Posterior Analysis

- Merge per-path memory write information at each control-flow joint point



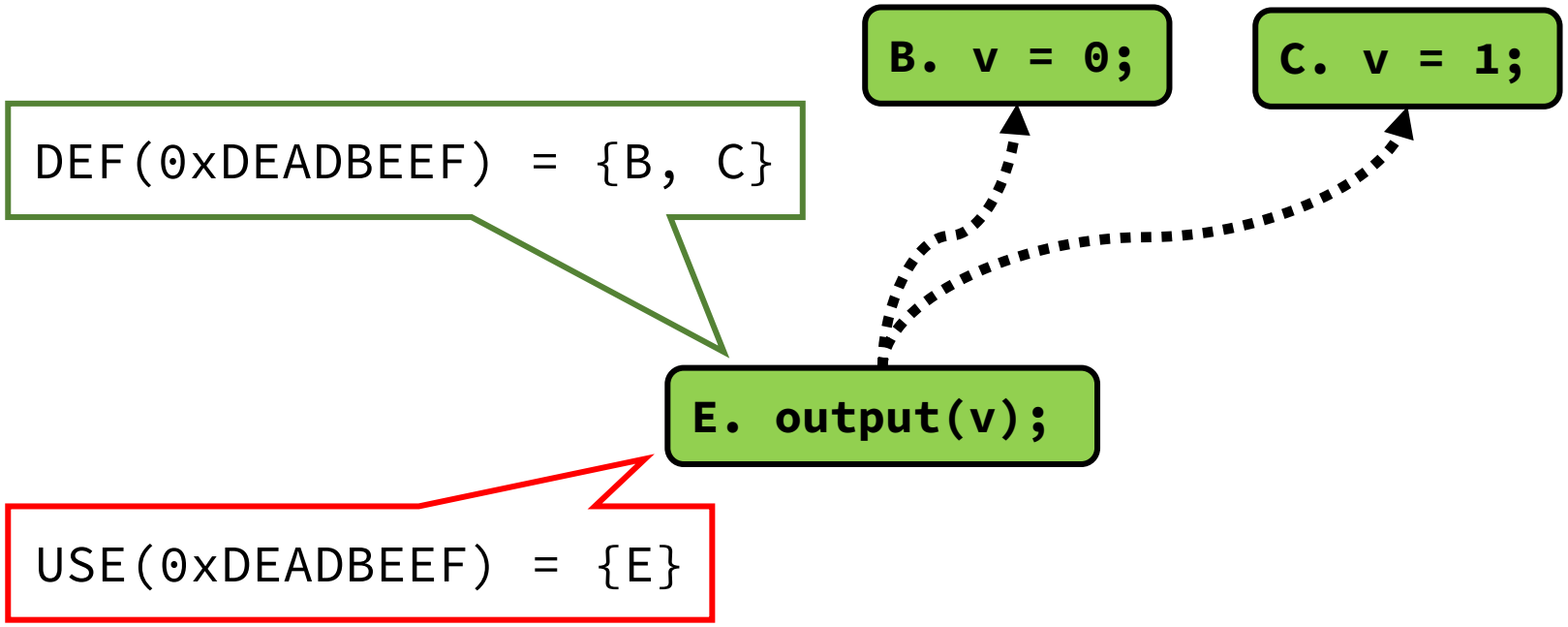
# Phase 3: Posterior Analysis

- Merge per-path memory write information at each control-flow joint point



# Phase 3: Posterior Analysis

- Cross-check the memory read information to detect dependence



# Probabilistic Guarantees

- Assume ***m*** out of total ***n*** paths disclose a dependence, and let ***k = m/n***
- For one sample, the probability $p_d$ of observing a given dependency ***d*** is:

$$\left(\frac{2^{63}}{2^{63}+1}\right)^{2L} \cdot k \leq p_d = \widetilde{p} \cdot m \leq \left(\frac{2^{63}+1}{2^{63}}\right)^{2L} \cdot k$$

- For N sample, the probability $P_d$ of observing a given dependency ***d*** is:

$$P_d = 1-(1-p_d)^N \geq 1-\left(1-\left(\frac{2^{63}}{2^{63}+1}\right)^{2L} \cdot k\right)^N \approx 1-(1-k)^N$$

# Probabilistic Guarantees

```
1 src[8] = '\x00';
2 strcpy(src, dst);
```

- Loop unrolling = ***15***
- Probabilities of observing the the dependence from strcpy to line 1:
  - Sample ***5*** times

$$p_d \geq 1 - (1 - \frac{8}{15})^5 = 0.967$$

  - Sample ***50*** times

$$p_d \geq 1 - (1 - \frac{8}{15})^{50} = 1 - 2.2 \times 10^{-14}$$

# Evaluation

- We implemented BDA in ***Rust***

- The system is available at

  https://github.com/bda-tool/bda/

# Evaluation

- Code and Intra-procedural Path Coverage
- Program Dependence Analysis
- Necessity of Posterior Analysis
- Effect of Sampling
- Analysis Overhead
- Downstream Analysis

# Evaluation: Program Dependence Analysis (Compared with ALTO)

| PROGRAM | #REFER | ALTO | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| 164.gzip | 3,580 | 2,229,749 | 0.00 | 2,226,169 | 13.55 |
| 175.vpr | 13,042 | 36,840,012 | 0.00 | 36,826,970 | 72.45 |
| 181.mcf | 2,050 | 588,076 | 0.00 | 586,026 | 55.20 |
| 186.crafty | 30,777 | 44,139,556 | 0.00 | 44,108,779 | 11.16 |
| 197.parser | 15,196 | 32,905,403 | 0.00 | 32,890,207 | 89.21 |
| 252.eon | 4,401 | 994,655 | 0.00 | 990,264 | 98.02 |
| 253.perlbmk | 57,507 | 102,068,477 | 0.00 | 100,349,485 | 92.69 |
| 254.gap | 7,935 | 10,611,636 | 0.00 | 10,603,701 | 94.06 |
| 255.vortex | 29,971 | 265,981,817 | 0.00 | 265,951,846 | 89.66 |
| 256.bzip2 | 4,306 | 2,466,876 | 0.00 | 2,462,570 | 28.71 |
| 300.twolf | 16,710 | 44,735,257 | 0.00 | 44,718,440 | 75.42 |
| ***Avg.*** | ***16.861*** | ***49,414,683*** | ***0.00*** | ***49,246,769*** | ***65.47*** |

Benchmark: ***SPECTINT 2000*** Timeout Budget: ***12 hours***

# Evaluation: Program Dependence Analysis (Compared with ALTO)

| PROGRAM | #REFER | ALTO | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| 164.gzip | 3,580 | 2,229, | 0.00 | 2,226,169 | 13.55 |
| 175.vpr | 13,042 | 36,8 | | | |
| 181.mcf | 2,050 | 588 | | | |
| 186.crafty | 30,777 | 44,1 | | | |
| 197.parser | 15,196 | 32,905,403 | 0.00 | 32,890,207 | 89.21 |
| 252.eon | 4,401 | 994,655 | 0.00 | 990,264 | 98.02 |
| 253.perlbmk | 57,507 | 102,068,477 | 0.00 | 100,349,485 | 92.69 |
| 254.gap | 7,935 | 10,611,636 | 0.00 | 10,603,701 | 94.06 |
| 255.vortex | 29,971 | 265,981,817 | 0.00 | 265,951,846 | 89.66 |
| 256.bzip2 | 4,306 | 2,466,876 | 0.00 | 2,462,570 | 28.71 |
| 300.twolf | 16,710 | 44,735,257 | 0.00 | 44,718,440 | 75.42 |
| ***Avg.*** | ***16.861*** | ***49,414,683*** | ***0.00*** | ***49,246,769*** | ***65.47*** |

Dependence observed by reference execution

# Evaluation: Program Dependence Analysis (Compared with ALTO)

| PROGRAM | #REFER | ALTO | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| 164.gzip | 3,580 | 2,[illegible]49 | 0.00 | 2,226,169 | 13.55 |
| 175.vpr | 13,[illegible] | [illegible]840,012 | 0.00 | 36,826,970 | 72.45 |
| [illegible] | [illegible] | [illegible] | 0.00 | 586,026 | 55.20 |
| [illegible] | [illegible] | [illegible]6 | 0.00 | 44,108,779 | 11.16 |
| [illegible] | [illegible] | [illegible]3 | 0.00 | 32,890,207 | 89.21 |
| 252.eon | 4,401 | 994,655 | 0.00 | 990,264 | 98.02 |
| 253.perlbmk | 57,507 | 102,068,477 | 0.00 | 100,349,485 | 92.69 |
| 254.gap | 7,935 | 10,611,636 | 0.00 | 10,603,701 | 94.06 |
| 255.vortex | 29,971 | 265,981,817 | 0.00 | 265,951,846 | 89.66 |
| 256.bzip2 | 4,306 | 2,466,876 | 0.00 | 2,462,570 | 28.71 |
| 300.twolf | 16,710 | 44,735,257 | 0.00 | 44,718,440 | 75.42 |
| ***Avg.*** | ***16.861*** | ***49,414,683*** | ***0.00*** | ***49,246,769*** | ***65.47*** |

Dependence detected by reference execution but not by the tool

# Evaluation: Program Dependence Analysis (Compared with ALTO)

| PROGRAM | #REFER | ALTO | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| | | | 00 | 2,226,169 | 13.55 |
| | | | 00 | 36,826,970 | 72.45 |
| | | | 00 | 586,026 | 55.20 |
| | | | 00 | 44,108,779 | 11.16 |
| 197.parser | 15,196 | 32,905,403 | 0.00 | 32,890,207 | 89.21 |
| 252.eon | 4,401 | 994,655 | 0.00 | 990,264 | 98.02 |
| 253.perlbmk | 57,507 | 102,068,477 | 0.00 | 100,349,485 | 92.69 |
| 254.gap | 7,935 | 10,611,636 | 0.00 | 10,603,701 | 94.06 |
| 255.vortex | 29,971 | 265,981,817 | 0.00 | 265,951,846 | 89.66 |
| 256.bzip2 | 4,306 | 2,466,876 | 0.00 | 2,462,570 | 28.71 |
| 300.twolf | 16,710 | 44,735,257 | 0.00 | 44,718,440 | 75.42 |
| ***Avg.*** | ***16.861*** | ***49,414,683*** | ***0.00*** | ***49,246,769*** | ***65.47*** |

Dependence detected by the tool but not by reference execution

# Evaluation: Program Dependence Analysis (Compared with ALTO)

| PROGRAM | #REFER | ALTO | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| 164.gzip | 3,580 | 2,229,749 | 0.00 | 2,226,169 | 13.55 |
| 175.vpr | | | | 970 | 72.45 |
| 181.mcf | | | | 26 | 55.20 |
| 186.crafty | | | | 779 | 11.16 |
| 197.parser | | | | 207 | 89.21 |
| 252.eon | 4,401 | 994,655 | 0.00 | 990,264 | 98.02 |
| 253.perlbmk | 57,507 | 102,068,477 | 0.00 | 100,349,485 | 92.69 |
| 254.gap | 7,935 | 10,611,636 | 0.00 | 10,603,701 | 94.06 |
| 255.vortex | 29,971 | 265,981,817 | 0.00 | 265,951,846 | 89.66 |
| 256.bzip2 | 4,306 | 2,466,876 | 0.00 | 2,462,570 | 28.71 |
| 300.twolf | 16,710 | 44,735,257 | 0.00 | 44,718,440 | 75.42 |
| ***Avg.*** | ***16.861*** | ***49,414,683*** | ***0.00*** | ***49,246,769*** | ***65.47*** |

Dependence whose source and destination have different types

The checker is implemented as an LLVM pass, propagating symbol information to individual instructions, registers and memory locations.

# Evaluation: Program Dependence Analysis (Compared with ALTO)

| PROGRAM | #REFER | ALTO | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| 164.gzip | 3,580 | 2,229,749 | 0.00 | 2,226,169 | 13.55 |
| 175.vpr | 13,042 | 36,840,012 | 0.00 | 36,826,970 | 72.45 |
| 181.mcf | 2,050 | 588,076 | 0.00 | 586,026 | 55.20 |
| 186.crafty | 30,777 | 44,139,556 | 0.00 | 44,108,779 | 11.16 |
| 197.parser | 15,196 | 32,905,403 | 0.00 | 32,890,207 | 89.21 |
| 252.eon | 4,401 | 994,655 | 0.00 | 990,264 | 98.02 |
| 253.perlbmk | 57,507 | 102,068,477 | 0.00 | 100,349,485 | 92.69 |
| 254.gap | 7,935 | 10,611,636 | 0.00 | 10,603,701 | 94.06 |
| 255.vortex | 29,971 | 265,981,817 | 0.00 | 265,951,846 | 89.66 |
| 256.bzip2 | 4,306 | 2,466,876 | 0.00 | 2,462,570 | 28.71 |
| 300.twolf | 16,710 | 44,735,257 | 0.00 | 44,718,440 | 75.42 |
| ***Avg.*** | ***16.861*** | ***49,414,683*** | ***0.00*** | ***49,246,769*** | ***65.47*** |

ALTO reports ***49M*** dependence, ***65%*** of them are mis-typed , without missing any.

# Evaluation: Program Dependence Analysis (Compared with ALTO)

| PROGRAM | #REFER | BDA | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| 164.gzip | 3,580 | 29,370 | 0.22 | 25,798 | 11.92 |
| 175.vpr | 13,042 | 559,460 | 0.08 | 546,428 | 61.88 |
| 181.mcf | 2,050 | 3,347 | 0.00 | 1,297 | 12.94 |
| 186.crafty | 30,777 | 1,077,346 | 0.15 | 1,046,614 | 7.31 |
| 197.parser | 15,196 | 659,867 | 0.01 | 644,673 | 81.12 |
| 252.eon | 4,401 | 28,855 | 0.00 | 24,454 | 78.11 |
| 253.perlbmk | 57,507 | 5,389,973 | 0.23 | 5,363,373 | 82.77 |
| 254.gap | 7,935 | 205,200 | 0.52 | 197,306 | 74.30 |
| 255.vortex | 29,971 | 2,159,444 | 0.33 | 2,129,473 | 64.10 |
| 256.bzip2 | 4,306 | 13,917 | 0.23 | 9,621 | 10.84 |
| 300.twolf | 16,710 | 2,285,090 | 0.34 | 2,268,436 | 73.45 |
| ***Avg.*** | ***16.861*** | ***1,128,352*** | ***0.19*** | ***1,114,316*** | ***50.80*** |

BDA reports ***1M*** dependence (***48*** times smaller than ALTO's), ***50%*** of them are mis-typed, only with ***0.19%*** missing rate

# Evaluation: Program Dependence Analysis (Compared with VSA)

| PROGRAM | #REFER | BDA | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| ... | ... | ... | ... | ... | ... |
| 175.vpr | 13,042 | 559,460 | 0.08 | 546,428 | 61.88 |
| 181.mcf | 2,050 | 3,347 | 0.00 | 1,297 | 12.94 |
| 186.crafty | 30,777 | 1,077,346 | 0.15 | 1,046,614 | 7.31 |
| ... | ... | ... | ... | **VSA** ... | ... |
| PROGRAM | #REFER | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| ... | ... | ... | ... | ... | ... |
| 175.vpr | 13,042 | **TIMEOUT** | **TIMEOUT** | **TIMEOUT** | **TIMEOUT** |
| 181.mcf | 2,050 | 23,068 | 0.00 | 21,018 | 54.33 |
| 186.crafty | 30,777 | **TIMEOUT** | **TIMEOUT** | **TIMEOUT** | **TIMEOUT** |
| ... | ... | ... | ... | ... | ... |

BAP-VSA only handles 181.mcf within 12 hours.

# Evaluation: Program Dependence Analysis (Compared with VSA)

| PROGRAM | #REFER | BDA | | | |
|---|---|---|---|---|---|
| | | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| ... | ... | ... | ... | ... | ... |
| 175.vpr | 13,042 | 559,460 | 0.08 | 546,428 | 61.88 |
| 181.mcf | 2,050 | 3,347 | 0.00 | 1,297 | 12.94 |
| 186.crafty | 30,777 | 1,077,346 | 0.15 | 1,046,614 | 7.31 |
| ... | ... | ... | ... | **VSA** ... | ... |
| PROGRAM | #REFER | #FOUND | MISS(%) | #EXTRA | MISTYPED(%) |
| ... | ... | ... | ... | ... | ... |
| 175.vpr | 13,042 | **TIMEOUT** | **TIMEOUT** | **TIMEOUT** | **TIMEOUT** |
| 181.mcf | 2,050 | 23,068 | 0.00 | 21,018 | 54.33 |
| 186.crafty | 30,777 | **TIMEOUT** | **TIMEOUT** | **TIMEOUT** | **TIMEOUT** |
| ... | ... | ... | ... | ... | ... |

BAP-VSA only handles 181.mcf within 12 hours.

BDA reports ***5*** times less dependence, with less mistyped ones.

# Evaluation: Downstream Analysis

# Evaluation: Identify Indirect Control Flow Targets

***IDA*** is a widely used commercial disassembling tools.

| PROGRAM | #INDIRECT JUMP EDGES | | | #INDIRECT CALL EDGES | | |
|---|---|---|---|---|---|---|
| | IDA | REFER | BDA | IDA | REFER | BDA |
| 164.gzip | 0 | 0 | 0 | 0 | 3 | 3 |
| 175.vpr | 49 | 0 | 49 | 0 | 1 | 1 |
| 176.gcc | 3,628 | 324 | 3,628 | 25 | 214 | 853 |
| 181.mcf | 0 | 0 | 0 | 0 | 1 | 1 |
| 186.crafty | 159 | 38 | 159 | 0 | 1 | 1 |
| 197.parser | 0 | 0 | 0 | 0 | 1 | 1 |
| 252.eon | 17 | 0 | 17 | 0 | 183 | 215 |
| 253.perlbmk | 1,454 | 229 | 1,454 | 24 | 243 | 261 |
| 254.gap | 63 | 5 | 63 | 2 | 1,438 | 7,836 |
| 255.vortex | 247 | 56 | 247 | 0 | 24 | 27 |
| 256.bzip2 | 0 | 0 | 0 | 0 | 1 | 1 |
| 300.twolf | 17 | 0 | 17 | 0 | 1 | 1 |
| ***Avg.*** | ***470*** | ***54*** | ***470*** | ***4*** | ***176*** | ***767*** |

# Evaluation: Identify Indirect Control Flow Targets

***IDA*** is a widely used commercial disassembling tools.

| PROGRAM | #INDIRECT JUMP EDGES | | | #INDIRECT CALL EDGES | | |
|---|---|---|---|---|---|---|
| | IDA | REFER | BDA | IDA | REFER | BDA |
| 164.gzip | 0 | 0 | 0 | 0 | 3 | 3 |
| 175.vpr | 49 | 0 | 49 | 0 | 1 | 1 |
| 176.gcc | 3,628 | 324 | 3,628 | 25 | 214 | 853 |
| 181.mcf | 0 | 0 | 0 | 0 | 1 | 1 |
| 186.crafty | 159 | 38 | 159 | 0 | 1 | 1 |
| 197.parser | 0 | 0 | 0 | 0 | 1 | 1 |
| 252.eon | 17 | 0 | 17 | 0 | 183 | 215 |
| 253.perlbmk | 1,454 | 229 | 1,454 | 24 | 243 | 261 |
| 254.gap | 63 | 5 | 63 | 2 | 1,438 | 7,836 |
| 255.vortex | 247 | 56 | 247 | 0 | 24 | 27 |
| 256.bzip2 | 0 | 0 | 0 | 0 | 1 | 1 |
| 300.twolf | 17 | 0 | 17 | 0 | 1 | 1 |
| ***Avg.*** | ***470*** | ***54*** | ***470*** | ***4*** | ***176*** | ***767*** |

# Evaluation: Identify Indirect Control Flow Targets

***IDA*** is a widely used commercial disassembling tools.

| PROGRAM | #INDIRECT JUMP EDGES | | | #INDIRECT CALL EDGES | | |
|---|---|---|---|---|---|---|
| | IDA | REFER | BDA | IDA | REFER | BDA |
| 164.gzip | 0 | 0 | 0 | 0 | 3 | 3 |
| 175.vpr | 49 | 0 | 49 | 0 | 1 | 1 |
| 176.gcc | 3,628 | 324 | 3,628 | 25 | 214 | 853 |
| 181.mcf | 0 | 0 | 0 | 0 | 1 | 1 |
| 186.crafty | 159 | 38 | 159 | 0 | 1 | 1 |
| 197.parser | 0 | 0 | 0 | 0 | 1 | 1 |
| 252.eon | 17 | 0 | 17 | 0 | 183 | 215 |
| 253.perlbmk | 1,454 | 229 | 1,454 | 24 | 243 | 261 |
| 254.gap | 63 | 5 | 63 | 2 | 1,438 | 7,836 |
| 255.vortex | 247 | 56 | 247 | 0 | 24 | 27 |
| 256.bzip2 | 0 | 0 | 0 | 0 | 1 | 1 |
| 300.twolf | 17 | 0 | 17 | 0 | 1 | 1 |
| ***Avg.*** | ***470*** | ***54*** | ***470*** | ***4*** | ***176*** | ***767*** |

# Evaluation: Identify Indirect Control Flow Targets

BDA performs as good as IDA in inferring indirect jump targets. (***470*** in average)

BDA reports ***767*** indirect call edges, without missing any observer ones .

| PROGRAM | #INDIRECT JUMP EDGES | | | #INDIRECT CALL EDGES | | |
|---|---|---|---|---|---|---|
| | IDA | REFER | BDA | IDA | REFER | BDA |
| 164.gzip | 0 | 0 | 0 | 0 | 3 | 3 |
| 175.vpr | 49 | 0 | 49 | 0 | 1 | 1 |
| 176.gcc | 3,628 | 324 | 3,628 | 25 | 214 | 853 |
| 181.mcf | 0 | 0 | 0 | 0 | 1 | 1 |
| 186.crafty | 159 | 38 | 159 | 0 | 1 | 1 |
| 197.parser | 0 | 0 | 0 | 0 | 1 | 1 |
| 252.eon | 17 | 0 | 17 | 0 | 183 | 215 |
| 253.perlbmk | 1,454 | 229 | 1,454 | 24 | 243 | 261 |
| 254.gap | 63 | 5 | 63 | 2 | 1,438 | 7,836 |
| 255.vortex | 247 | 56 | 247 | 0 | 24 | 27 |
| 256.bzip2 | 0 | 0 | 0 | 0 | 1 | 1 |
| 300.twolf | 17 | 0 | 17 | 0 | 1 | 1 |
| ***Avg.*** | ***470*** | ***54*** | ***470*** | ***4*** | ***176*** | ***767*** |

# Evaluation: Malware Analysis

- Malware behavior is largely defined by its system and library calls, together with parameter values.

- BDA performs static constant propagation through dependence, to identify the parameter values.

# Evaluation: Malware Analysis

***Cuckoo*** is the state-of-the-art malware analysis tool.

BDA reports ***3*** times more hidden malicious behaviors than cuckoo.

| MD5 OF MALWARE | REPORT DATE | #LIBRARY CALL | |
|---|---|---|---|
| | | CUCKOO | BDA |
| 1a0b96488c4be390ce2072735ffb0e49 | 2019-01-22 | 50 | 164 |
| 3fb857173602653861b4d0547a49b395 | 2018-07-24 | 20 | 112 |
| 49c178976c50cf77db3f6234efce5eeb | 2019-01-23 | 23 | 48 |
| 5e890cb3f6cba8168d078fdede090996 | 2019-01-25 | 28 | 138 |
| 6dc1f557eac7093ee9e5807385dbcb05 | 2018-12-23 | 20 | 75 |
| 72afccb455faa4bc1e5f16ee67c6f915 | 2019-07-02 | 6 | 81 |
| 74124dae8fdbb903bece57d5be31246b | 2019-03-21 | 36 | 203 |
| 912bca5947944fdcd09e9620d7aa8c4a | 2019-10-04 | 20 | 68 |
| a664df72a34b863fc0a6e04c96866d4c | 2018-12-20 | 23 | 99 |
| c38d08b904d5e1c7c798e840f1d8f1ee | 2018-08-28 | 34 | 151 |
| c63cef04d931d8171d0c40b7521855e9 | 2019-01-23 | 20 | 81 |
| dc4db38f6d3c1e751dcf06bea072ba9c | 2018-10-23 | 20 | 77 |
| ***Avg.*** | ***/*** | ***25*** | ***108*** |

# Evaluation: Malware Analysis

- open("/etc/passwd", O_RDONLY)
  → Read user password
- system("/bin/sh")
  → Return a remote shell
- system("rm -rf /")
  → Remove all the files on disk

BDA reports ***3*** times more hidden malicious behaviors than cuckoo.

| | REPORT DATE | #LIBRARY CALL | |
|---|---|---|---|
| | | CUCKOO | BDA |
| | 2019-01-22 | 50 | 164 |
| | 2018-07-24 | 20 | 112 |
| | 2019-01-23 | 23 | 48 |
| | 2019-01-25 | 28 | 138 |
| 6dc1f557eac7093ee9[illegible]cb05 | 2018-12-23 | 20 | 75 |
| 72afccb455faa4bc1e5f16ee67c[illegible] | 2019-07-02 | 6 | 81 |
| 74124dae8fdbb903bece57d5be31246b | 2019[illegible]-21 | 36 | 203 |
| 912bca5947944fdcd09e9620d7aa8c4a | 2019-10-04 | 20 | 68 |
| a664df72a34b863fc0a6e04c96866d4c | 2018-12-20 | 23 | 99 |
| c38d08b904d5e1c7c798e840f1d8f1ee | 2018-08-28 | 34 | 151 |
| c63cef04d931d8171d0c40b7521855e9 | 2019-01-23 | 20 | 81 |
| dc4db38f6d3c1e751dcf06bea072ba9c | 2018-10-23 | 20 | 77 |
| ***Avg.*** | ***/*** | ***25*** | ***108*** |

# Closely Related Work

Random abstract interpretation

- Discovering affine equalities using random interpretation [POPL 03]
- Global value numbering using random interpretation [POPL 04]
- Precise inter-procedural analysis using random interpretation [POPL 05]

Path encoding

- Efficient path profiling [MICRO 96]
- Precise Calling Context Encoding [ICSE 10]

Reducing the runtime complexity of path-sensitive analysis

- ESP: Path-sensitive program verification in polynomial time [PLDI 02]
- Sound, complete and scalable path-sensitive analysis [PLDI 08]

# Conclusion

- We propose a practical program dependence analysis for binary executables
  - A novel unbiased whole-program path sampling algorithm
  - A per-path abstract interpretation
  - Probabilistic guarantees in disclosing a dependence relation
- Result
  - Improve the state-of-the-art, such as Value Set Analysis
  - Improve performance of downstream applications

# Thank you!

# Q&A

BDA Repo
https://github.com/bda-tool/bda/

Email Address
zhan3299@purdue.edu