

Directed Symbolic Execution

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks

University of Maryland, College Park

SAS 2011

Motivation

▶ Line reachability problem

- ▶ Given a **target line** in the program, can we find an **input** that drives the program to that line?
- ▶ Equivalent to: find an input that causes the program to **enter a particular state** expressed as a conditional
 - ▶ E.g., `if (i > MAX) assert (0) ;`

▶ Applications

- ▶ **Reproduce the error** in a bug report
 - ▶ Target line from stack trace
- ▶ Triage errors reported by static analysis
 - ▶ Target line from error report: detect **false positives**
- ▶ Improve code understanding

Symbolic Execution

- ▶ Systematically explore execution paths
 - ▶ Interpret programs with **symbolic values**
 - ▶ At branch points, query a **constraint solver**
 - ▶ If both branches are feasible, **fork** the execution
 - ▶ **NO abstraction; path sensitive; NO false positives**
 - ▶ However, infeasible to explore all paths in a program
 - ▶ Guide the search based on **strategies**, e.g.,
 - ❖ *Random-path*: probability of exploring a path with length $n \propto 1/2^n$
 - ❖ *KLEE* [OSDI08]: random-path + bias towards uncovered code
 - ❖ *SAGE* [NDSS08]: coverage-guided generational search
- These strategies are aimed at **maximizing coverage**

Directed Symbolic Execution

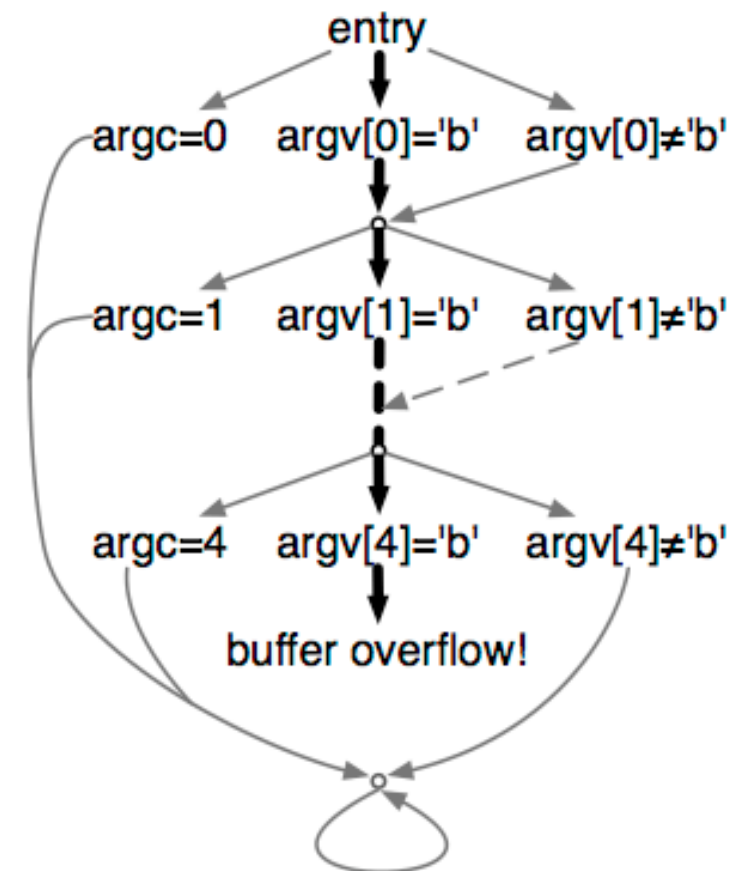
- ▶ Guide the search towards a target line
- ▶ Our contributions:
 - ▶ **Shortest-Distance Symbolic Execution (SDSE)**
 - ▶ Bias the search using *inter-procedural CFG-distances* to target lines
 - ▶ **Call-Chain-Backward Symbolic Execution (CCBSE)**
 - ▶ Start symbolic execution *in the function* containing the target line;
 - ▶ Set the function as *target*, and repeat this *backward up the call chain*, until we find a feasible path from the start of the program.
 - ▶ **Mixing CCBSE with a forward strategy (Mix-CCBSE)**
 - ▶ A forward strategy is any of RP, KLEE, SAGE, SDSE, etc.

Shortest-Distance Symbolic Execution (SDSE)

Shortest-Distance Symbolic Execution (SDSE): Example

```
1  int main(void) {  
2    int argc; char argv[MAX_ARGC][1];  
3    symbolic(&argc); symbolic(&argv);  
4    int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5    for (i = 0; i < argc; i++) {  
6      if (*argv[i] == 'b') {  
7  
8  
9        b[n++] = 1; /* potential buf. overflow */  
10     } else  
11       foo(); /* some expensive function */  
12   }  
13   while (1) {  
14     if (getchar()) /* get symbolic input */  
15       /* ...do something... */;  
16   }  
17   return 0;  
18 }
```

First 5 iterations:

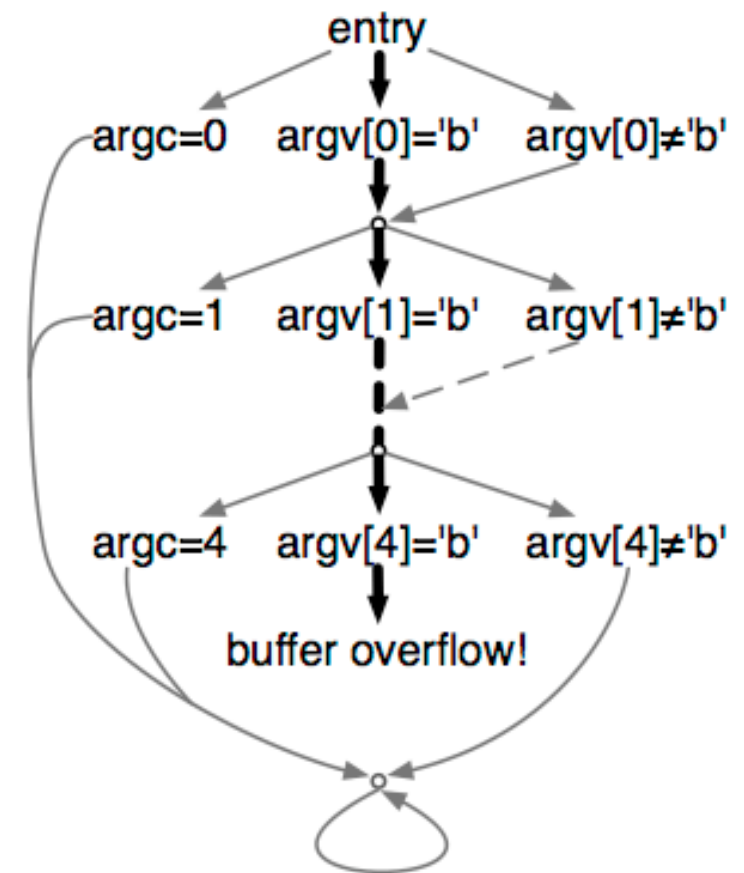


Shortest-Distance Symbolic Execution (SDSE): Example

```
1 int main(void) {  
2   int argc; char argv[MAX_ARGC][1];  
3   symbolic(&argc); symbolic(&argv);  
4   int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5   for (i = 0; i < argc; i++) {  
6     if (*argv[i] == 'b') {  
7  
8  
9       b[n++] = 1; /* potential buf. overflow */  
10    } else  
11      foo(); /* some expensive function */  
12  }  
13  while (1) {  
14    if (getchar()) /* get symbolic input */  
15      /* ...do something... */;  
16  }  
17  return 0;  
18 }
```

Program input,
made symbolic

First 5 iterations:

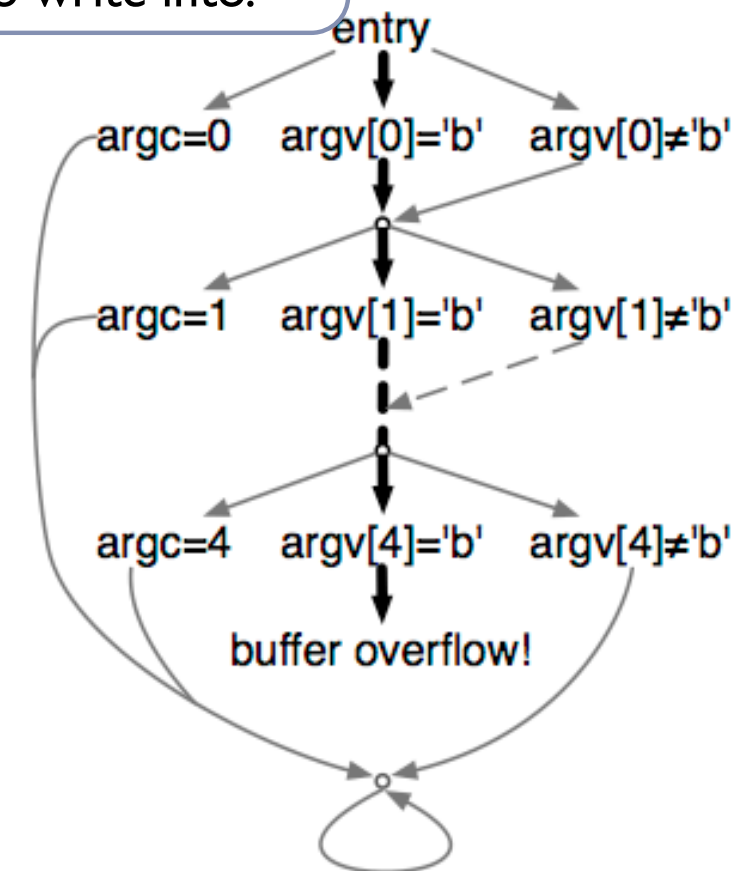


Shortest-Distance Symbolic Execution (SDSE): Example

```
1 int main(void) {  
2   int argc; char argv[MAX_ARGC][1];  
3   symbolic(&argc); symbolic(&argv);  
4   int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5   for (i = 0; i < argc; i++) {  
6     if (*argv[i] == 'b') {  
7  
8  
9       b[n++] = 1; /* potential buf. overflow */  
10    } else  
11      foo(); /* some expensive function */  
12  }  
13  while (1) {  
14    if (getchar()) /* get symbolic input */  
15      /* ...do something... */;  
16  }  
17  return 0;  
18 }
```

b is a buffer of size 4.
n is the next position
in b to write into.

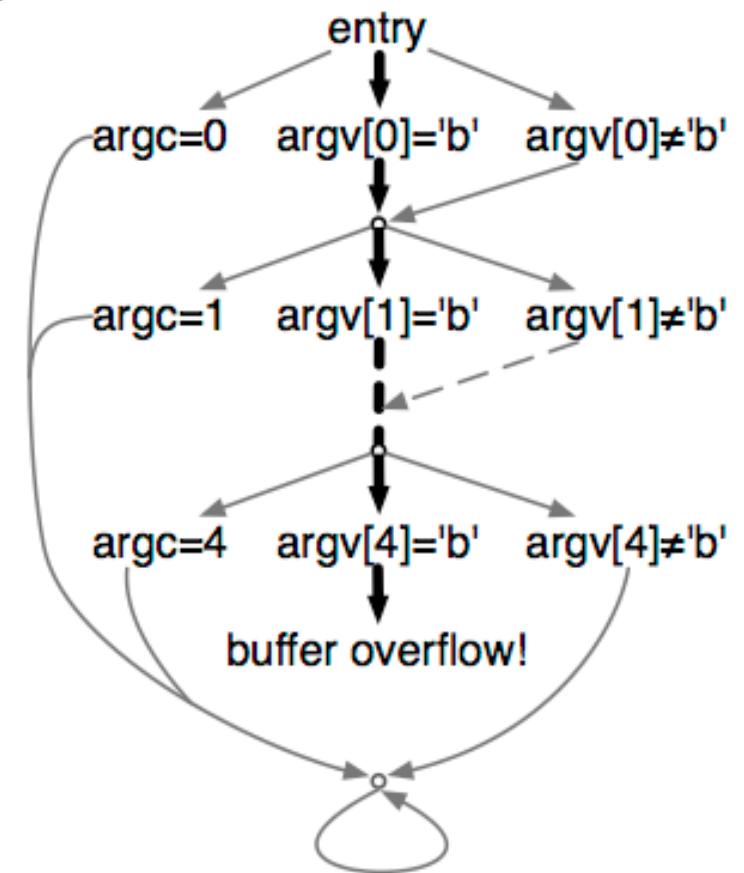
Iterations:



Shortest-Distance Symbolic Execution (SDSE): Example

```
1 int main(void) {  
2   int argc; char argv[MAX_ARGC][1];  
3   symbolic(&argc); symbolic(&argv);  
4   int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5   for (i = 0; i < argc; i++) {  
6     if (*argv[i] == 'b') {  
7  
8  
9       b[n++] = 1; /* potential buf. overflow */  
10    } else  
11      foo(); /* some expensive function */  
12  }  
13  while (1) {  
14    if (getchar()) /* get symbolic input */  
15      /* ...do something... */;  
16  }  
17  return 0;  
18 }
```

a for-loop
iterating over argv at 5 iterations:



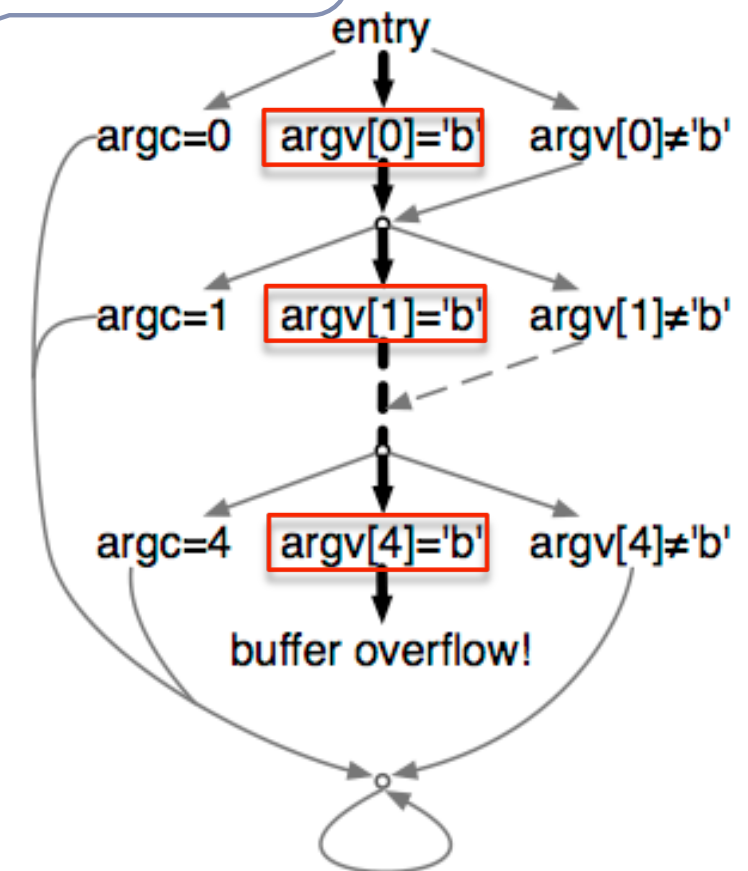
Shortest-Distance Symbolic Execution (SDSE): Example

```

1  int main(void) {
2      int argc; char argv[MAX_ARGC][1];
3      symbolic(&argc); symbolic(&argv);
4      int i, n = 0, b[4] = { 0, 0, 0, 0 };
5      for (i = 0; i < argc; i++) {
6          if (*argv[i] == 'b') {
7              // ...
8              // ...
9              b[n++] = 1; /* potential buf. overflow */
10         } else
11             foo(); /* some expensive function */
12     }
13     while (1) {
14         if (getchar()) /* get symbolic input */
15             /* ...do something... */;
16     }
17     return 0;
18 }

```

if the argument is a letter b, put a 1 into b iterations:

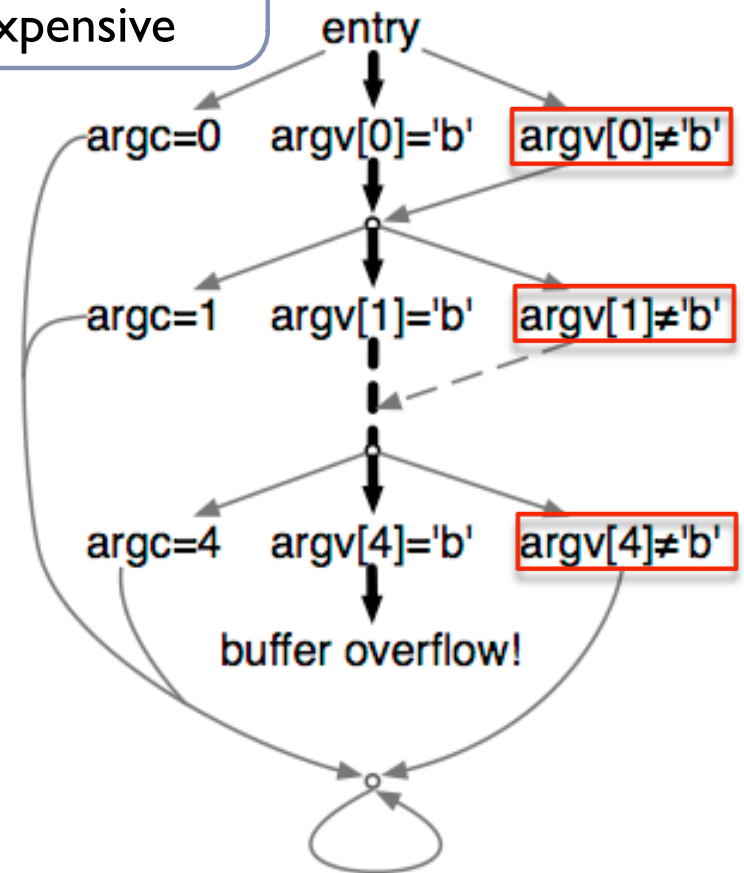


Shortest-Distance Symbolic Execution (SDSE): Example

```
1 int main(void) {  
2   int argc; char argv[MAX_ARGC][1];  
3   symbolic(&argc); symbolic(&argv);  
4   int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5   for (i = 0; i < argc; i++) {  
6     if (*argv[i] == 'b') {  
7  
8  
9       b[n++] = 1; /* potential buf. overflow */  
10    } else  
11    { foo(); /* some expensive function */  
12  }  
13  while (1) {  
14    if (getchar()) /* get symbolic input */  
15      /* ...do something... */;  
16  }  
17  return 0;  
18 }
```

if the argument is NOT
a letter b, run foo()
which is expensive

at 5 iterations:

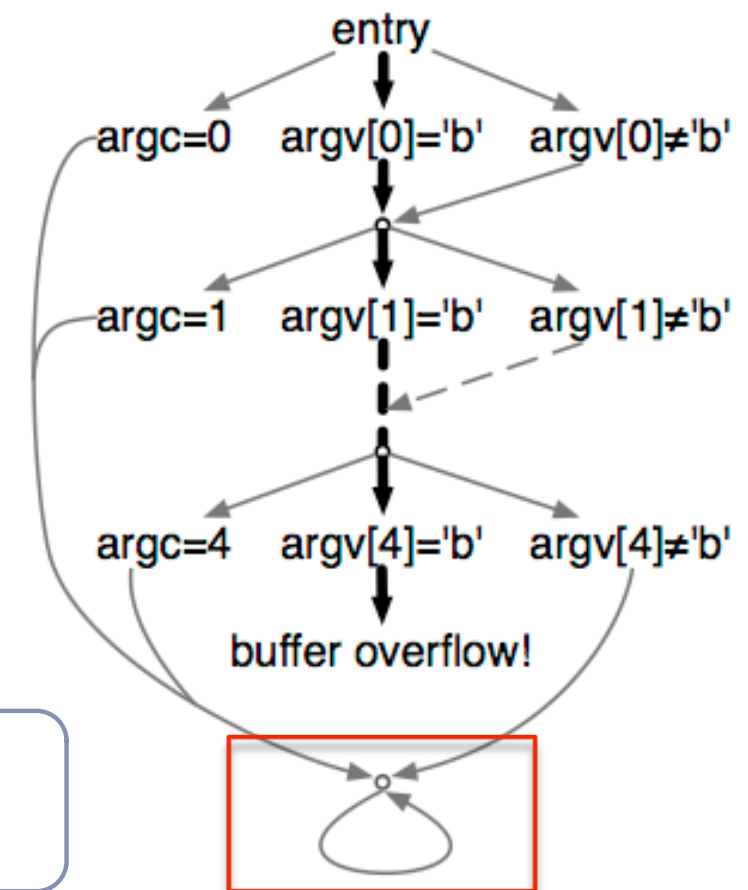


Shortest-Distance Symbolic Execution (SDSE): Example

```
1  int main(void) {  
2    int argc; char argv[MAX_ARGC][1];  
3    symbolic(&argc); symbolic(&argv);  
4    int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5    for (i = 0; i < argc; i++) {  
6      if (*argv[i] == 'b') {  
7  
8  
9        b[n++] = 1; /* potential buf. overflow */  
10     } else  
11     foo(); /* some expensive function */  
12   }  
13   while (1) {  
14     if (getchar()) /* get symbolic input */  
15       /* ...do something... */;  
16   }  
17   return 0;  
18 }
```

an infinite loop,
doing work

First 5 iterations:

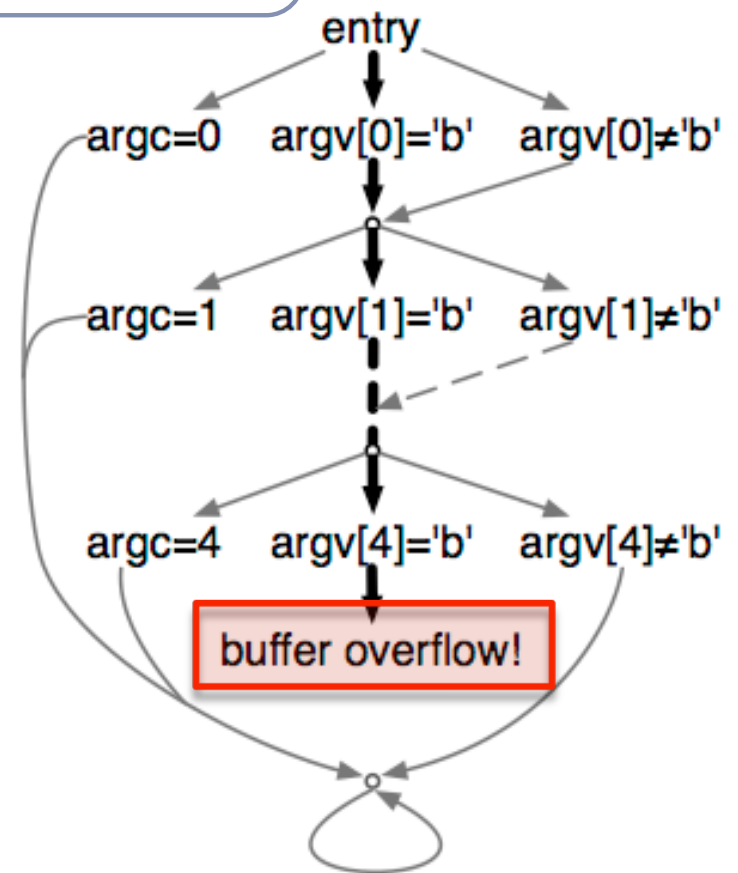


Shortest-Distance Symbolic Execution (SDSE): Example

```
1 int main(void) {  
2   int argc; char argv[MAX_ARGC][1];  
3   symbolic(&argc); symbolic(&argv);  
4   int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5   for (i = 0; i < argc; i++) {  
6     if (*argv[i] == 'b') {  
7  
8  
9       b[n++] = 1; /* potential buf. overflow */  
10    } else  
11      foo(); /* some expensive function */  
12  }  
13  while (1) {  
14    if (getchar()) /* get symbolic input */  
15      /* ...do something... */;  
16  }  
17  return 0;  
18 }
```

buffer overflow occurs
when $n \geq 4$

5 iterations:

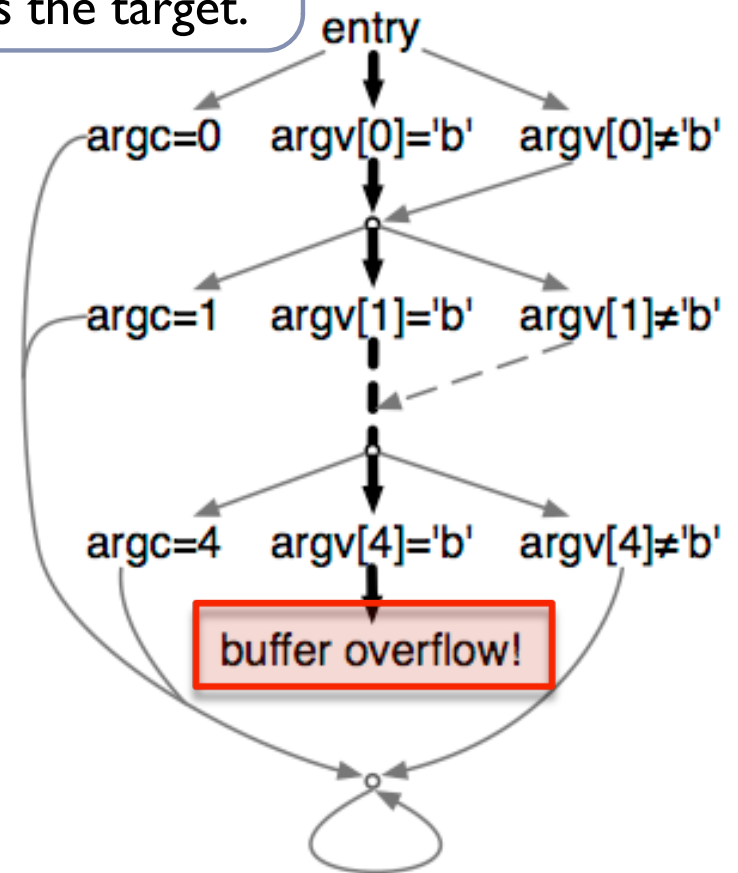


Shortest-Distance Symbolic Execution (SDSE): Example

```
1 int main(void) {  
2   int argc; char argv[MAX_ARGC][1];  
3   symbolic(&argc); symbolic(&argv);  
4   int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5   for (i = 0; i < argc; i++) {  
6     if (*argv[i] == 'b') {  
7       if (n >= 4)  
8         assert(0);  
9       b[n++] = 1; /* potential buf. overflow */  
10    } else  
11      foo(); /* some expensive function */  
12  }  
13  while (1) {  
14    if (getchar()) /* get symbolic input */  
15      /* ...do something... */;  
16  }  
17  return 0;  
18 }
```

A conditional to detect
buffer overflow.
assert(0) is the target.

5 iterations:

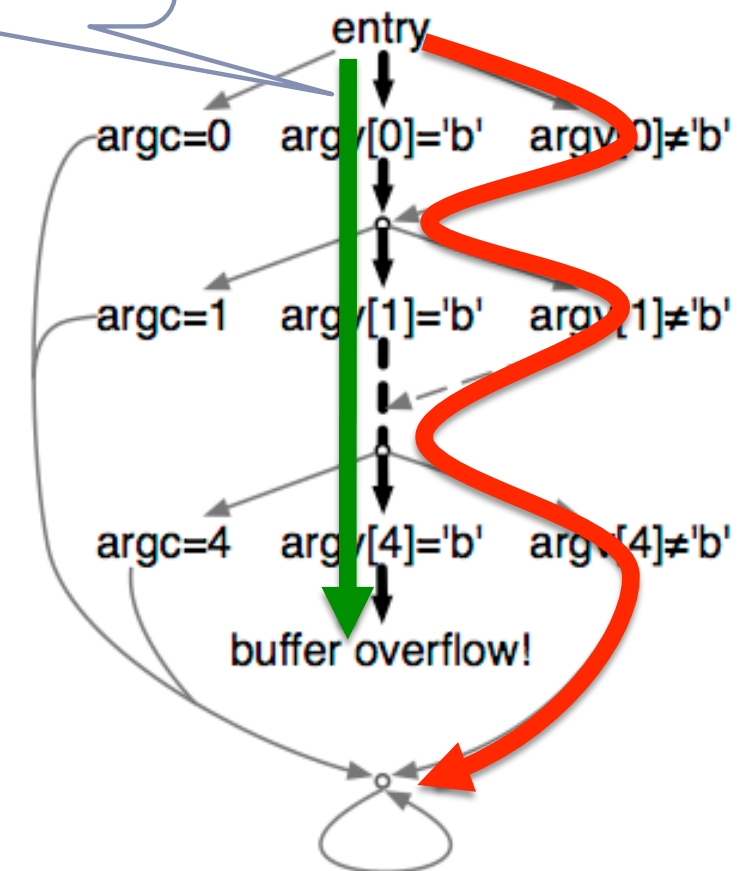


Shortest-Distance Symbolic Execution (SDSE): Example

```
1 int main(void) {
2   int argc; char argv[MAX_ARGC][1];
3   symbolic(&argc); symbolic(&argv);
4   int i, n = 0, b[4] = { 0, 0, 0, 0 };
5   for (i = 0; i < argc; i++) {
6     if (*argv[i] == 'b') {
7       if (n >= 4)
8         assert(0);
9       b[n++] = 1; /* potential buf. overflow */
10    } else
11      foo(); /* some expensive function */
12  }
13  while (1) {
14    if (getchar()) /* get symbolic input */
15      /* ...do something... */;
16  }
17  return 0;
18 }
```

Green path is the
shortest path

First 5 iterations:



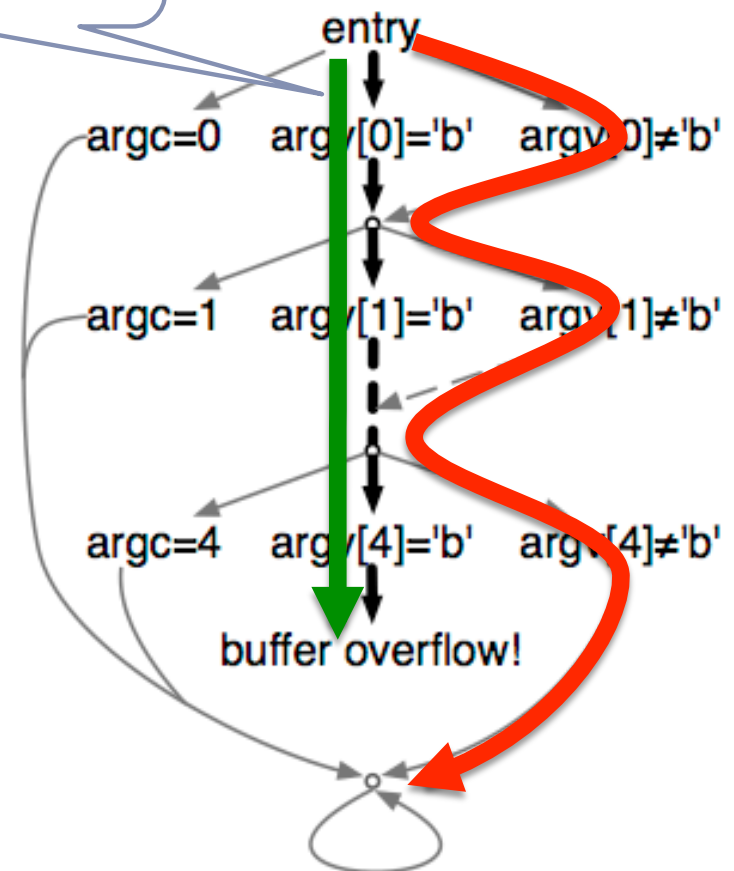
Shortest-Distance Symbolic Execution (SDSE): Example

Green path is the shortest path

First 5 iterations:

```
1 int main(void) {  
2   int argc; char argv[MAX_ARGC][1];  
3   symbolic(&argc); symbolic(&argv);  
4   int i, n = 0, b[4] = { 0, 0, 0, 0 };  
5   for (i = 0; i < argc; i++) {  
6     if (*argv[i] == 'b') {  
7       if (n >= 4)  
8         assert(0);  
9       b[n++] = 1; /* potential buf. overflow */  
10    } else  
11    foo(); /* some expensive function */  
12  }  
13  while (1) {  
14    if (getchar()) /* get symbolic input */  
15      /* ...do something... */;  
16  }  
17  return 0;  
18 }
```

distance = 2



Shortest-Distance Symbolic Execution (SDSE): Distance Calculation

main



bar

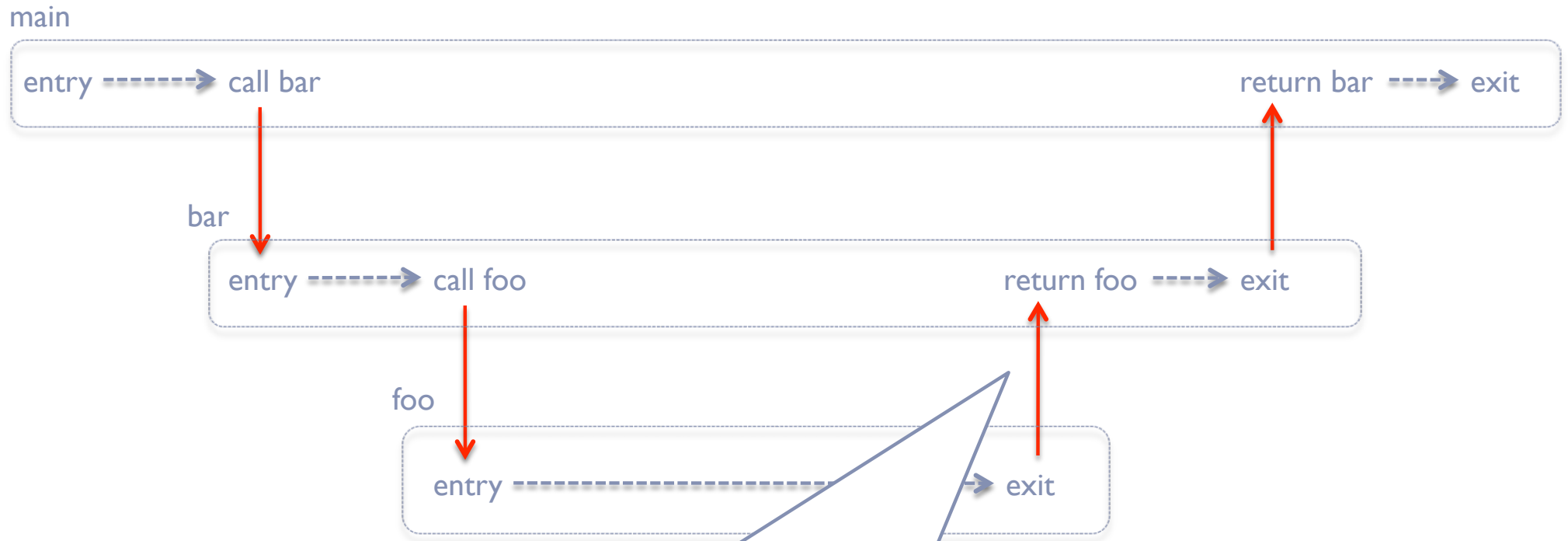


foo



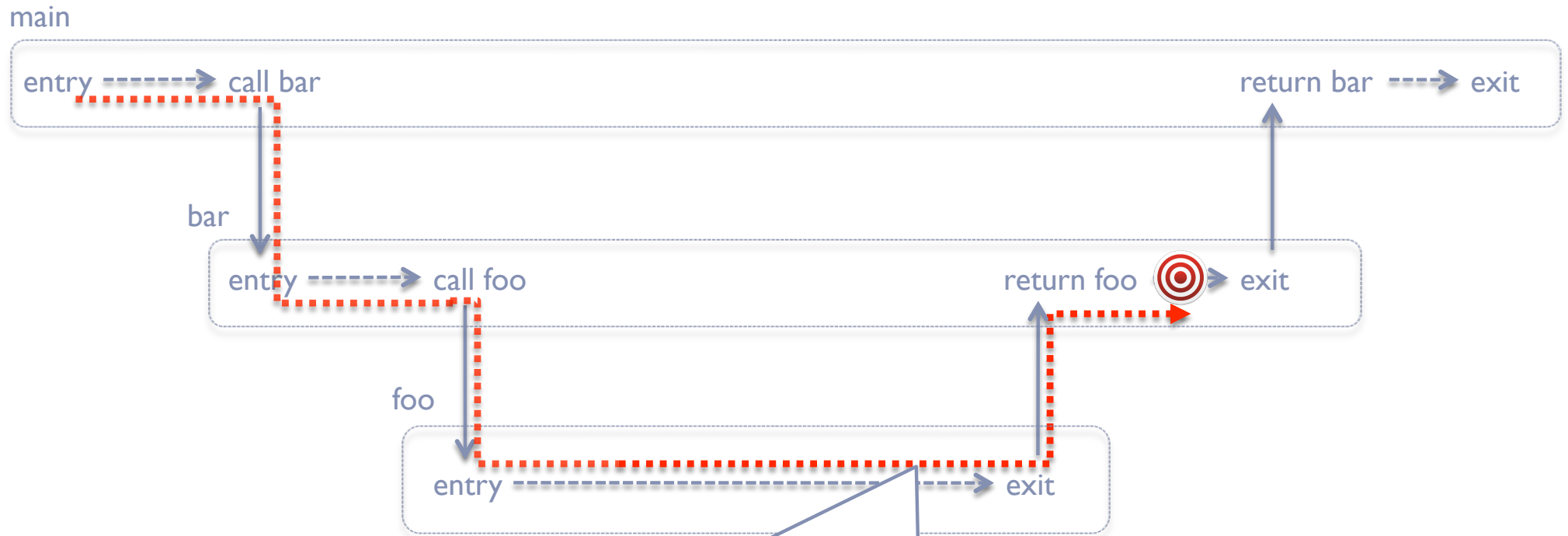
- ## 1. Build a CFG for each function

Shortest-Distance Symbolic Execution (SDSE): Distance Calculation



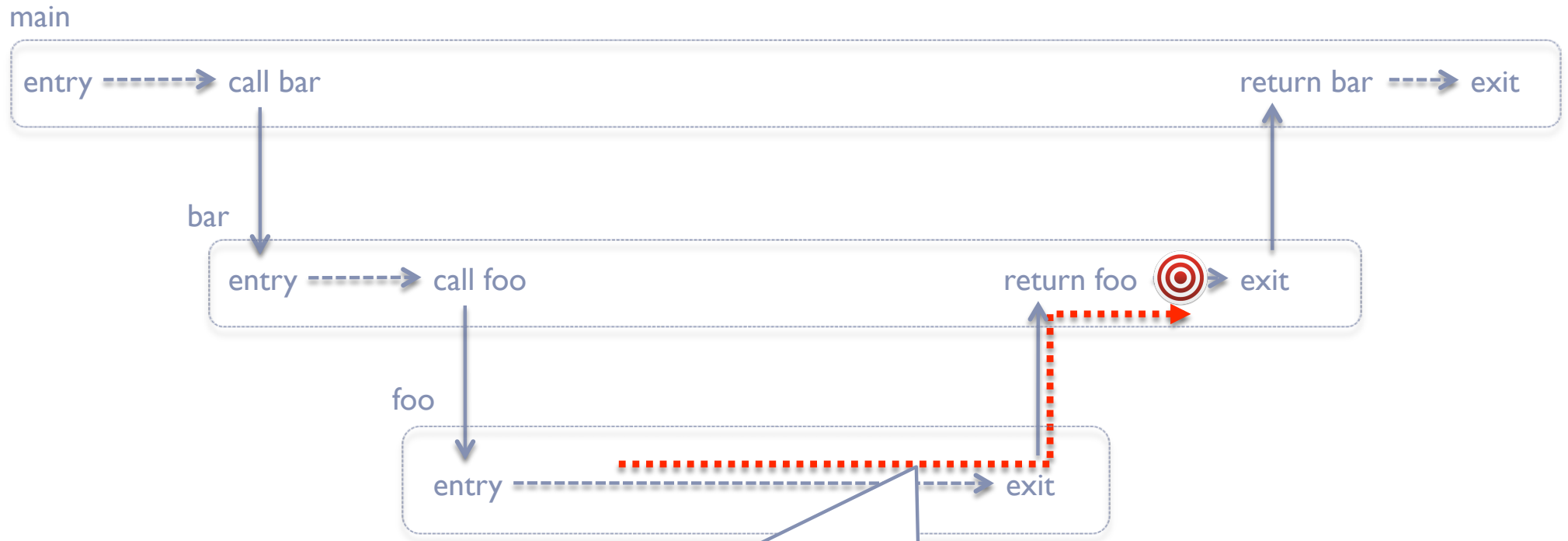
2. Build an inter-procedural CFG by splitting function calls into call and return nodes, and connecting them to function entries and exits.

Shortest-Distance Symbolic Execution (SDSE): Distance Calculation



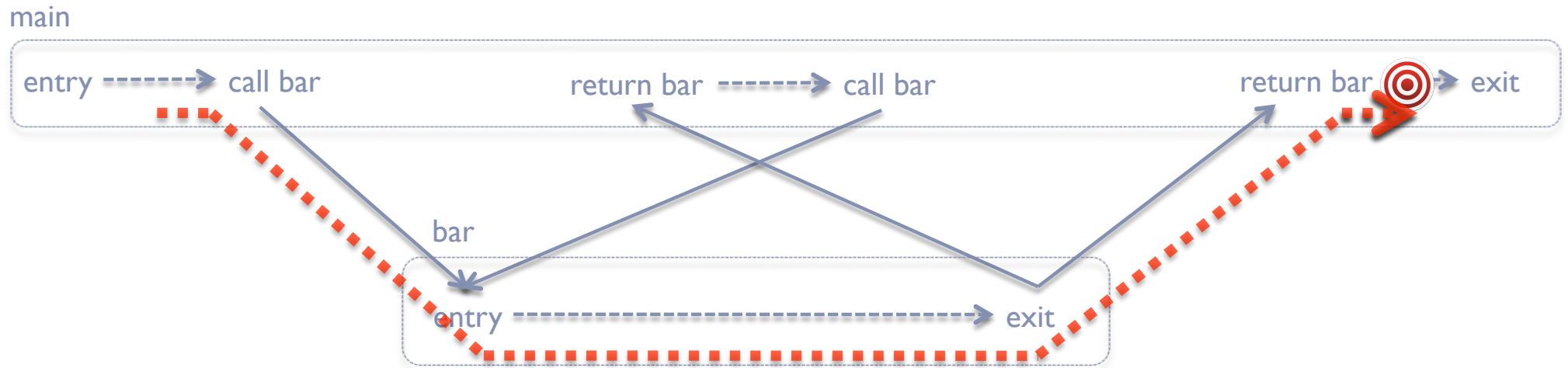
3. Compute shortest distances using CFL-reachability, specifically PN-paths.

Shortest-Distance Symbolic Execution (SDSE): Distance Calculation



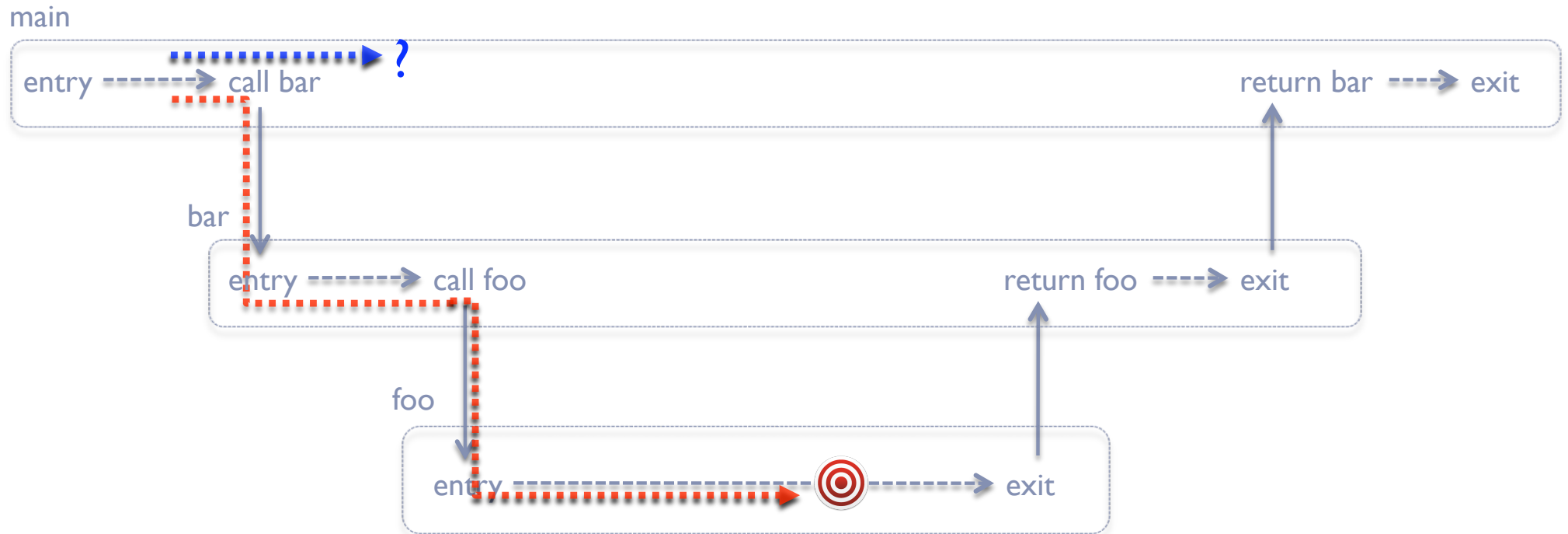
3. Compute shortest distances using CFL-reachability, specifically PN-paths.

Shortest-Distance Symbolic Execution (SDSE): Distance Calculation



Invalid PN-path, because path enters bar from one call site and returns to another call site.

Shortest-Distance Symbolic Execution (SDSE): Distance Calculation

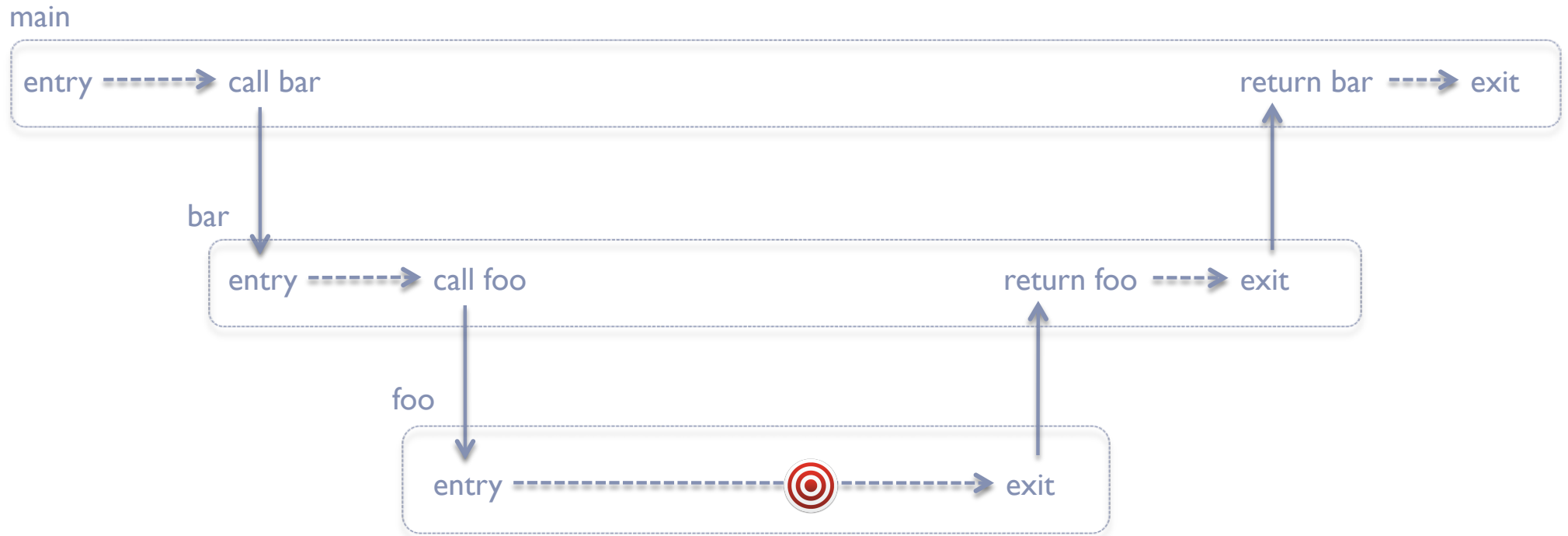


InterSDSE: use **inter**-procedural distance

IntraSDSE: use **intra**-procedural distance, where the distance to target is **infinity** when the target is not in the same function as the source

Call-Chain-Backward Symbolic Execution (CCBSE)

Call-Chain-Backward Symbolic Execution (CCBSE): Idea

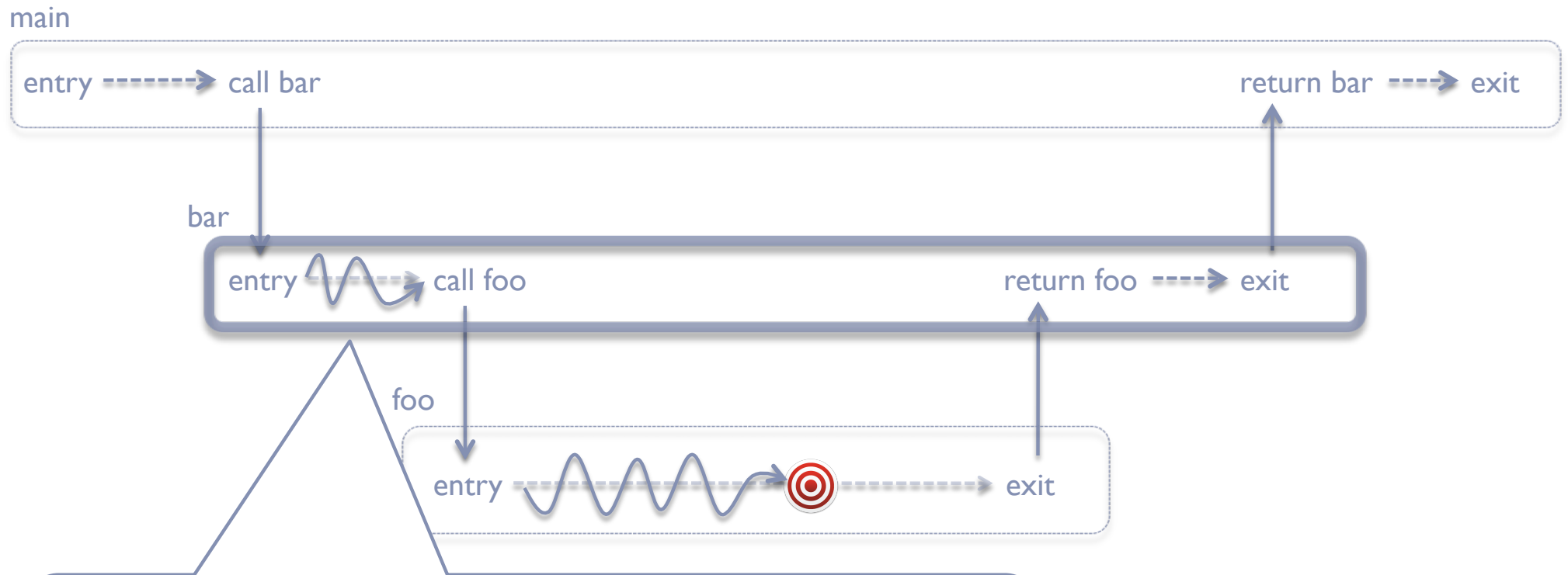


Call-Chain-Backward Symbolic Execution (CCBSE): Idea



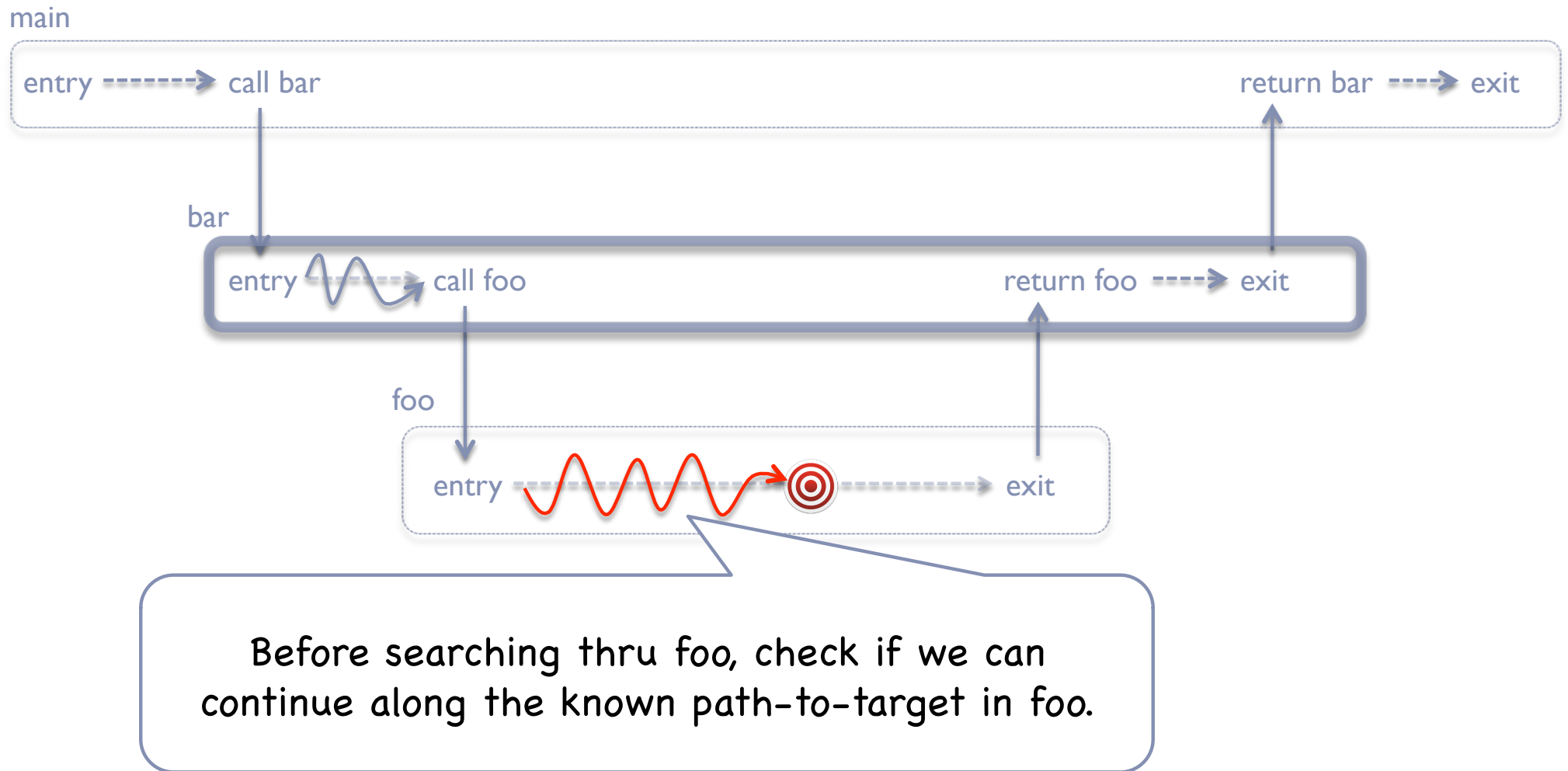
Start at function (foo) containing the target.
Run forward symbolic execution **using some strategy X** to find the target.

Call-Chain-Backward Symbolic Execution (CCBSE): Idea

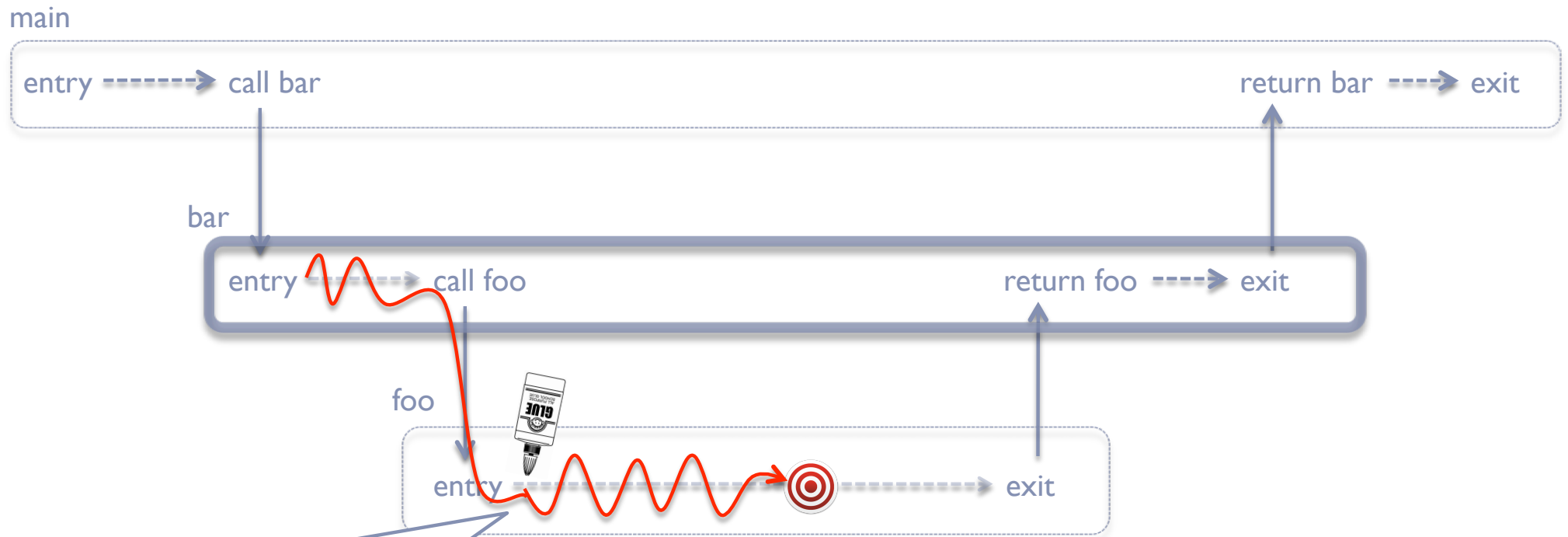


Once found, go back to its caller (bar). Run forward symbolic execution to find the call to foo.

Call-Chain-Backward Symbolic Execution (CCBSE): Idea



Call-Chain-Backward Symbolic Execution (CCBSE): Idea



If realizable, we immediately have a path-to-target from bar by stitching the two paths. We don't need to explore foo **again**.

Call-Chain-Backward Symbolic Execution (CCBSE): Idea



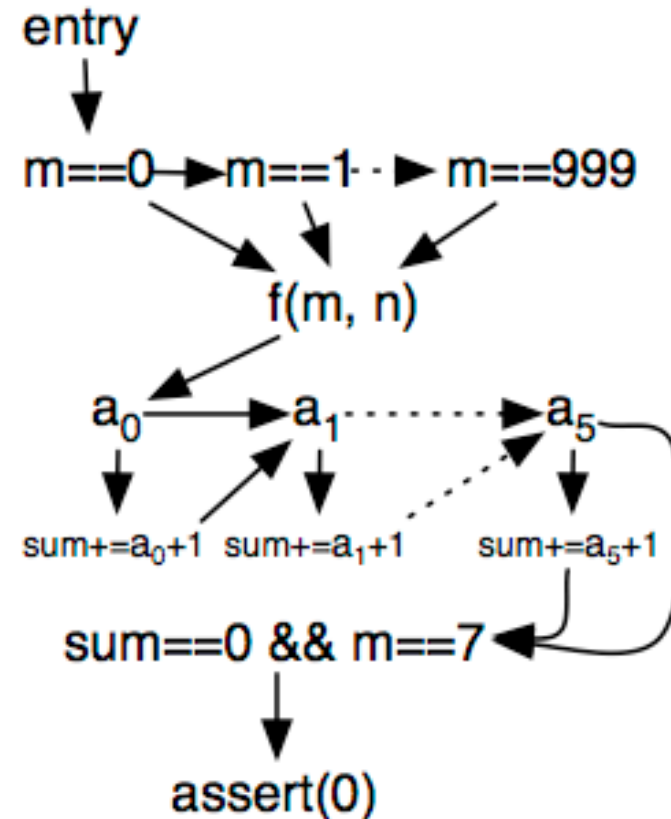
- ▶ Underlying forward strategy as a **parameter** of CCBSE: CCBSE(**X**)
- ▶ Paths as summaries: can **reuse the symbolic executor machinery**

Call-Chain-Backward Symbolic Execution (CCBSE): Requirement

- ▶ Require starting symbolic execution at **any function**
- ▶ Initialize function **parameters** and **global variables**
 - ▶ Primitive types: purely symbolic
 - ▶ **Pointers**: one of
 - ▶ NULL
 - ▶ A single object of the base type
 - ▶ For primitive base types, an array of objects of the base type
 - To model things like a C-string (**char***)
- ▶ **Lazy** initialization for arbitrarily recursive data structures
 - ▶ A pointer is not initialized until program “uses” it

Call-Chain-Backward Symbolic Execution (CCBSE): Example

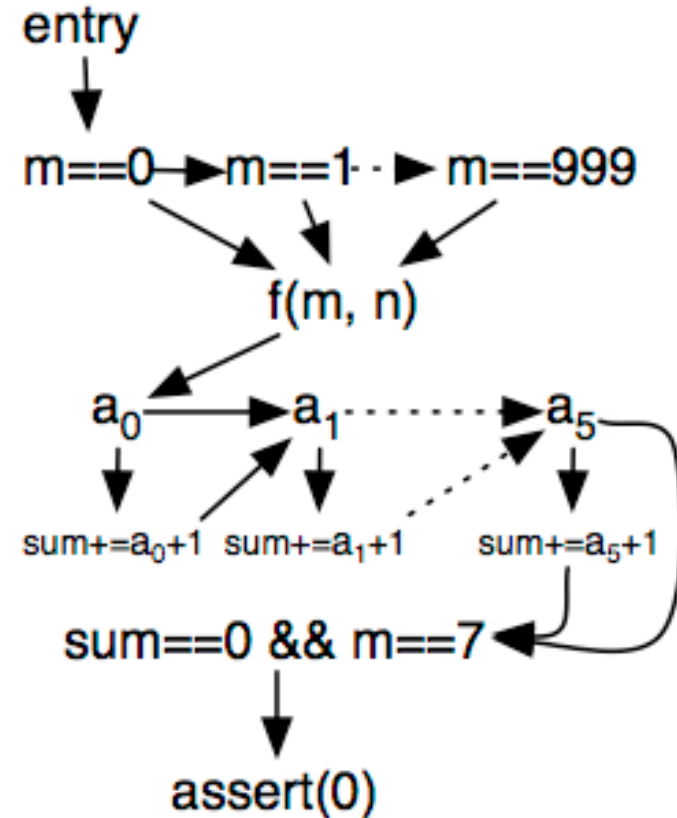
```
1 void main() {  
2   int m, n, i;  
3   symbolic(&m); symbolic(&n);  
4   for (i=0;i<1000;i++)  
5     if (m == i) f(m, n);  
6 }  
7 void f(int m, int n) {  
8   int i, a, sum=0;  
9   for (i=0;i<6;i++) {  
10    a = n%2;  
11    if (a) sum += a+1;  
12    n/=2;  
13  }  
14  while(1) {  
15    if (sum==0 && m==7)  
16      assert(0);  
17  }  
18 }
```



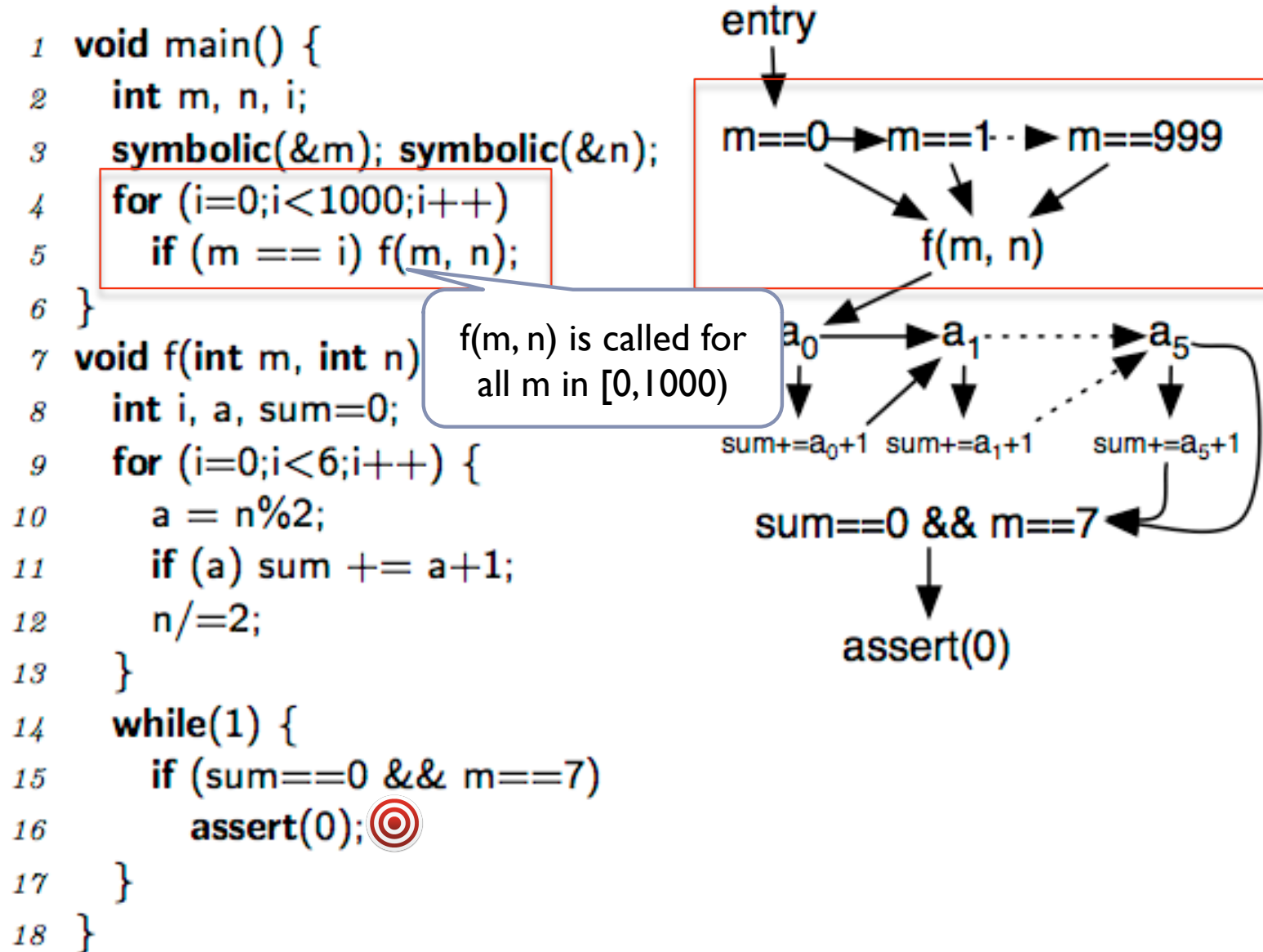
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```
1 void main() {  
    int m, n, i;  
    symbolic(&m); symbolic(&n);  
    for (i=0; i<1000; i++)  
        if (m == i) f(m, n);  
}  
7 void f(int m, int n) {  
    int i, a, sum=0;  
    for (i=0; i<6; i++) {  
        a = n%2;  
        if (a) sum += a+1;  
        n/=2;  
    }  
    while(1) {  
        if (sum==0 && m==7)  
            assert(0);  
    }  
}
```

m, n are input

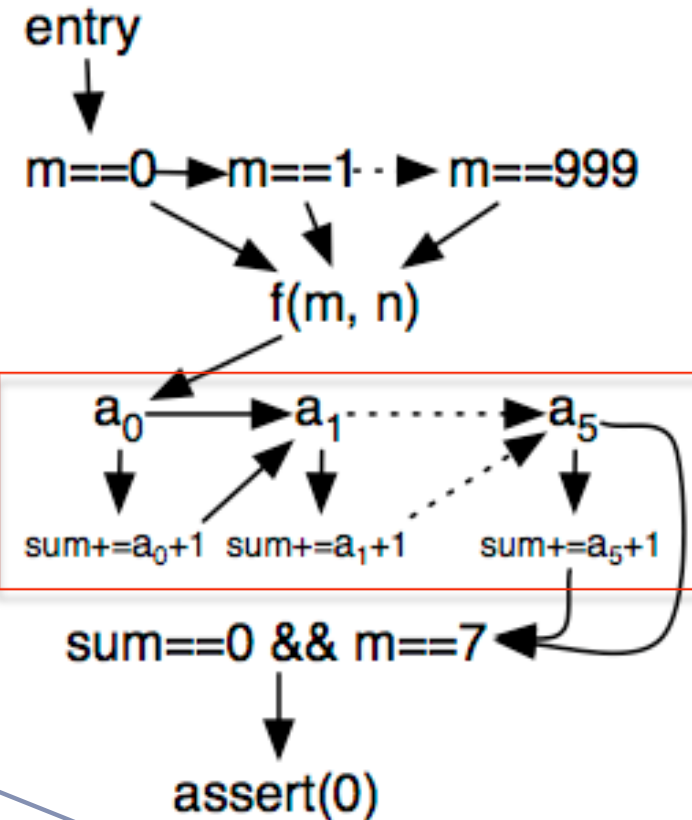


Call-Chain-Backward Symbolic Execution (CCBSE): Example



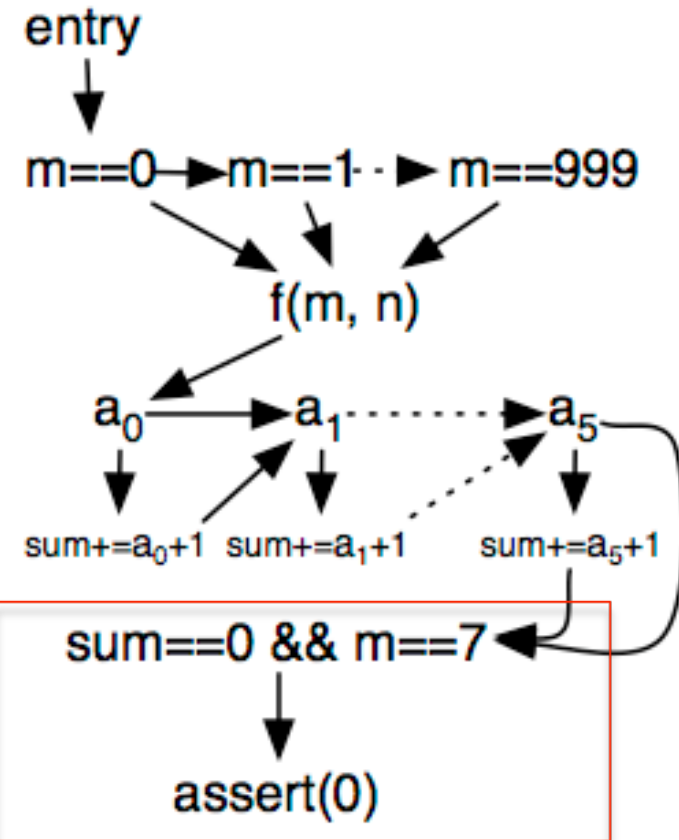
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```
1 void main() {
2   int m, n, i;
3   symbolic(&m); symbolic(&n);
4   for (i=0;i<1000;i++)
5     if (m == i) f(m, n);
6 }
7 void f(int m, int n) {
8   int i, a, sum=0;
9   for (i=0;i<6;i++) {
10    a = n%2;
11    if (a) sum += a+1;
12    n/=2;
13  }
14  while(1) {
15    if (sum==0 && m==7)
16      assert(0);
17  }
18 }
```



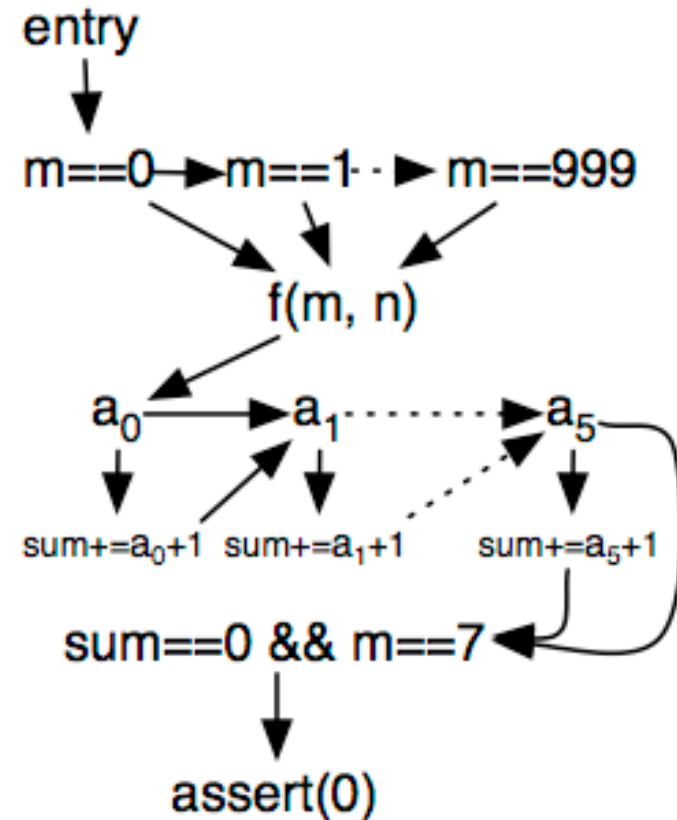
Call-Chain-Backward Symbolic Execution (CCBSE): Example

```
1 void main() {  
2   int m, n, i;  
3   symbolic(&m); symbolic(&n);  
4   for (i=0;i<1000;i++)  
5     if (m == i) f(m, n);  
6 }  
7 void f(int m, int n) {  
8   int i, a, sum=0;  
9   for (i=0;i<6;i++) {  
10    a = n%2;  
11    if (a) sum += a+1;  
12    n/=2;  
13  }  
14  while(1) {  
15    if (sum==0 && m==7)  
16      assert(0);  
17  }  
18 }
```



Call-Chain-Backward Symbolic Execution (CCBSE): Example

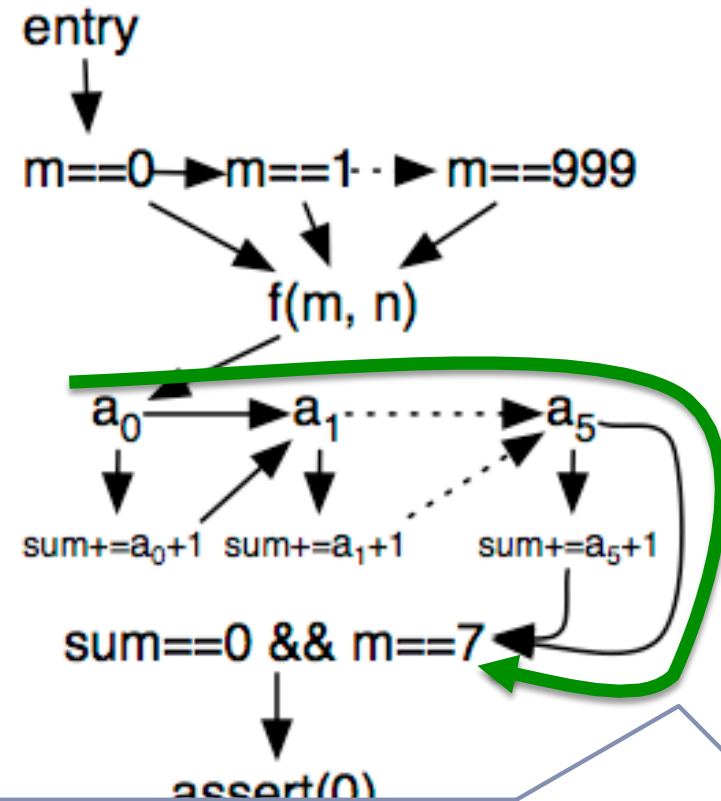
```
1 void main() {
2   int m, n, i;
3   symbolic(&m); symbolic(&n);
4   for (i=0;i<1000;i++)
5     if (m == i) f(m, n);
6 }
7 void f(int m, int n) {
8   int i, a, sum=0;
9   for (i=0;i<6;i++) {
10    a = n%2;
11    if (a) sum += a+1;
12    n/=2;
13  }
14  while(1) {
15    if (sum==0 && m==7)
16      assert(0);
17  }
18 }
```



Failing condition I: $m==7$;
directly determined by m in $f(m,n)$

Call-Chain-Backward Symbolic Execution (CCBSE): Example

```
1 void main() {  
2   int m, n, i;  
3   symbolic(&m); symbolic(&n);  
4   for (i=0;i<1000;i++)  
5     if (m == i) f(m, n);  
6 }  
7 void f(int m, int n) {  
8   int i, a, sum=0;  
9   for (i=0;i<6;i++) {  
10    a = n%2;  
11    if (a) sum += a+1;  
12    n/=2;  
13  }  
14  while(1) {  
15    if (sum==0 && m==7)  
16      assert(0);  
17  }  
18 }
```

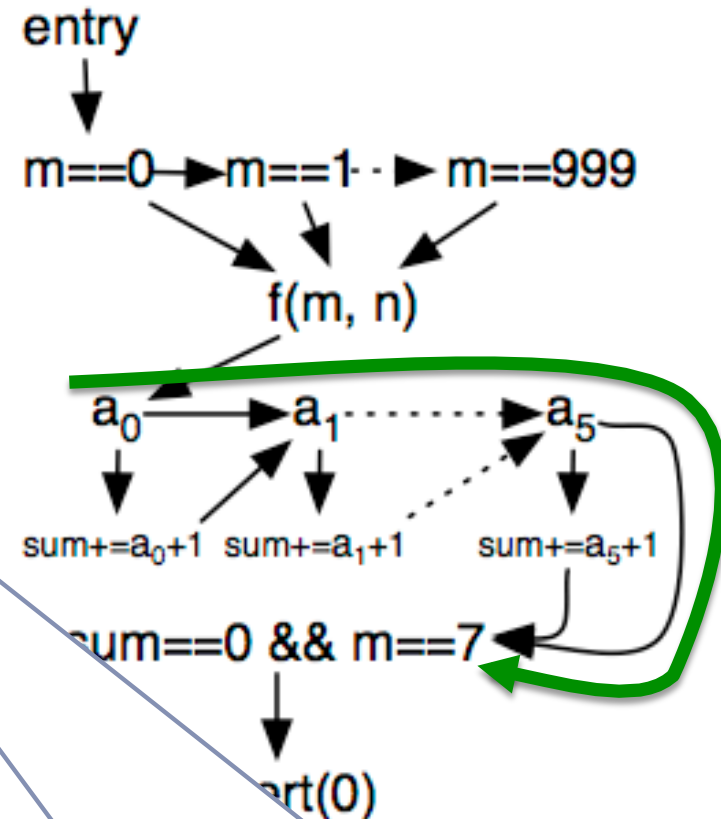


The only path among 2^6 paths that has `sum==0`: the true branch at line 11 is never taken.

Failing condition 2: `sum==0`; indirectly determined by `n` in `f(m,n)`

Call-Chain-Backward Symbolic Execution (CCBSE): Example

```
1 void main() {  
2   int m, n, i;  
3   symbolic(&m); symbolic(&n);  
4   for (i=0;i<1000;i++)  
5     if (m == i) f(m, n);  
6 }  
7 void f(int m, int n) {  
8   int i, a, sum=0;  
9   for (i=0;i<6;i++) {  
10    a = n%2;  
11    if (a) sum += a+1;  
12    n/=2;  
13  }  
14  while(1) {  
15    if (sum==0 && m==7)  
16      assert(0);  
17  }  
18 }
```



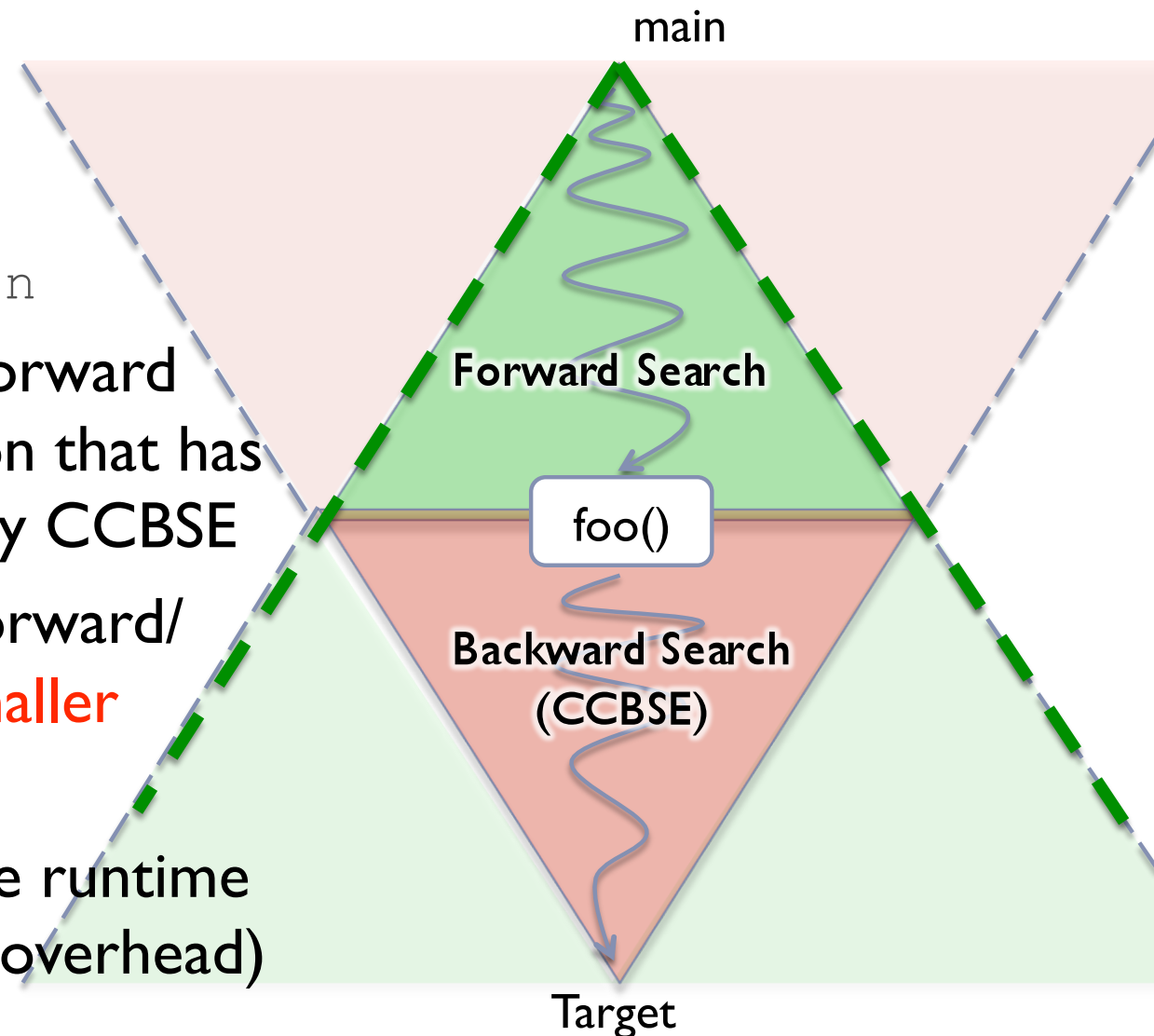
Every time $f(m, n)$ is called, CCBSE first checks if the green path is realizable. It's always realizable since n is unconstrained.

When $m==7$, the assertion is reachable.

CCBSE avoids re-exploring the 2^6 paths in f .

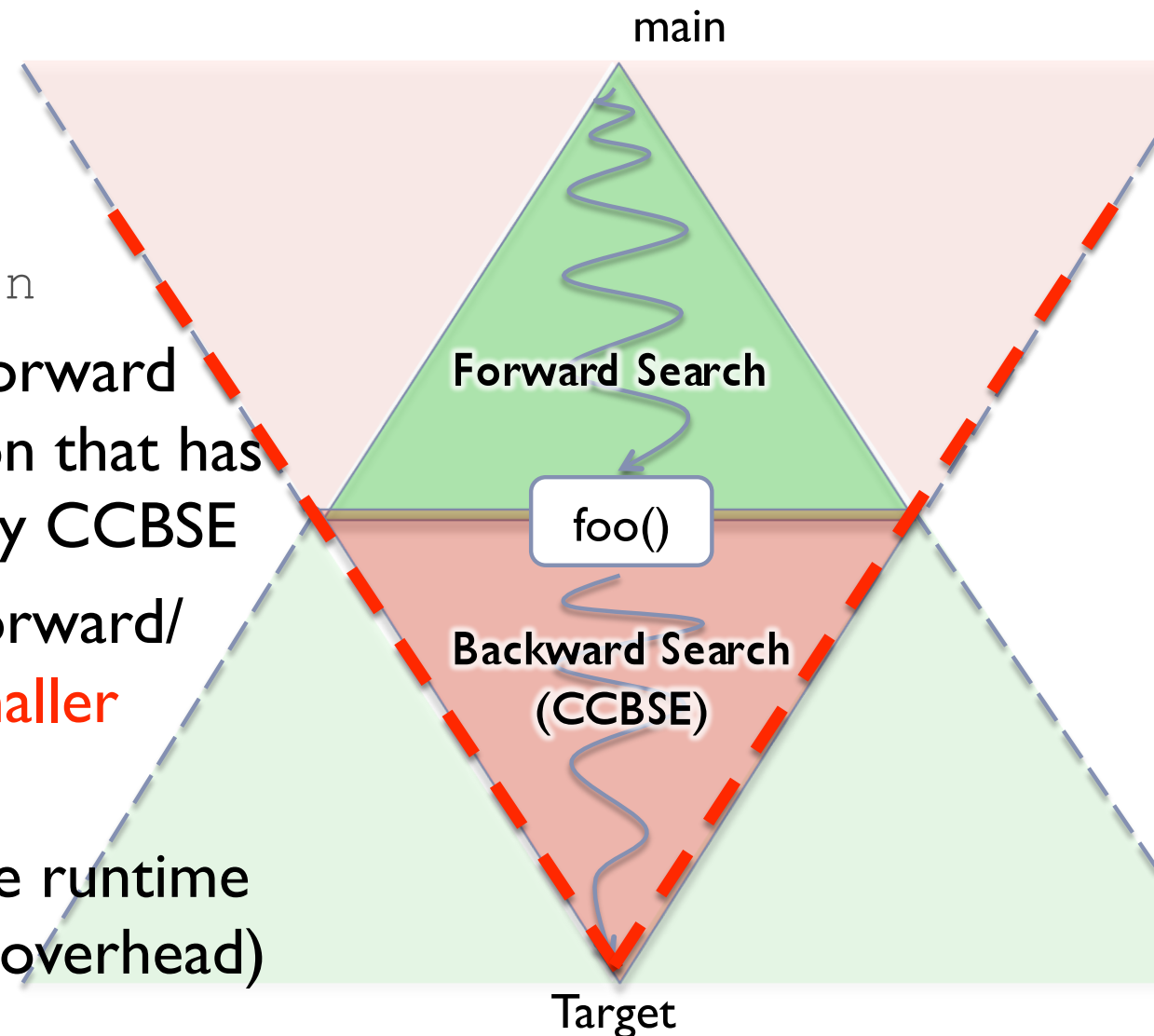
Mixing CCBSE with Forward Strategies

- ▶ **Simultaneously** run
 - ▶ CCBSE
 - ▶ Forward search from `main`
- ▶ **Try merging** when the forward search reaches a function that has a path to target found by CCBSE
- ▶ Best case: better than forward/CCBSE alone due to **smaller search spaces**
- ▶ Worst case: **doubling** the runtime of forward search (plus overhead)



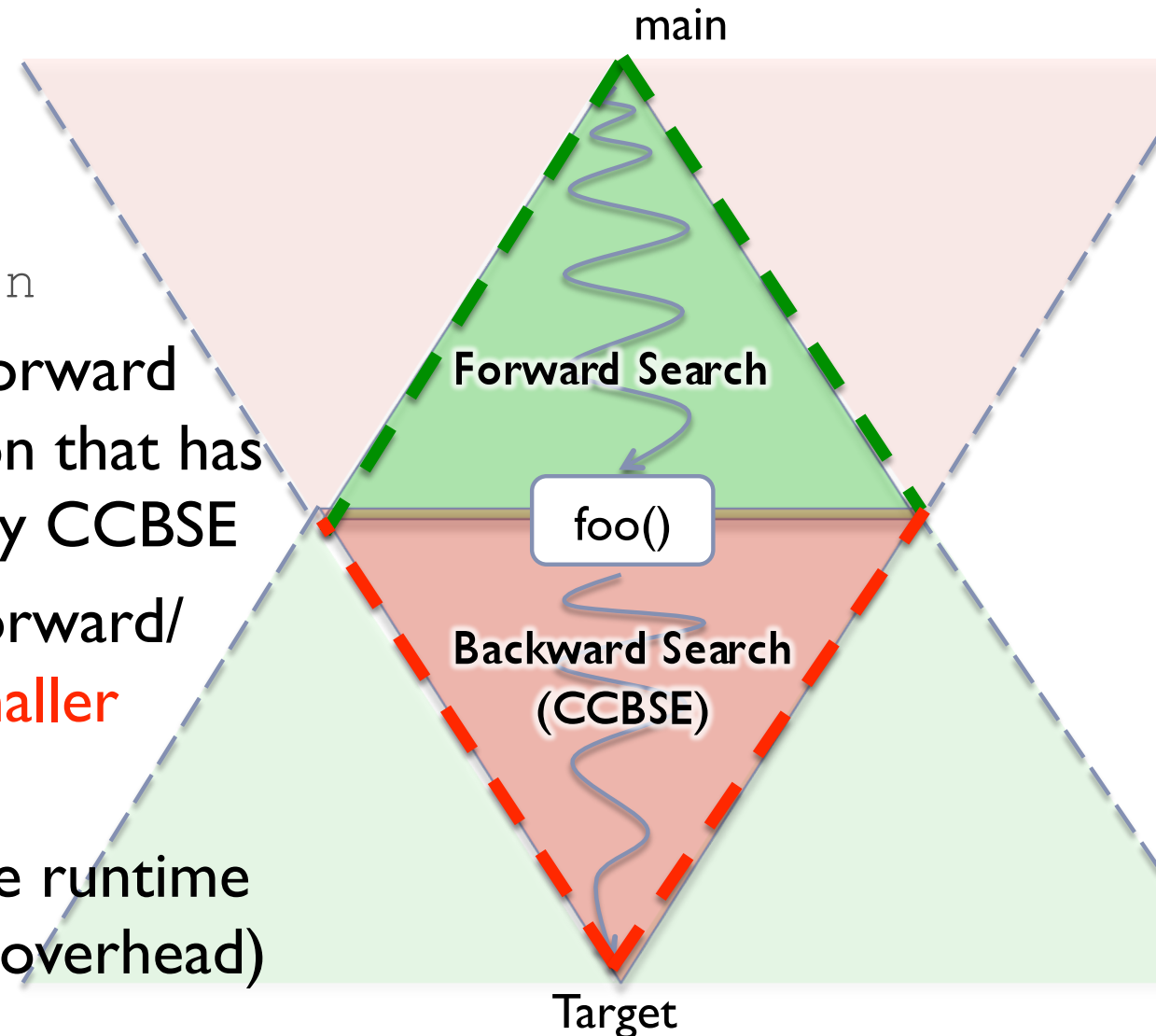
Mixing CCBSE with Forward Strategies

- ▶ **Simultaneously** run
 - ▶ CCBSE
 - ▶ Forward search from `main`
- ▶ **Try merging** when the forward search reaches a function that has a path to target found by CCBSE
- ▶ Best case: better than forward/CCBSE alone due to **smaller search spaces**
- ▶ Worst case: **doubling** the runtime of forward search (plus overhead)



Mixing CCBSE with Forward Strategies

- ▶ **Simultaneously** run
 - ▶ CCBSE
 - ▶ Forward search from `main`
- ▶ **Try merging** when the forward search reaches a function that has a path to target found by CCBSE
- ▶ Best case: better than forward/CCBSE alone due to **smaller search spaces**
- ▶ Worst case: **doubling** the runtime of forward search (plus overhead)



Evaluation

- ▶ **Otter**, a symbolic executor for C
- ▶ Compare different strategies
 - ▶ **InterSDSE**, **IntraSDSE**
 - ▶ **CCBSE** with *Random-Path*, *InterSDSE* and *IntraSDSE*
 - ▶ **Random-Path**, **Otter-KLEE** and **Otter-SAGE**
 - ▶ *With and without mixing with CCBSE(Random-Path)*
 - ▶ **KLEE**, as a reference
- ▶ Benchmark: 6 programs from **Coreutils 6.10**
 - ▶ Targets are bugs previously reported by KLEE
- ▶ Each test is run **41 times** using different seeds
 - ▶ To measure the effect of **randomness**

Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP) **RP:** random-path Key: Median SIQR(Outliers) ∞ : time out

	Inter-SDSE	Intra-SDSE	CCBSE(X) where X is			KLEE
			RP	InterSDSE	IntraSDSE	
mkdir	34.7 19.7(10)	∞	163.0 42.5	150.3 93.4	150.7 93.9	∞
mkfifo	13.1 0.4	∞	70.2 17.3	49.7 21.8	49.3 23.2(1)	274.2 315.6(9)
mknod	∞	∞	216.5 60.7	∞	∞	851.6 554.2(8)
paste	12.6 0.5	56.4 5.4	26.0 0.5(1)	31.0 4.8	32.1 4.0	30.6 9.7(8)
ptx	18.4 0.6(4)	103.5 19.7(1)	24.2 0.7(1)	24.5 0.9(3)	24.1 1.1(2)	93.8 81.7(7)
seq	12.1 0.4(1)	∞	30.9 1.4	369.3 425.9(6)	391.8 411.1(6)	38.2 14.5(8)
Total	1891.0	7360.0	530.9	2424.8	2448.0	3088.55

	Otter-KLEE		Otter-SAGE		Random Path	
	Pure	w/CCBSE	Pure	w/CCBSE	Pure	w/CCBSE
mkdir	168.9 31.0	124.7 12.1(2)	365.3 354.2(5)	1667.7 ∞	143.5 5.3	136.4 7.9
mkfifo	41.7 5.2(1)	38.2 4.6	77.6 101.1(2)	251.9 257.0(8)	59.4 3.7	52.7 1.8(1)
mknod	174.8 24.1	93.1 12.7	108.5 158.7(5)	236.4 215.0(5)	196.7 3.9(2)	148.9 11.8
paste	22.6 0.5(4)	28.6 0.9(3)	54.9 36.2(5)	60.4 52.1(3)	22.1 0.6	27.3 1.0(1)
ptx	33.2 3.9	27.1 2.7	∞	∞	28.9 0.8	28.1 1.1(2)
seq	354.8 94.3(1)	49.3 5.1(1)	∞	288.8 ∞	170.8 3.7(3)	35.9 1.4(1)
Total	795.8	360.9	4206.4	4305.3	621.3	429.4

Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP) **RP:** random-path Key: Median SIQR(Outliers) ∞ : time out

	Inter-SDSE		Intra-SDSE		CCBSE(X) where X is			KLEE	
					RP	InterSDSE	IntraSDSE		
mkdir	34.7	19.7(10)	∞		163.0	42.5	150.3	∞	
mkfifo	13.1	0.4	∞		70.2	17.3		15.6(9)	
mknod	∞		∞		216.5	60.7		4.2(8)	
paste	12.6	0.5	56.4	5.4	26.0	0.5(1)	31.8	9.7(8)	
ptx	6.4	0		(1)	24.2	0.7(1)	24.5	0.9(3)	93.8 81.7(7)
seq	2.1				30.9	1.4	369.2		38.2 14.5(8)
Total	18				530.9				3088.55

Semi-interquartile range (SIQR) (seconds)

Median (seconds)

of outliers

Time out (1800 seconds)

	Otter-KLEE		Otter-SAGE		Random Path	
	w/CCBSE	Pure	w/CCBSE	Pure	w/CCBSE	Pure
mkdir	124.7 12.1(2)	365.3 354.2(5)	1667.7 ∞	143.5 5.3	136.4 7.9	
mkfifo	38.2 4.6	77.6 101.1(2)	251.9 257.0(8)	59.4 3.7	52.7 1.8(1)	
mknod	93.1 12.7	108.5 158.7(5)	236.4	23.7 3.9(2)	148.9 11.8	
paste	28.6 0.9(3)				27.3 1.0(1)	
ptx	27.1 2.7	∞			28.1 1.1(2)	
seq	49.3 5.1(1)	∞			35.9 1.4(1)	
Total	795.8	360.9	4206.4		429.4	

Fastest two times per row are highlighted

Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP) **RP**: random-path Key: Median SIQR(Outliers) ∞ : time out

	Inter-SDSE		Intra-SDSE		CCBSE(X) where X is						KLEE	
					RP		InterSDSE		IntraSDSE			
mkdir	34.7	19.7(10)		∞	163.0	42.5	150.3	93.4	150.7	93.9		∞
mkfifo	13.1	0.4		∞	70.2	17.3	49.7	21.8	49.3	23.2(1)	274.2	315.6(9)
mknod	∞			∞	216.5	60.7	∞		∞		851.6	554.2(8)
paste	12.6	0.5	56.4	5.4	26.0	0.5(1)	31.0	4.8	32.1	4.0	30.6	9.7(8)
ptx			23.5	19.7(1)	24.2	0.7(1)	24.5	0.9(3)	24.1	1.1(2)	93.8	81.7(7)
seq					30.9	1.4	369.3	425.9(6)	391.8	411.1(6)	38.2	14.5(8)
Total					530.9		2424.8		2448.0		3088.55	

Huge SIQRs, and many outliers...

	CCBSE		Pure		w/CCBSE		Pure		w/CCBSE	
mkdir	166.5	61.0	124.7	12.1(2)	365.3	354.2(5)	1667.7	∞	143.5	5.3
mkfifo	41.7	5.2(1)	38.2	4.6	77.6	101.1(2)	251.9	257.0(8)	59.4	3.7
mknod	174.8	24.1	93.1	12.7	108.5	158.7(5)	236.4	215.0(5)	196.7	3.9(2)
paste	22.6	0.5(4)	28.6	0.9(3)	54.9	36.2(5)	60.4	52.1(3)	22.1	0.6
ptx	33.2	3.9	27.1	2.7	∞		∞		28.9	0.8
seq	354.8	94.3(1)	49.3	5.1(1)	∞		288.8	∞	170.8	3.7(3)
Total	795.8		360.9		4206.4		4305.3		621.3	

Huge SIQRs,
and many
outliers...

Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP) **RP**: random-path Key: Median SIQR(Outliers) ∞ : time out

	Inter-SDSE	Intra-SDSE	CCBSE(X) where X is			KLEE
			RP	InterSDSE	IntraSDSE	
mkdir	34.7 19.7(10)	∞	163.0 42.5	150.3 ∞		∞
mkfifo	13.1 0.4	∞	70.2 17.3			5.6(9)
mknod	∞	∞	216.0			
paste	12.6 0.5	56.4 5.4	26.0			
ptx	18.4 0.6(4)	103.5 19.7(1)	24.2			
seq	12.1 0.4(1)	∞	30.9			
Total	1891.0	7360.0	530.0			

Using inter-procedural distance is crucial for SDSE

	Otter-EE		Otter-SAGE			
	Pure	w/CCBSE	Pure	w/CCBSE	Pure	w/CCBSE
mkdir	1667.7 ∞		143.5 5.3	136.4 7.9		
mkfifo	251.9 257.0(8)		59.4 3.7	52.7 1.8(1)		
mknod	236.4 215.0(5)		196.7 3.9(2)	148.9 11.8		
paste	60.4 52.1(3)		22.1 0.6	27.3 1.0(1)		
ptx	∞		28.9 0.8	28.1 1.1(2)		
seq	354.8 316.0(4)	∞	288.8 ∞	170.8 3.7(3)	35.9 1.4(1)	
Total	795.8	360.9	4206.4	4305.3	621.3	429.4

Distance heuristic works very well in practice

Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP) **RP:** random-path **Key:** Median SIQR(Outliers) ∞ : time out

	Inter-SDSE	Intra-SDSE	CCBSE(X) where X is						KLEE
			RP		InterSDSE		IntraSDSE		
mkdir	34.7 19.7(10)	∞	163.0 42.5	150.3 93.4	150.7 93.9	∞			
mkfifo	13.1 0.4	∞	70.2 17.3	49.7 21.8	49.3 23.2(1)			274.2 315.6(9)	
mknod	∞	∞	216.5 60.7	∞	∞			851.6 554.2(8)	
paste	12.6 0.5	56.4 5.4	26.0 0.5(1)	31.0 4.8	32.1 4.0			30.6 9.7(8)	
ptx	18.4 0.6	103.5 19.7(1)	24.2 0.7(1)	24.5 0.9(3)	24.1 1.1(2)			93.8 81.7(7)	
seq	12.1 0.4	∞	30.9 1.4	369.3 425.9(6)	391.8 411.1(6)			38.2 14.5(8)	
Total	1891.1	7360.0	530.9	2424.8	2448.0			3088.55	

	SAGE		Random Path		w/CCBSE
	Pur	Pur	Pur	Pur	
mkdir	41.7 5.2(1)	58.2 4.6			7.9
mkfifo	174.8 24.1	93.1 12.7			
mknod	22.6 0.5(4)	28.6 0.9(3)			
paste	33.2 3.9	27.1 2.7			
ptx	354.8 94.3(1)	49.3 5.1(1)			1.4(1)
seq					
Total	795.8	360.9	4206.4		429.4

SDSE timed out

CCBSE(RP) did not time out on mknod, and it has better overall performance than InterSDSE

Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP) **RP**: random-path Key: Median SIQR(Outliers) ∞ : time out

	Inter-SDSE	Intra-SDSE	CCBSE(X) where X is			KLEE
			RP	InterSDSE	IntraSDSE	
mkdir	34.7 19.7(10)	∞	163.0 42.5	150.3 93.4	150.7 93.9	∞
mkfifo	13.1 0.4	∞	70.2 17.3	49.7 21.8	49.3 23.2(1)	274.2 315.6(9)
mknod	∞	∞	216.5 60.7	∞	∞	851.6 554.2(8)
paste	12.6 0.5	56.4 5.4	26.0 0.5(1)	31.0 4.8	32.1 4.0	30.6 9.7(8)
ptx	18.4 0.6(4)	103.5 19.7(1)	24.2 0.7(1)	24.5 0.9(3)	24.1 1.1(2)	93.8 81.7(7)
seq	12.1 0.4(1)	∞	30.9 1.4	369.3 425.9(6)	391.8 411.1(6)	38.2 14.5(8)
Total	1891.0	7360.0	530.9	2424.8	2448.0	3088.55

	Otter-KLEE		Otter-SAGE		Random Path	
	Pure	w/CCBSE	Pure	w/CCBSE	Pure	w/CCBSE
mkdir	168.9 31.0	124.7 12.1(2)	365.3 354.2(5)	1667.7 ∞	143.5 5.3	136.4 7.9
mkfifo	41.7 5.2(1)	38.2 4.6	77.6 101.1(9)	∞	∞	∞
mknod	174.8 24.1	93.1 12.7	174.8 24.1	93.1 12.7	174.8 24.1	93.1 12.7
paste	22.6 0.5(4)	28.6 0.9(3)	54.1 1.4(1)	28.6 0.9(3)	22.6 0.5(4)	28.6 0.9(3)
ptx	33.2 3.9	27.1 2.7	∞	∞	33.2 3.9	27.1 2.7
seq	354.8 94.3(1)	49.3 5.1(1)	∞	288.8 ∞	170.8 3.7(3)	35.9 1.4(1)
Total	795.8	360.9	4206.4	4305.3	621.3	429.4

Mixing CCBSE with Otter-KLEE
is good, even better than
CCBSE and Otter-KLEE alone

Results

Inter(Intra)-SDSE: inter(intra)-procedural shortest-distance symbolic execution
CCBSE(X): call-chain-backward symbolic execution, with X as the forward strategy
w/CCBSE: mix with CCBSE(RP) **RP:** random-path Key: Median SIQR(Outliers) ∞ : time out

	Inter-SDSE	Intra-SDSE	CCBSE(X) where X is			KLEE
			RP	InterSDSE	IntraSDSE	
mkdir	34.7 19.7(10)	∞	163.0 48			∞
mkfifo	13.1 0.4	∞				9 315.6(9)
mknod	∞	∞				1.2(8)
paste	12.6 0.5	56.4 5.4				1.7(8)
ptx	18.4 0.6(4)	103.5 19.7(1)				81.7(7)
seq	12.1 0.4(1)		50.9			38.2 14.5(8)
Total	1891.0	7360.0	530.9	2424.8	2448.0	3088.55

SDSE is fast in many cases, but can perform very poorly sometimes.

	Otter-KLEE		Otter-CCBSE		Path	
	Pure	w/CCBSE	Pure	CCBSE	Pure	CCBSE
mkdir	168.9 31.0	124.7 12.1(2)	5			
mkfifo	41.7 5.2(1)	38.2 4.6				
mknod	174.8 24.1	93.1 12.7	10			
paste	22.6 0.5(4)	28.6 0.9(3)	54.9		17.3 1.0(1)	
ptx	33.2 3.9	27.1 2.7		∞	28.9 0.8	28.1 1.1(2)
seq	354.8 94.3(1)	49.3 5.1(1)	∞	288.8 ∞	170.8 3.7(3)	35.9 1.4(1)
Total	795.8	360.9	4206.4	4305.3	621.3	429.4

Mixing CCBSE with Otter-KLEE gives the best overall performance.

Limitations and Future Work

▶ Threats to validity

- ▶ Our implementation of the **KLEE** and **SAGE** strategies in Otter unavoidably differs from the originals
- ▶ Different front-end/parser and standard libraries from **original KLEE**

▶ Future work

- ▶ Compare with **ESD**
 - ▶ Another work on directed symbolic execution (not publicly available)
- ▶ Expanding our benchmark suite
- ▶ **Multiple targets** in one program
 - ▶ Could we do better than running directed symbolic execution on each target alone?

Conclusions

- ▶ Directed symbolic execution
 - ▶ Guide the search towards a target line
- ▶ Our contributions:
 - ▶ Shortest-Distance Symbolic Execution (SDSE)
 - ▶ Call-Chain-Backward Symbolic Execution (CCBSE)
 - ▶ Mixing CCBSE with a forward strategy (Mix-CCBSE)
- ▶ Results show that
 - ▶ SDSE is fast in many cases, but can perform poorly sometimes
 - ▶ Mixing CCBSE with KLEE has the best overall performance