Lecture 11 – Dynamic Analysis

SWS128: Software Analysis

Jihyeok Park



2026 Fall

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Symbolic execution answered with **inputs**:

f returns	0	1	2
input	$(10, 0)$	$(2, 9)$	none

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Symbolic execution answered with **inputs**:

f returns	0	1	2
input	$(10, 0)$	$(2, 9)$	none

And we can check them: run `f(2, 9)`, get `1`.

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Symbolic execution answered with **inputs**:

f returns	0	1	2
input	$(10, 0)$	$(2, 9)$	none

And we can check them: run `f(2, 9)`, get `1`.

- Every analysis so far reasoned about the program **without running it**.

Recall

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

Symbolic execution answered with **inputs**:

f returns	0	1	2
input	$(10, 0)$	$(2, 9)$	none

And we can check them: run `f(2, 9)`, get `1`.

- Every analysis so far reasoned about the program **without running it**.
- Today: **run** it, and learn from what it does. **Dynamic analysis**.

Contents

1. Watching a Run

2. What a Trace Tells
   - Coverage
   - Taint
   - More Analyses

3. Generating Inputs

4. Concolic Execution
   - Flip and Solve
   - Hybrid Fuzzing

5. In Practice

Contents

1. Watching a Run

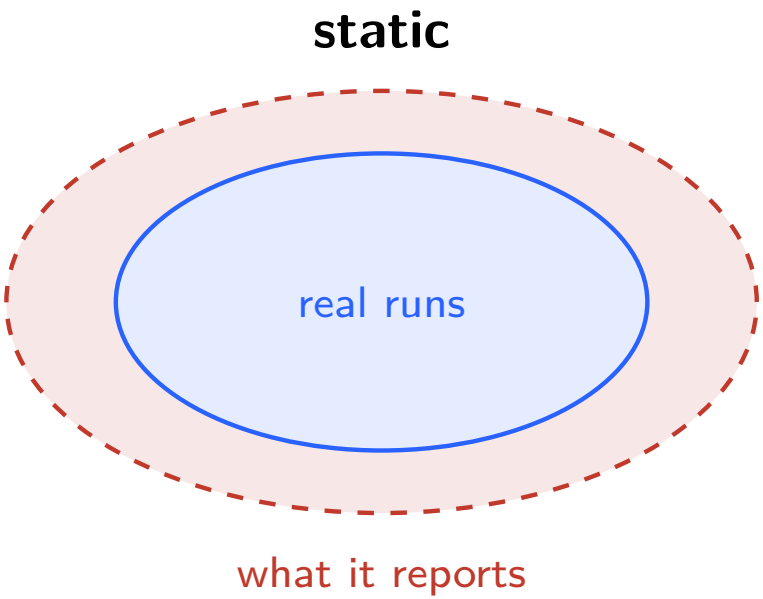
2. What a Trace Tells
   - Coverage
   - Taint
   - More Analyses

3. Generating Inputs

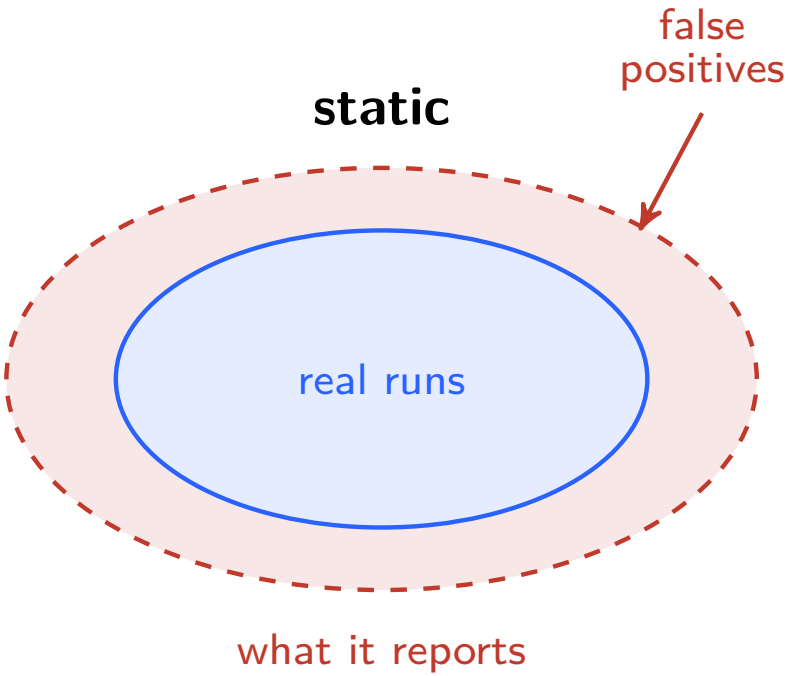
4. Concolic Execution
   - Flip and Solve
   - Hybrid Fuzzing

5. In Practice

All Runs, or Some Runs

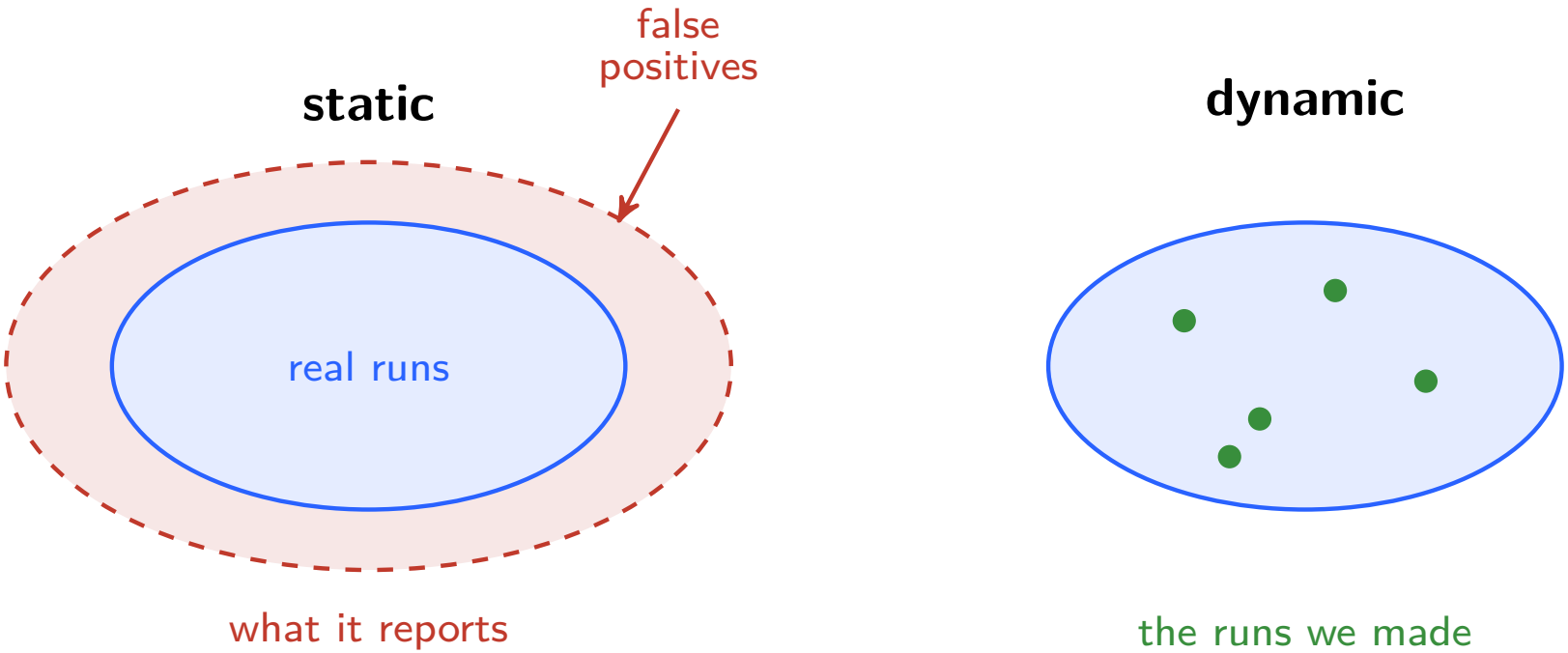


All Runs, or Some Runs



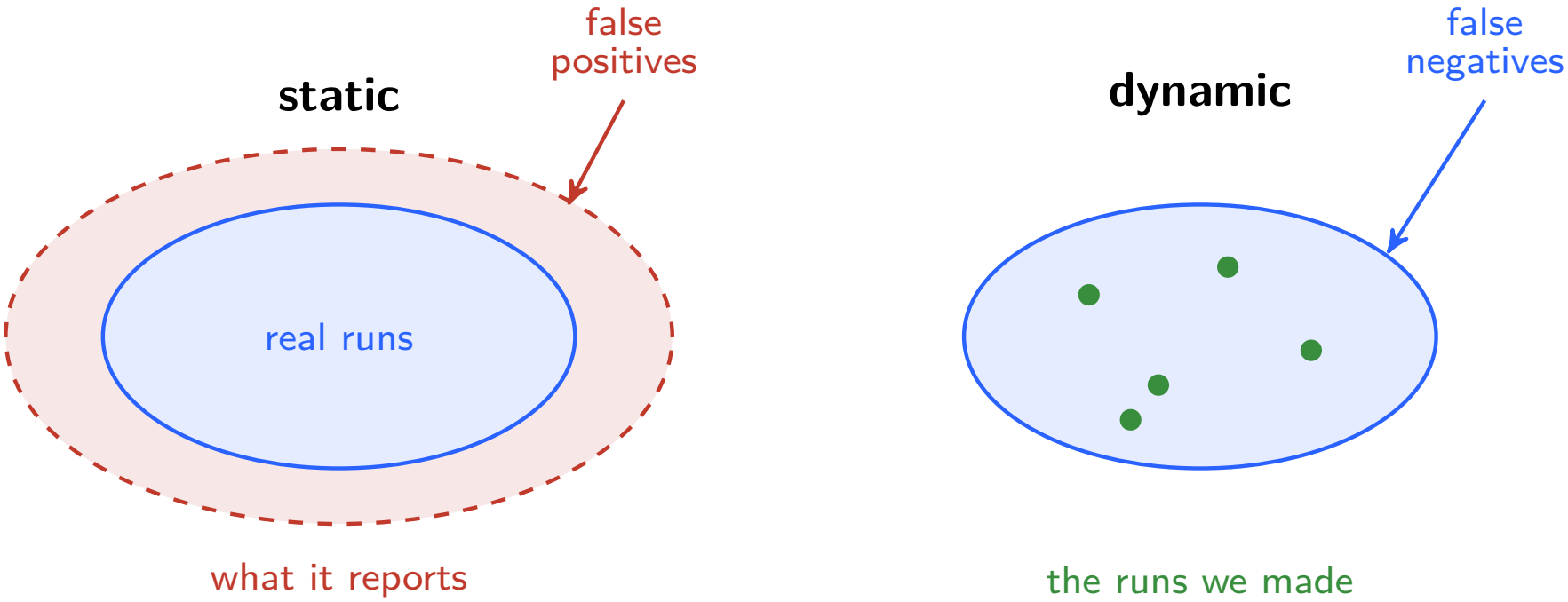
- **Static**: looks at every run, so some reports are about runs that never happen.

All Runs, or Some Runs



- **Static**: looks at every run, so some reports are about runs that never happen.

All Runs, or Some Runs



- **Static**: looks at every run, so some reports are about runs that never happen.
- **Dynamic**: every report is real, but runs we did not make are **missed**.

A Trace

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

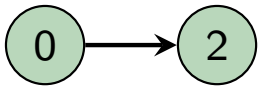
point	ran	state
0	f(2, 9)	$x = 2, y = 9$

trace: 0

A Trace

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

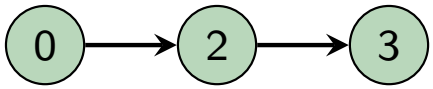
point	ran	state
0	f(2, 9)	$x = 2, y = 9$
2	r = 0	$r = 0$

trace: 

A Trace

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

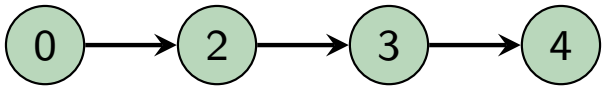
point	ran	state
0	`f(2, 9)`	$\texttt{x} = 2, \texttt{y} = 9$
2	`r = 0`	$\texttt{r} = 0$
3	`z = x + y`	$\texttt{z} = 11$

trace: 

A Trace

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

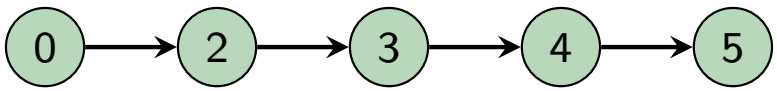
point	ran	state
0	`f(2, 9)`	$\texttt{x} = 2, \texttt{y} = 9$
2	`r = 0`	$\texttt{r} = 0$
3	`z = x + y`	$\texttt{z} = 11$
4	`10 < z`	`True`

trace: 

A Trace

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

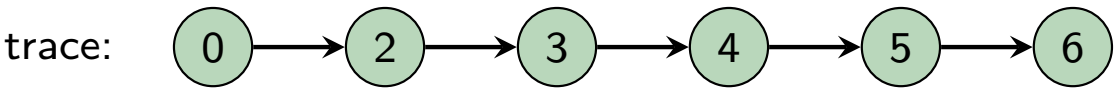
point	ran	state
0	`f(2, 9)`	$x = 2, y = 9$
2	`r = 0`	$r = 0$
3	`z = x + y`	$z = 11$
4	`10 < z`	`True`
5	`x < 3`	`True`

trace: 

A Trace

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

point	ran	state
0	`f(2, 9)`	$x = 2, y = 9$
2	`r = 0`	$r = 0$
3	`z = x + y`	$z = 11$
4	`10 < z`	`True`
5	`x < 3`	`True`
6	`r = 1`	$r = 1$



A Trace

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

point	ran	state
0	f(2, 9)	x = 2, y = 9
2	r = 0	r = 0
3	z = x + y	z = 11
4	10 < z	True
5	x < 3	True
6	r = 1	r = 1
1	y < 8	False

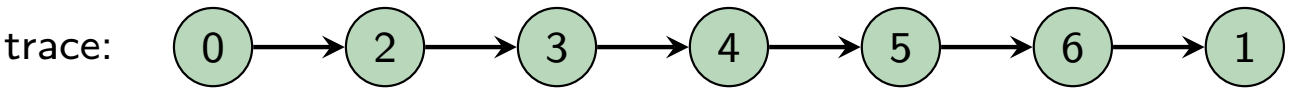
trace:



A Trace

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

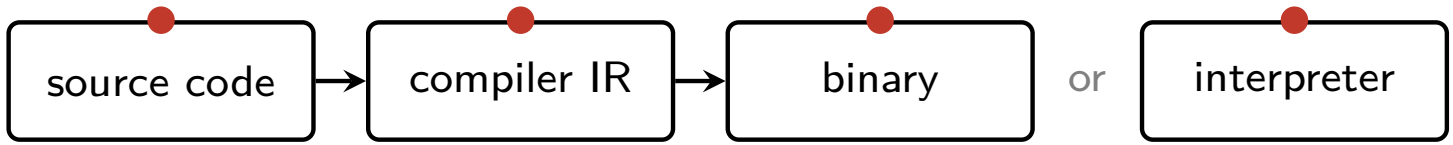
point	ran	state
0	`f(2, 9)`	$x = 2, y = 9$
2	`r = 0`	$r = 0$
3	`z = x + y`	$z = 11$
4	`10 < z`	`True`
5	`x < 3`	`True`
6	`r = 1`	$r = 1$
1	`y < 8`	`False`
	`return r`	returns 1



Record each point as the program passes it. Everything today is read off **traces** like this one.

Where the Probes Go

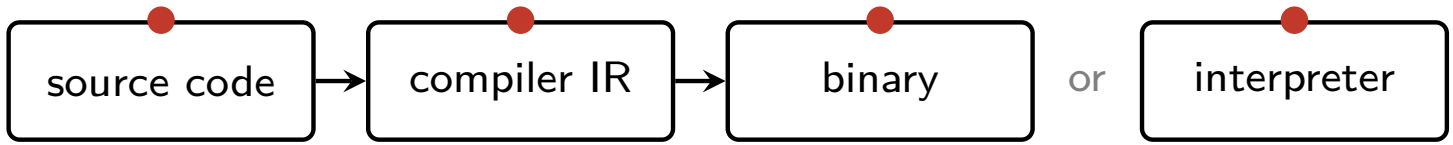
To record a trace, add **probes** to the program: **instrumentation**.



rewrite the program	a compiler pass	rewrite the machine code	a hook in the interpreter
Istanbul (JS)	AFL++, ASan	Valgrind, Pin	coverage.py

Where the Probes Go

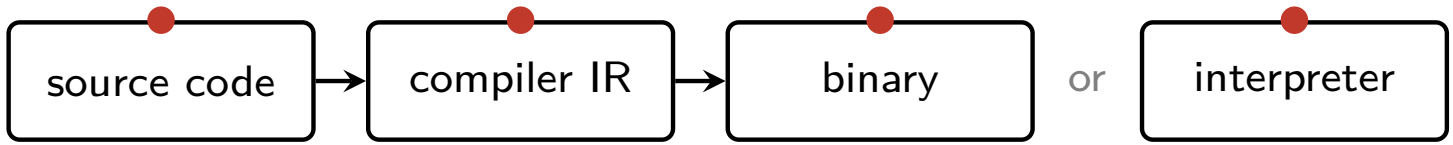
To record a trace, add **probes** to the program: **instrumentation**.



	rewrite the program	a compiler pass	rewrite the machine code	a hook in the interpreter
	Istanbul (JS)	AFL++, ASan	Valgrind, Pin	coverage.py
source?	needed	needed	not needed	not needed
added	at compile time	at compile time	at run time	at run time

Where the Probes Go

To record a trace, add **probes** to the program: **instrumentation**.

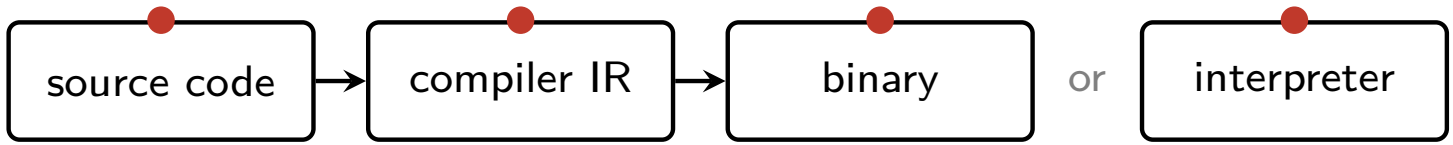


	rewrite the program Istanbul (JS)	a compiler pass AFL++, ASan	rewrite the machine code Valgrind, Pin	a hook in the interpreter coverage.py
source?	needed	needed	not needed	not needed
added	at compile time	at compile time	at run time	at run time

- The cost is in the probes, not the level: put them only where the analysis needs them.

Where the Probes Go

To record a trace, add **probes** to the program: **instrumentation**.



	rewrite the program Istanbul (JS)	a compiler pass AFL++, ASan	rewrite the machine code Valgrind, Pin	a hook in the interpreter coverage.py
source?	needed	needed	not needed	not needed
added	at compile time	at compile time	at run time	at run time

- The cost is in the probes, not the level: put them only where the analysis needs them.
- Coverage: one probe per branch, and only its first hit matters. Python's `sys.monitoring` can switch a hook off once it has fired.

Contents

1. Watching a Run

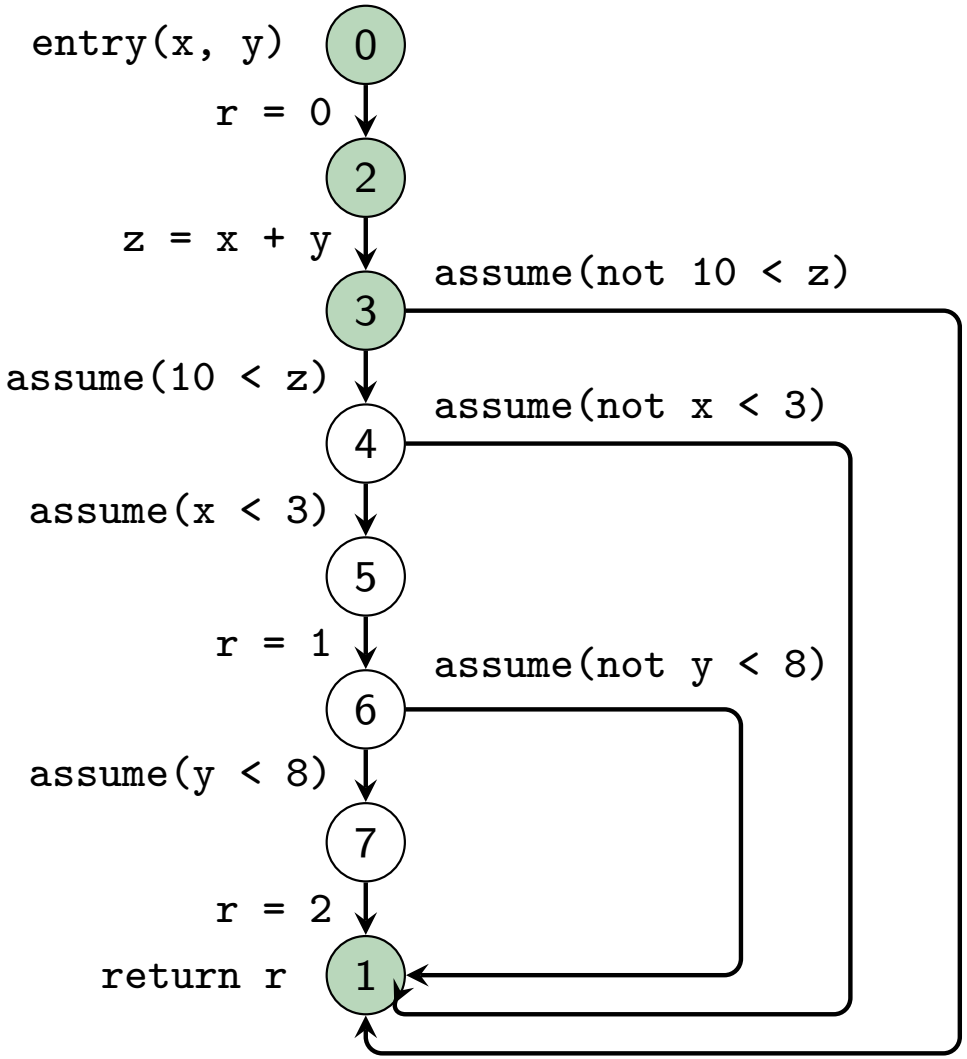
2. What a Trace Tells
   - Coverage
   - Taint
   - More Analyses

3. Generating Inputs

4. Concolic Execution
   - Flip and Solve
   - Hybrid Fuzzing

5. In Practice

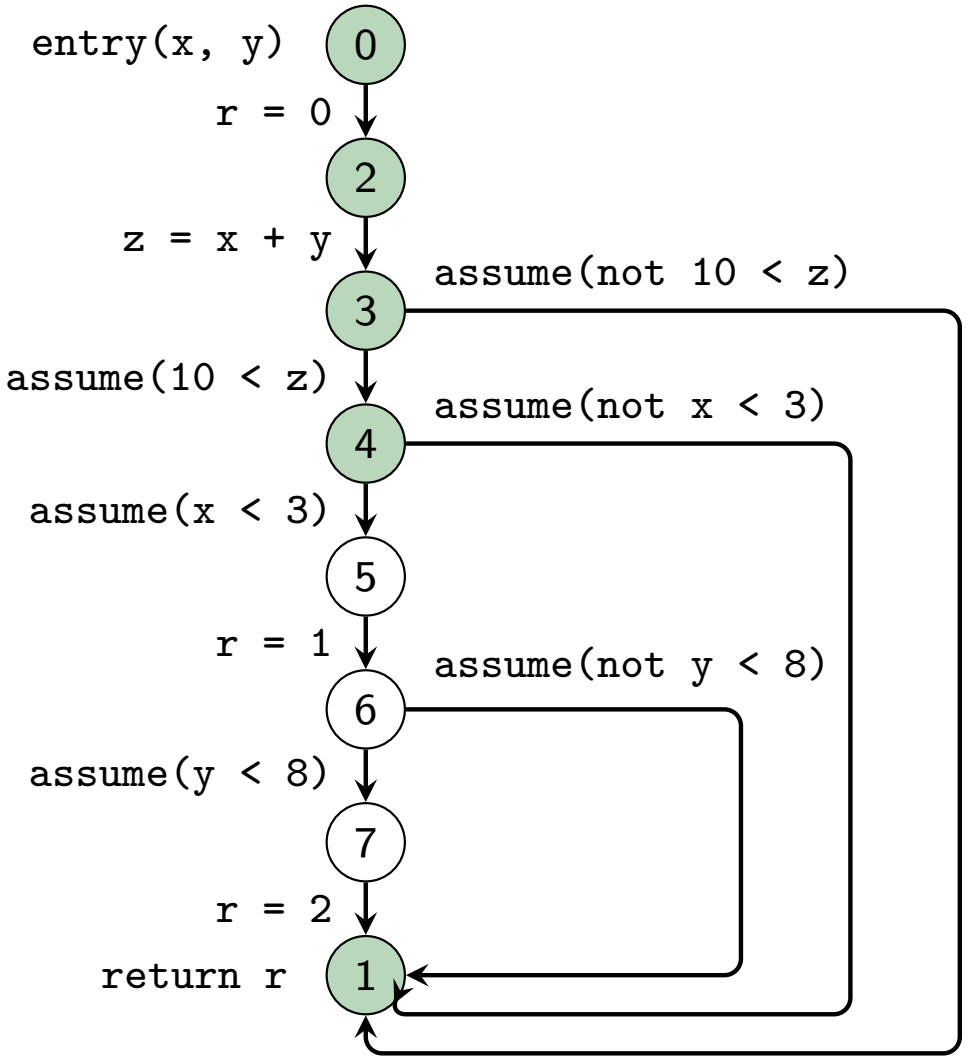
Coverage



Which points and branches ran?

inputs run	points	branches
$(0, 0)$	$4/8$	$1/6$

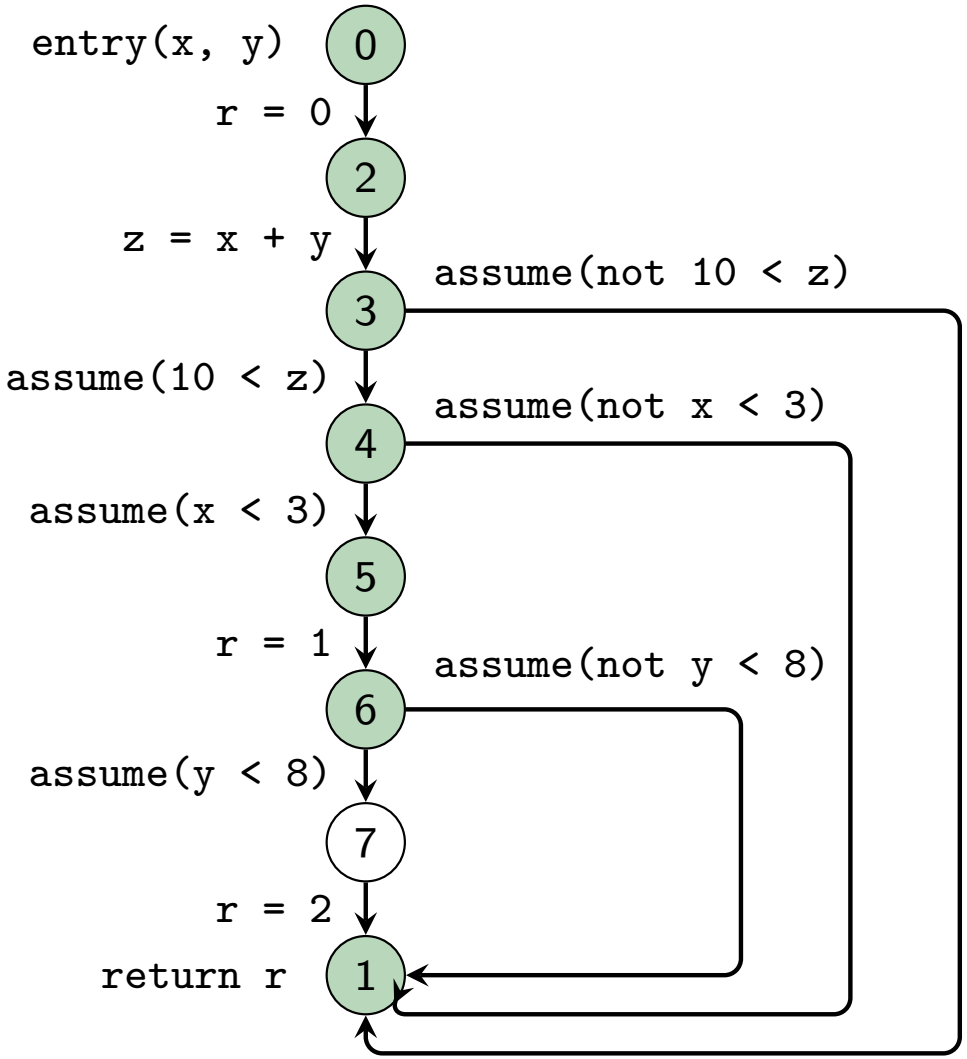
Coverage



Which points and branches ran?

inputs run	points	branches
$(0, 0)$	$4/8$	$1/6$
$+ (100, 8)$	$5/8$	$3/6$

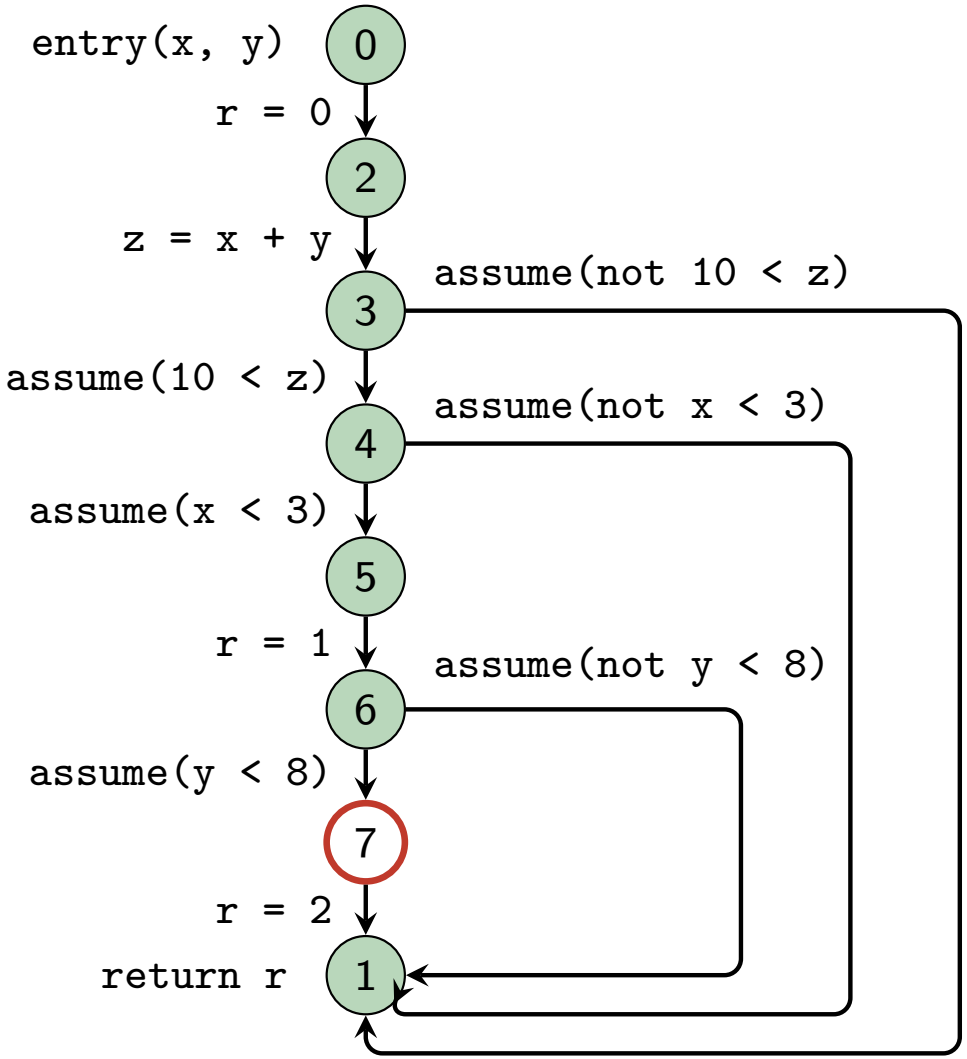
Coverage



Which points and branches ran?

inputs run	points	branches
$(0, 0)$	$4/8$	$1/6$
$+ (100, 8)$	$5/8$	$3/6$
$+ (-1, 100)$	$7/8$	$5/6$

Coverage

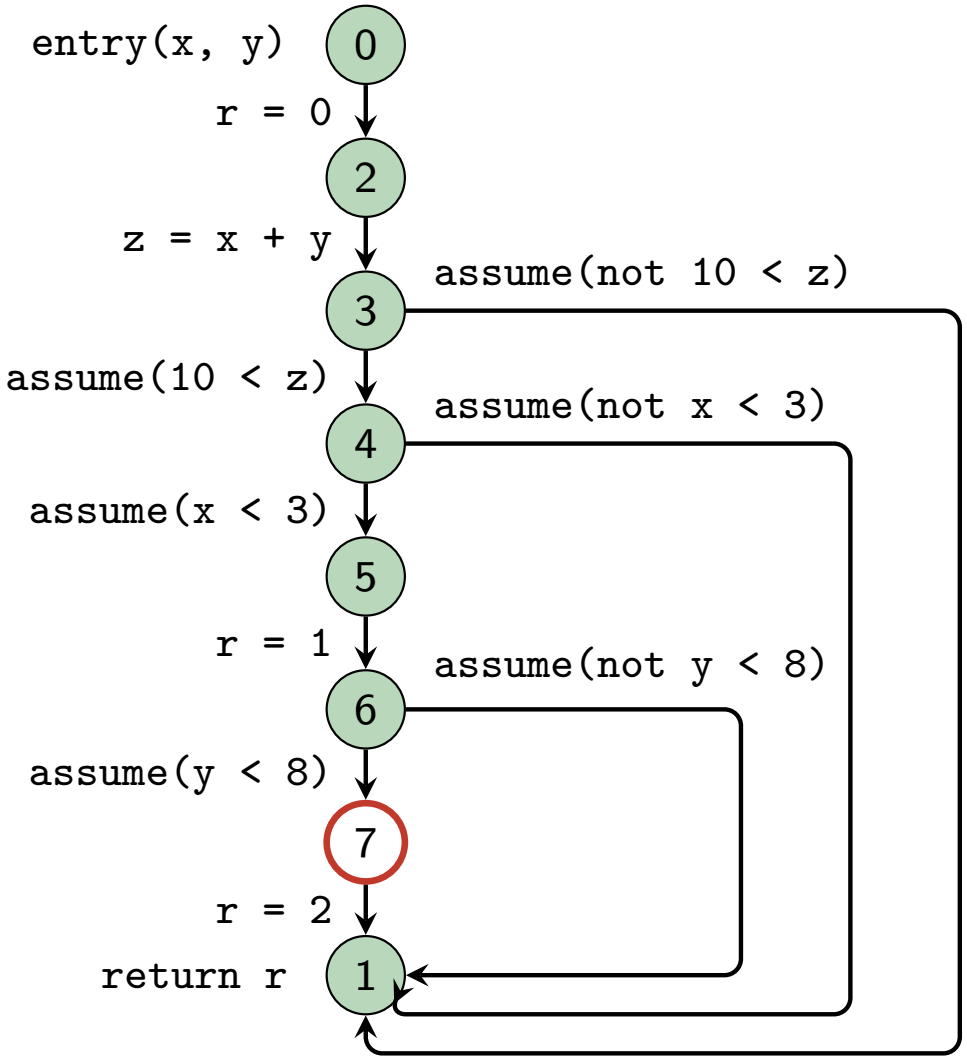


Which points and branches ran?

inputs run	points	branches
$(0, 0)$	$4/8$	$1/6$
$+ (100, 8)$	$5/8$	$3/6$
$+ (-1, 100)$	$7/8$	$5/6$

- Point 7 never ran. **Untested**, or **dead**? Coverage cannot tell.

Coverage



Which points and branches ran?

inputs run	points	branches
$(0, 0)$	$4/8$	$1/6$
$+ (100, 8)$	$5/8$	$3/6$
$+ (-1, 100)$	$7/8$	$5/6$

- Point 7 never ran. **Untested**, or **dead**? Coverage cannot tell.
- Last time, symbolic execution proved it dead.

Taint at Runtime

```
def query(uid, safe):
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp
    n = 0
    return sql
```

Every value carries a **label**.
Red: it came from a parameter.

Taint at Runtime

```
def query(uid, safe):
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp
    n = 0
    return sql
```

Every value carries a **label**.
Red: it came from a parameter.

- Red in, red out: a result is red if an operand is.

	query("1 OR 1=1", True)	query("1 OR 1=1", False)
uid	"1 OR 1=1"	"1 OR 1=1"
tmp	"1 OR 1=1"	"1 OR 1=1"
sql	"SELECT ... id = 1"	"SELECT ... id = 1 OR 1=1"

Taint at Runtime

```
def query(uid, safe):
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp
    n = 0
    return sql
```

Every value carries a **label**.
Red: it came from a parameter.

- Red in, red out: a result is red if an operand is.
- Red at `return`: a **leak**.

	query("1 OR 1=1", True)	query("1 OR 1=1", False)
uid	"1 OR 1=1"	"1 OR 1=1"
tmp	"1 OR 1=1"	"1 OR 1=1"
sql	"SELECT ... id = 1"	"SELECT ... id = 1 OR 1=1"
return	clean	**leak**

Static versus Dynamic Taint

```
def query(uid, safe):
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp
    n = 0
    return sql
```

tests run	static	dynamic
`safe = True`	**leak**	clean

Static versus Dynamic Taint

```
def query(uid, safe):
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp
    n = 0
    return sql
```

tests run	static	dynamic
`safe = True`	**leak**	clean
`safe = False`	**leak**	**leak**

Static versus Dynamic Taint

```
def query(uid, safe):
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp
    n = 0
    return sql
```

tests run	static	dynamic
`safe = True`	**leak**	clean
`safe = False`	**leak**	**leak**

- **Static**: *leak*, whatever we test. It is about the program, not a run.

Static versus Dynamic Taint

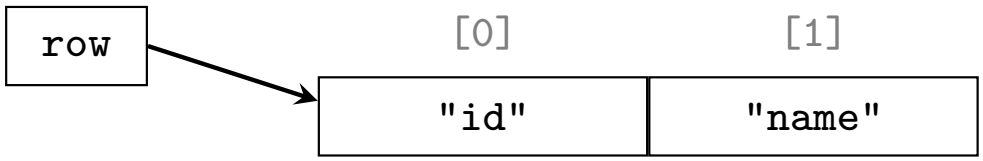
```
def query(uid, safe):
    sql = "SELECT * FROM users WHERE id = "
    tmp = uid
    if safe:
        sql = sql + "1"
    else:
        sql = sql + tmp
    n = 0
    return sql
```

tests run	static	dynamic
`safe = True`	leak	clean
`safe = False`	leak	leak

- **Static**: *leak*, whatever we test. It is about the program, not a run.
- **Dynamic**: only what ran. Test only `safe = True`, and it misses the leak: a **false negative**.

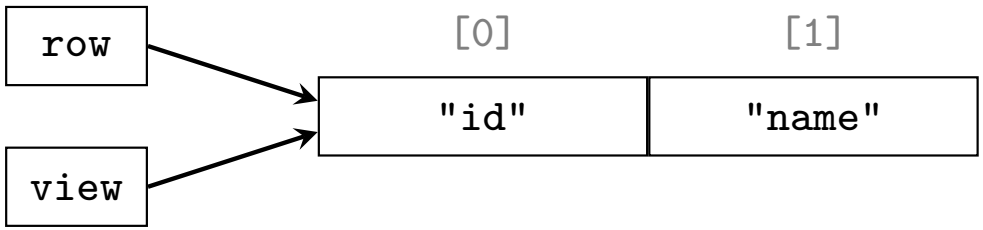
Aliasing, for Free

```
def alias(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```



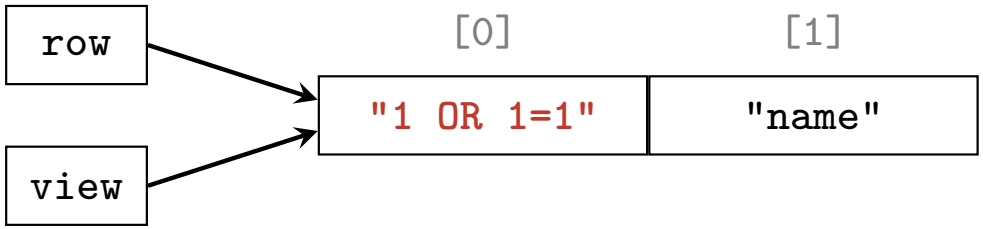
Aliasing, for Free

```
def alias(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```



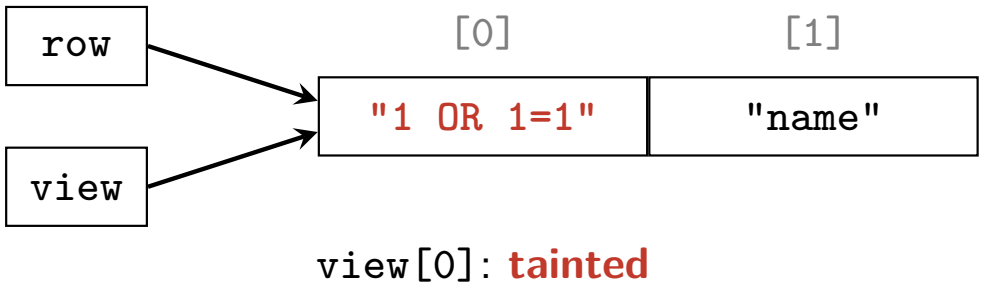
Aliasing, for Free

```
def alias(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```



Aliasing, for Free

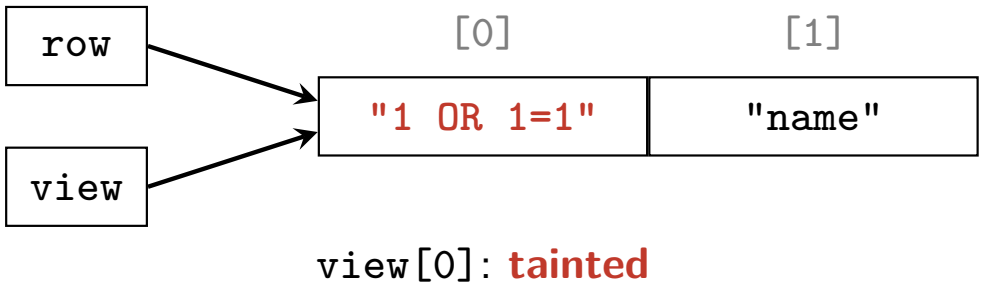
```
def alias(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```



- Statically, we needed a **points-to analysis** to know that `row` and `view` are one list.

Aliasing, for Free

```
def alias(uid):
    row = ["id", "name"]
    view = row
    row[0] = uid
    return view[0]
```

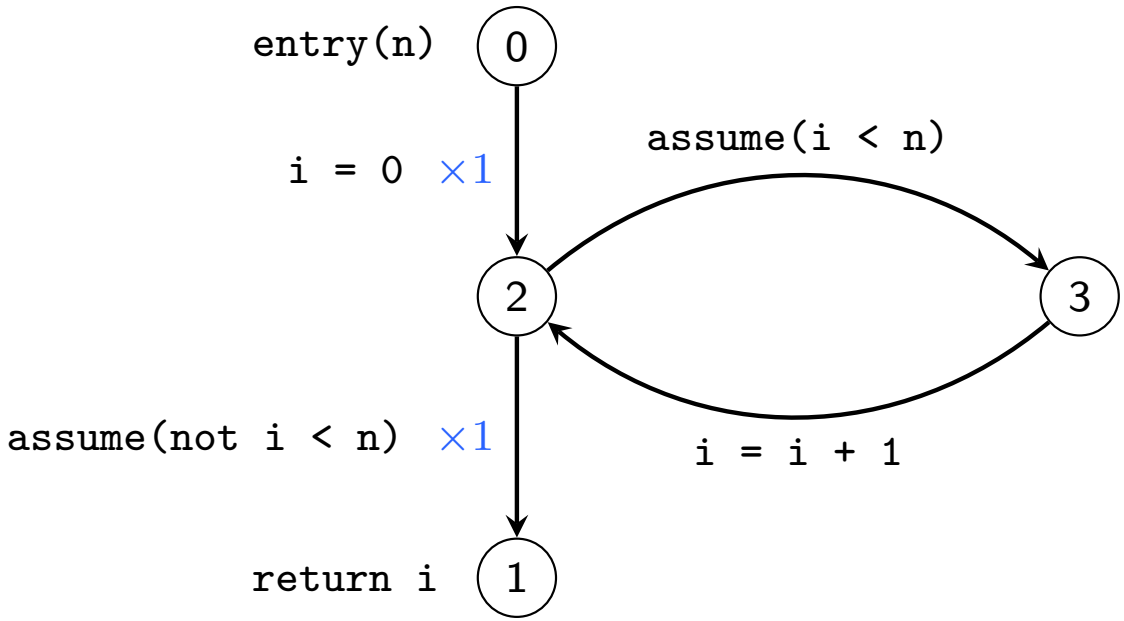


- Statically, we needed a **points-to analysis** to know that `row` and `view` are one list.
- At run time there **is** one list, and the label is in its cell.

Profiling

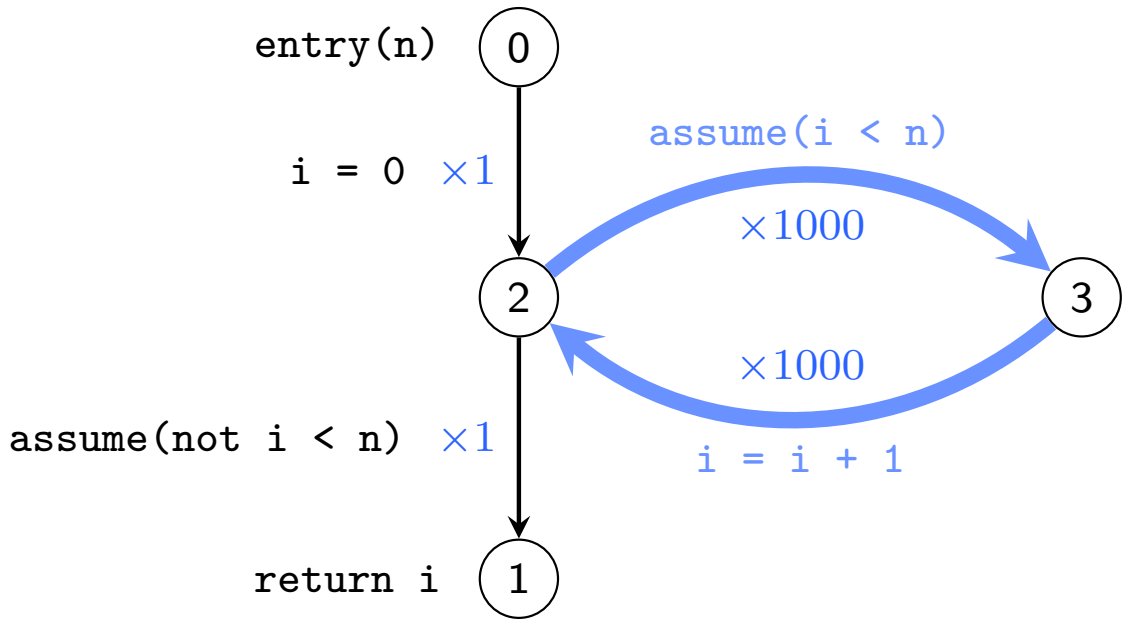
```
def loop(n):
    i = 0
    while i < n:
        i = i + 1
    return i
```

Run `loop(1000)`, and **count** how many times each edge runs.



Profiling

```
def loop(n):
    i = 0
    while i < n:
        i = i + 1
    return i
```

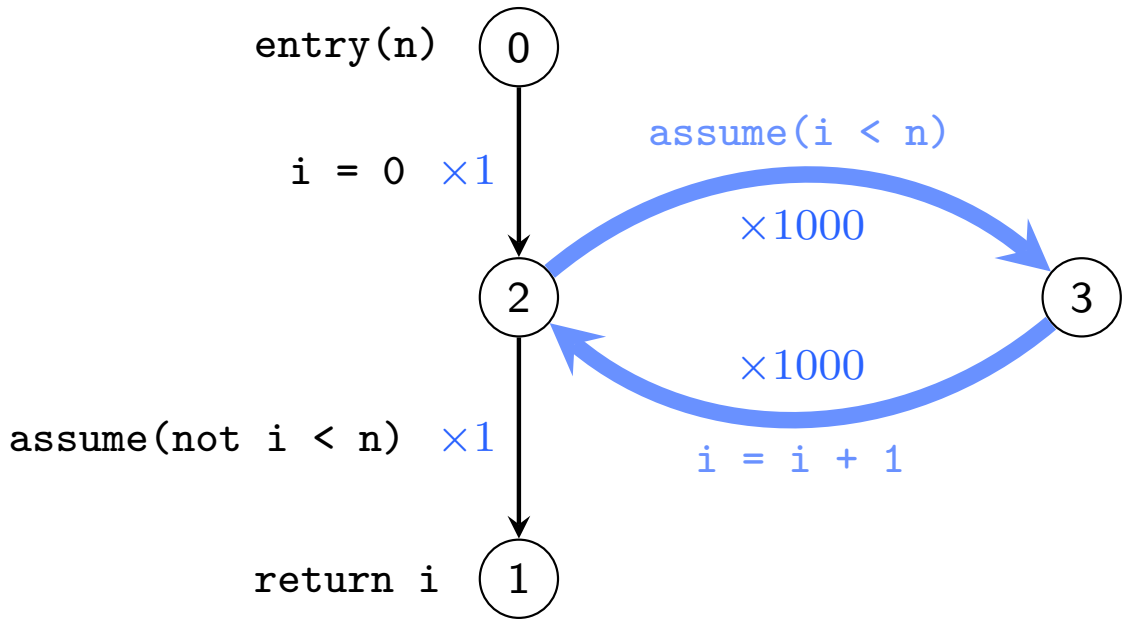


Run `loop(1000)`, and **count** how many times each edge runs.

- The loop runs 1000 times, the rest once. The time goes into the loop, so speed up the loop.

Profiling

```
def loop(n):
    i = 0
    while i < n:
        i = i + 1
    return i
```



Run `loop(1000)`, and **count** how many times each edge runs.

- The loop runs 1000 times, the rest once. The time goes into the loop, so speed up the loop.
- This is what a **profiler** shows you: `perf`, Python's `cProfile`.

Likely Invariants

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

runs that reach `r = 1`	
x	y
-1	100
0	11
1	15
2	9

At `r = 1`, what was true in **every** run?

fact	every run	always
$x < 3$	yes	
$9 \leq y$	yes	
$-1 \leq x$	yes	

Likely Invariants

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

runs that reach `r = 1`	
x	y
-1	100
0	11
1	15
2	9

At `r = 1`, what was true in **every** run?

fact	every run	always
$x < 3$	yes	yes
$9 \leq y$	yes	yes
$-1 \leq x$	yes	

- True in every run so far: a **likely invariant**. Daikon finds these.

Likely Invariants

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

runs that reach `r = 1`	
x	y
-1	100
0	11
1	15
2	9

At `r = 1`, what was true in **every** run?

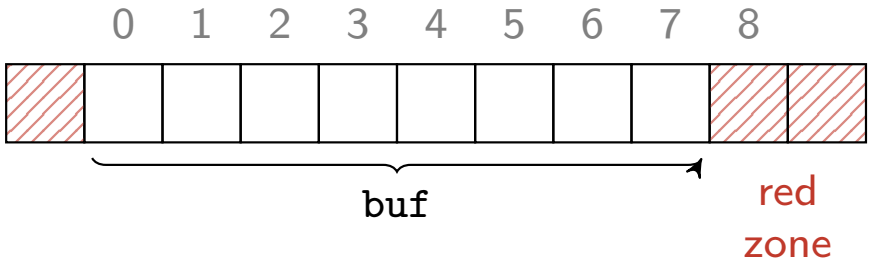
fact	every run	always
$x < 3$	yes	yes
$9 \leq y$	yes	yes
$-1 \leq x$	yes	no: $(-5, 20)$

- True in every run so far: a **likely invariant**. Daikon finds these.
- $-1 \leq x$ only says that no test tried a smaller x.

Memory Errors

```
char buf[8];
for (int i = 0; i <= 8; i++)
  buf[i] = 'A';
```

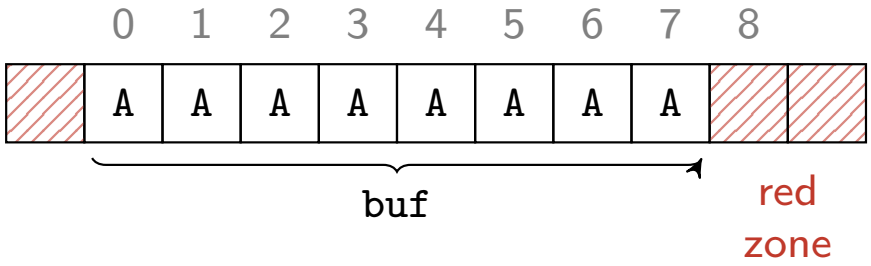
C: `buf[8]` is past the end.



Memory Errors

```
char buf[8];
for (int i = 0; i <= 8; i++)
  buf[i] = 'A';
```

C: `buf[8]` is past the end.

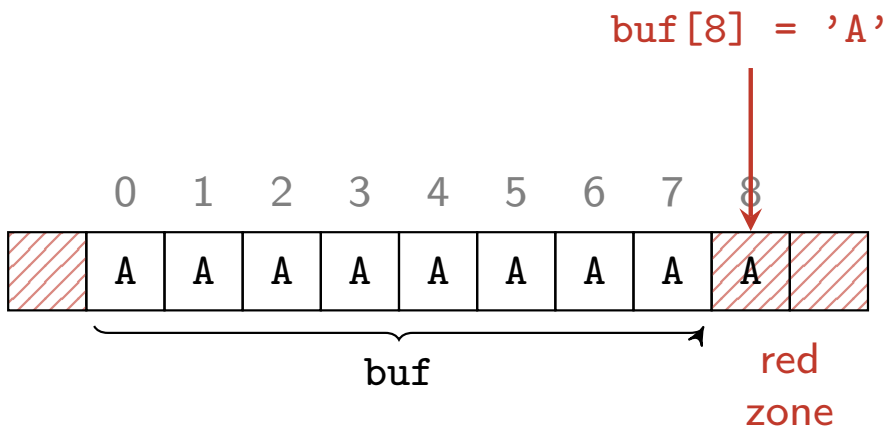


- A **sanitizer** puts a **red zone** around `buf`. Touching it stops the program:

Memory Errors

```
char buf[8];
for (int i = 0; i <= 8; i++)
  buf[i] = 'A';
```

C: `buf[8]` is past the end.



- A **sanitizer** puts a **red zone** around `buf`. Touching it stops the program:

```
ERROR: AddressSanitizer: stack-buffer-overflow ...
```

One Trace, Many Analyses

analysis	**records at each step**	**reports**
coverage	the point	points never reached
taint	a label on each value	a label at the sink
profiling	a count per edge	where the time goes
invariants	the values of variables	facts true in every run
sanitizer	which bytes are red zone	a touch of a red zone

One Trace, Many Analyses

analysis	**records at each step**	**reports**
coverage	the point	points never reached
taint	a label on each value	a label at the sink
profiling	a count per edge	where the time goes
invariants	the values of variables	facts true in every run
sanitizer	which bytes are red zone	a touch of a red zone

- Same probes, different records.

One Trace, Many Analyses

analysis	**records at each step**	**reports**
coverage	the point	points never reached
taint	a label on each value	a label at the sink
profiling	a count per edge	where the time goes
invariants	the values of variables	facts true in every run
sanitizer	which bytes are red zone	a touch of a red zone

- Same probes, different records.
- All of them see **only the runs we made**. So which runs?

Contents

1. Watching a Run

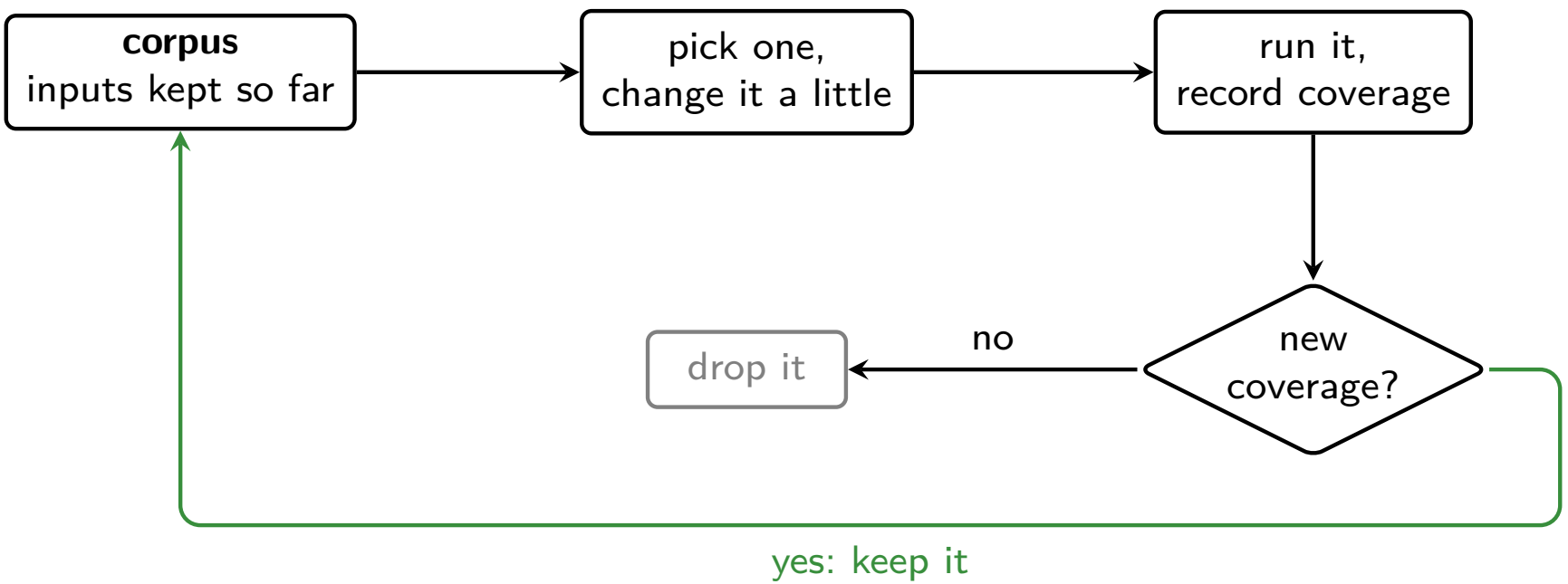
2. What a Trace Tells
   - Coverage
   - Taint
   - More Analyses

3. Generating Inputs

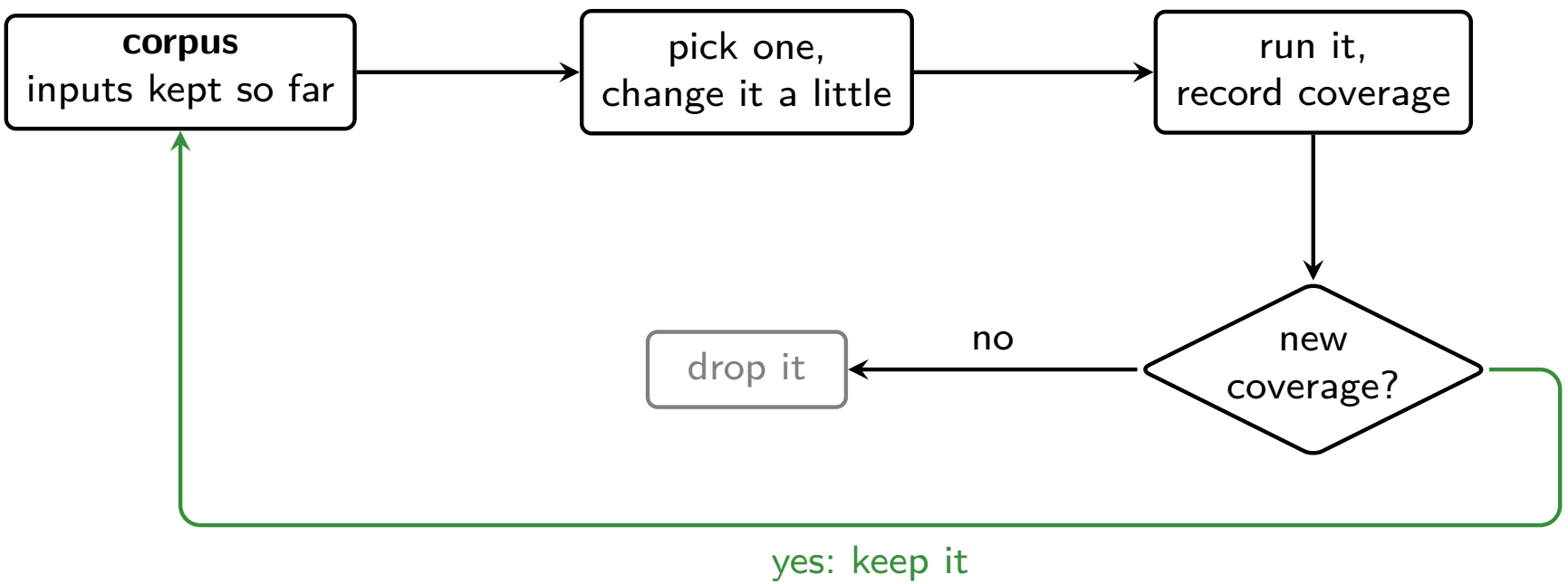
4. Concolic Execution
   - Flip and Solve
   - Hybrid Fuzzing

5. In Practice

Coverage-Guided Fuzzing

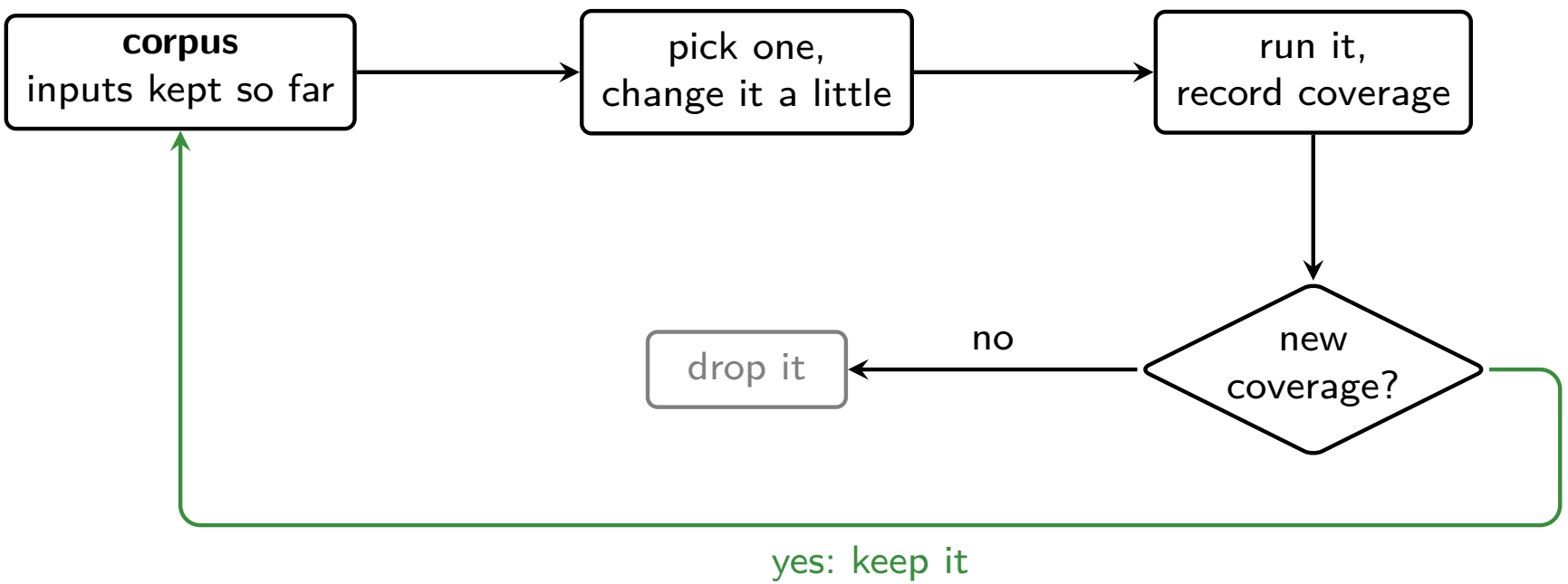


Coverage-Guided Fuzzing



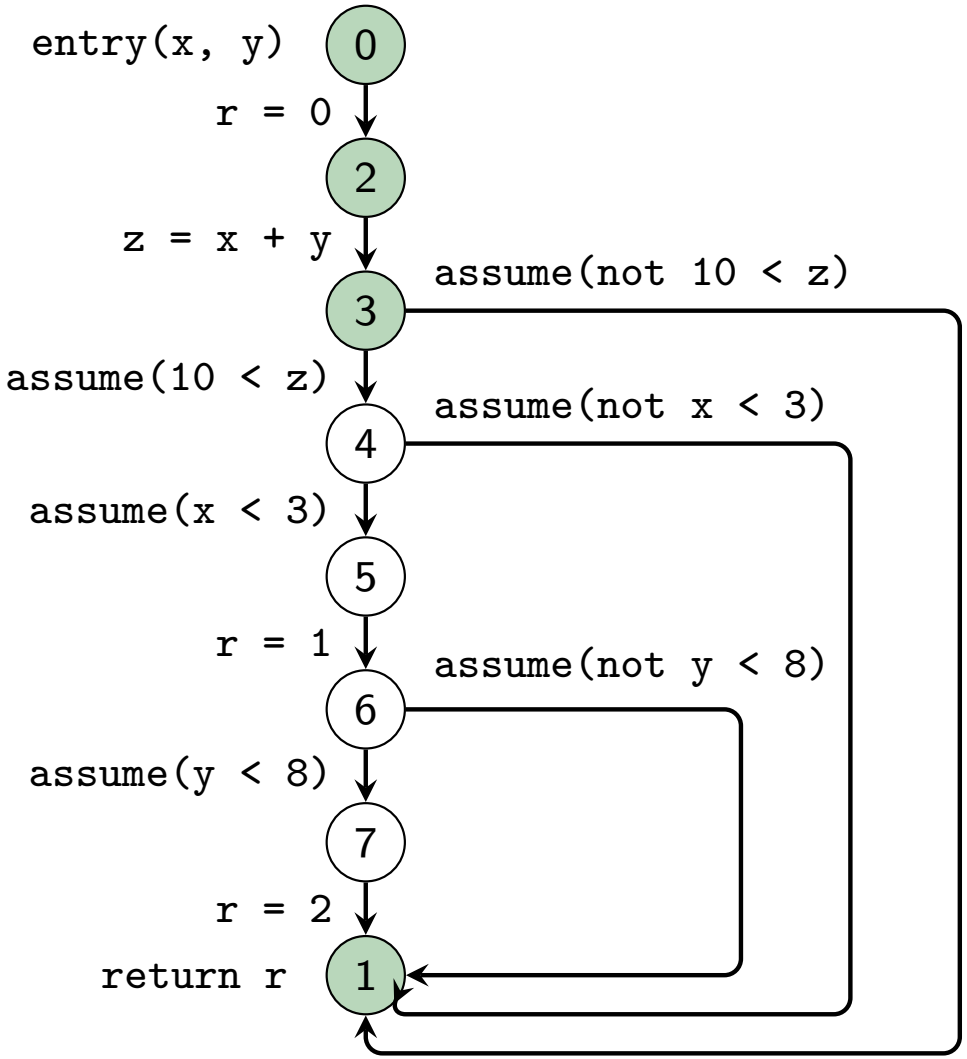
- Keep an input only if it reached something **new**.

Coverage-Guided Fuzzing



- Keep an input only if it reached something **new**.
- AFL++ and libFuzzer: this loop, made fast.

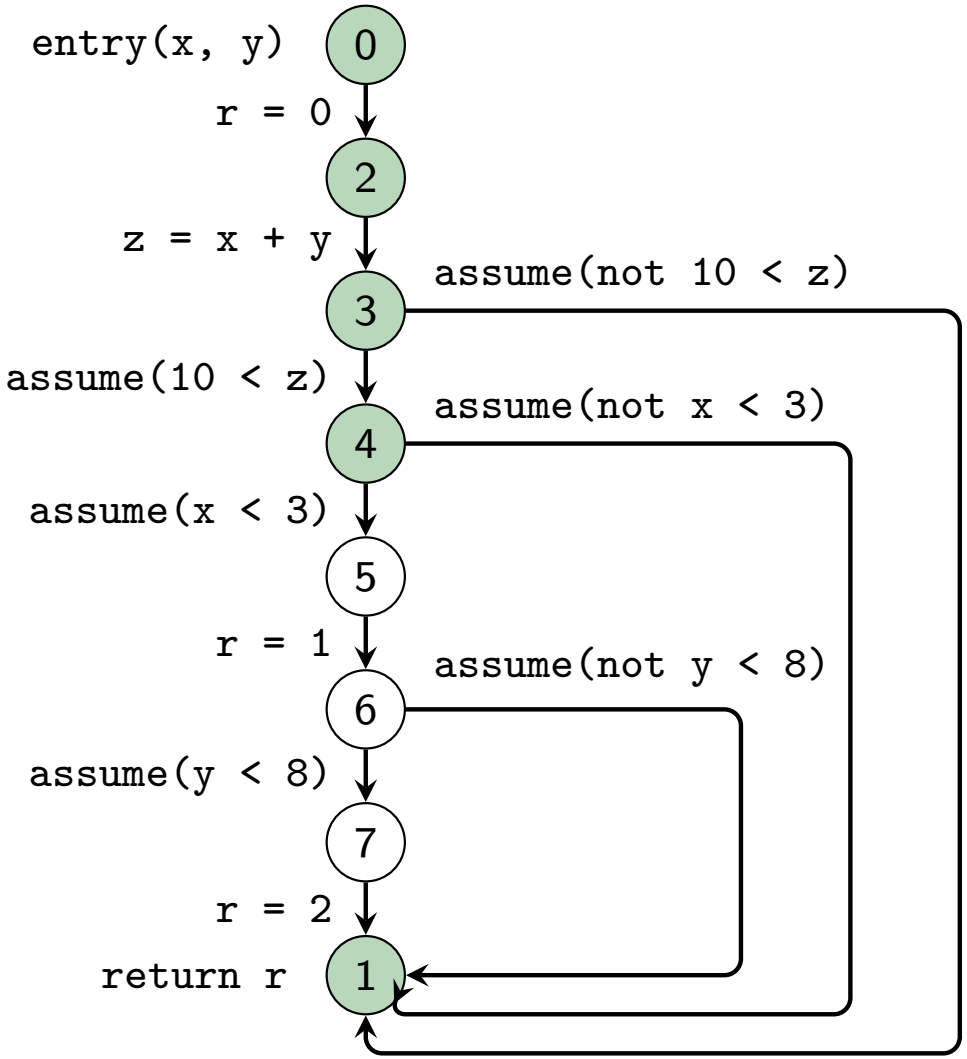
Fuzzing f



Change: add -8, -1, 1, 8, or 100 to each input.

run	kept	new
seed	$(0, 0)$	$0, 2, 3, 1$

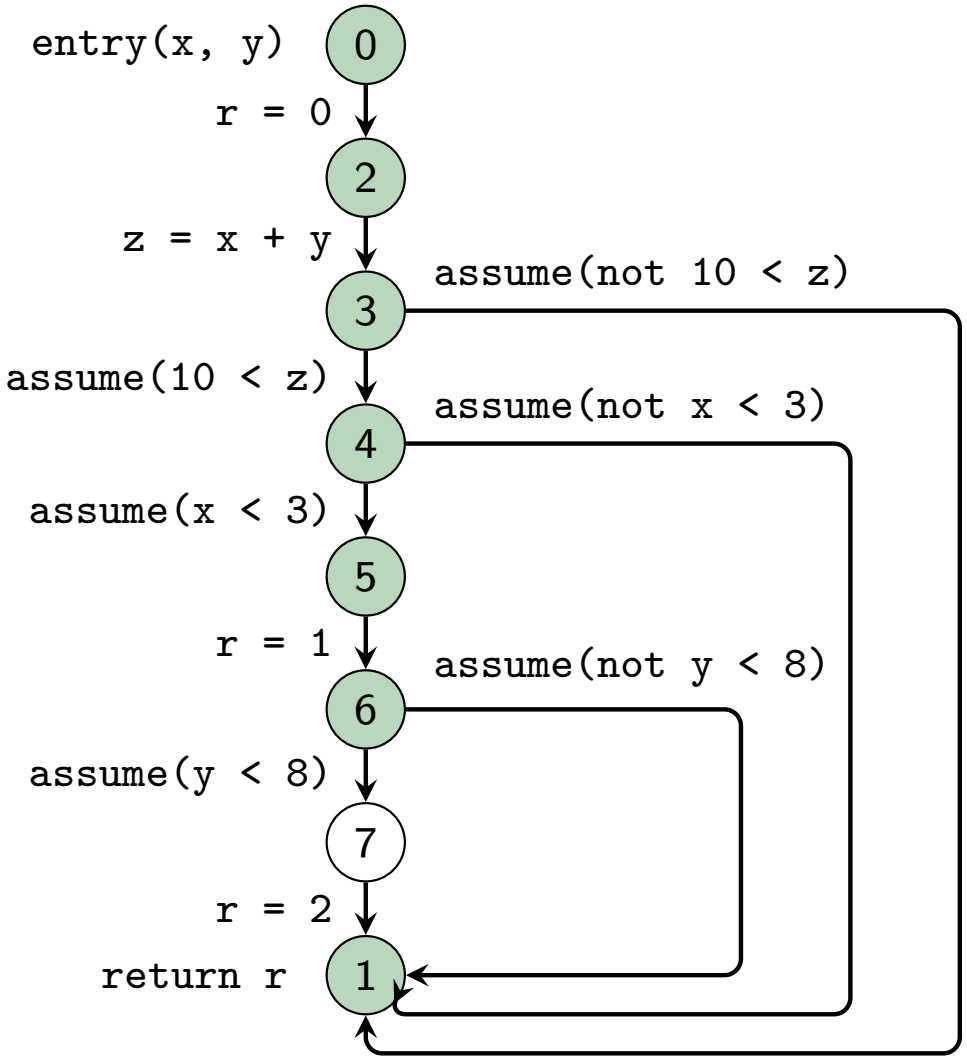
Fuzzing f



Change: add -8, -1, 1, 8, or 100 to each input.

run	kept	new
seed	$(0, 0)$	$0, 2, 3, 1$
2	$(100, 8)$	4

Fuzzing f

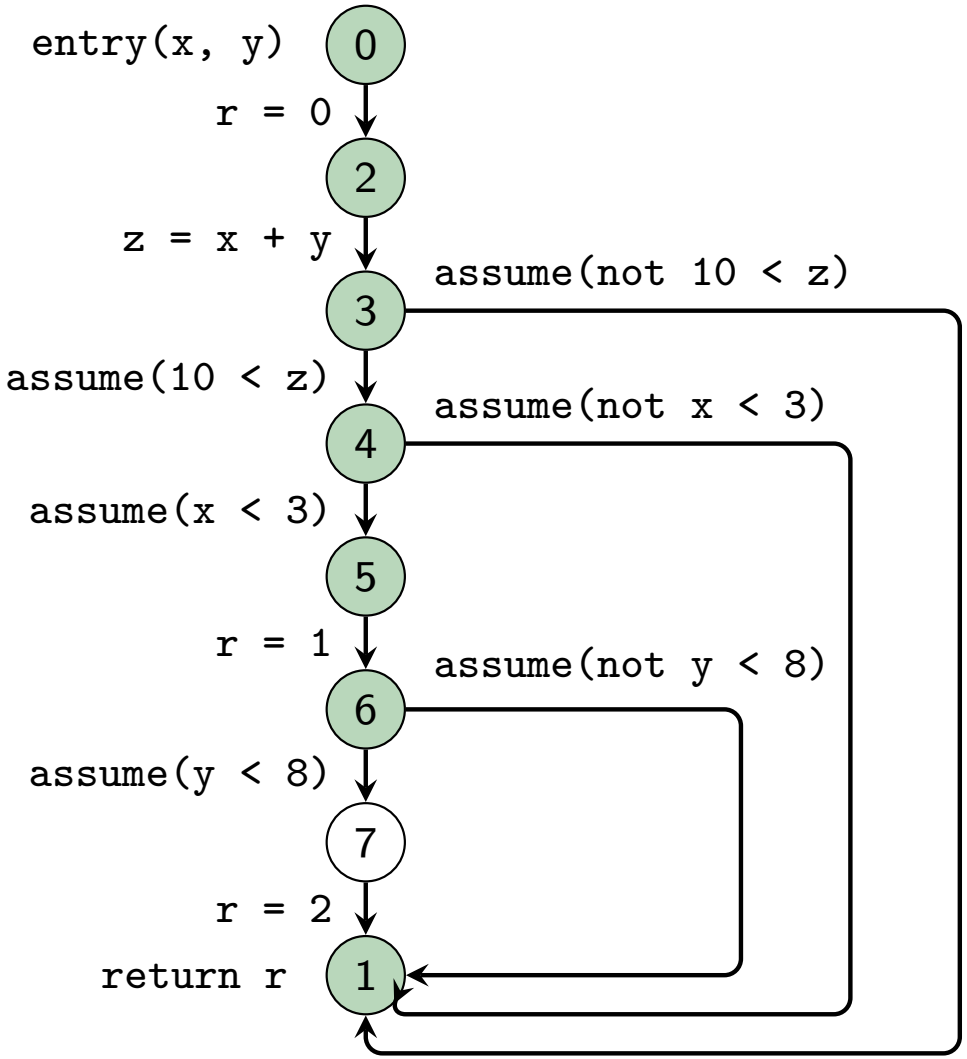


Change: add -8, -1, 1, 8, or 100 to each input.

run	kept	new
seed	$(0, 0)$	$0, 2, 3, 1$
2	$(100, 8)$	4
16	$(-1, 100)$	$5, 6$

- 7 of 8 points in 16 runs.

Fuzzing f



Change: add -8, -1, 1, 8, or 100 to each input.

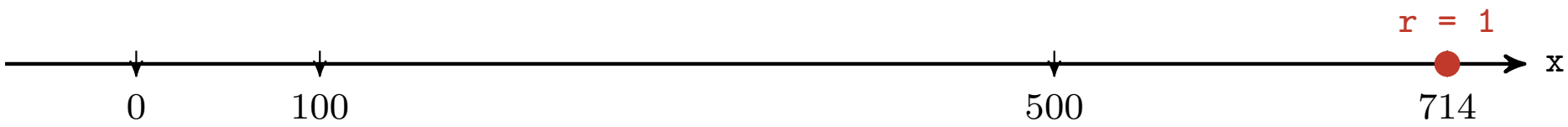
run	kept	new
seed	$(0, 0)$	$0, 2, 3, 1$
2	$(100, 8)$	4
16	$(-1, 100)$	$5, 6$
2000	—	—

- 7 of 8 points in 16 runs.
- Then nothing new: point 7 is dead.

Where Fuzzing Gets Stuck

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

runs	10	100	1000	10000
points	3/4	3/4	3/4	3/4

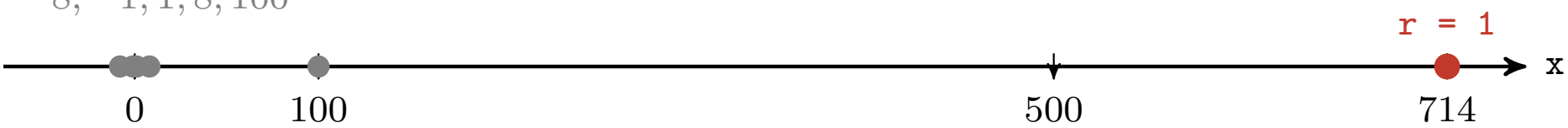


Where Fuzzing Gets Stuck

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

runs	10	100	1000	10000
points	3/4	3/4	3/4	3/4

all 10000 runs try only these five:
$-8, -1, 1, 8, 100$

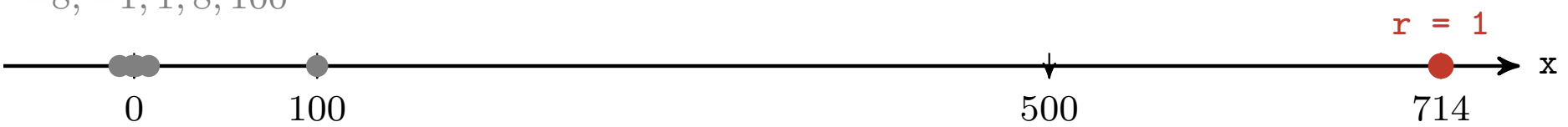


Where Fuzzing Gets Stuck

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

runs	10	100	1000	10000
points	3/4	3/4	3/4	3/4

all 10000 runs try only these five:
$-8, -1, 1, 8, 100$



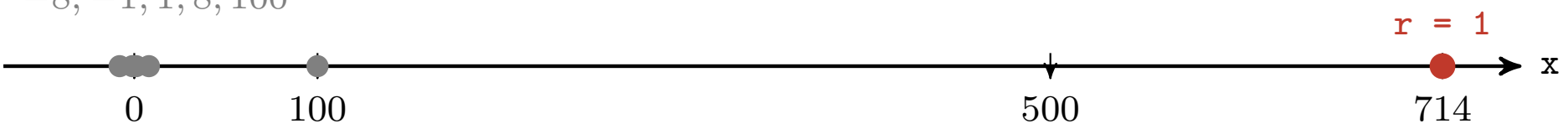
- No input is "closer" to 714: the branch is just false, so nothing new is ever kept.

Where Fuzzing Gets Stuck

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

runs	10	100	1000	10000
points	3/4	3/4	3/4	3/4

all 10000 runs try only these five:
$-8, -1, 1, 8, 100$



- No input is "closer" to 714: the branch is just false, so nothing new is ever kept.
- Magic numbers and checksums look like this. A solver finds 714 at once.

Contents

1. Watching a Run

2. What a Trace Tells
   - Coverage
   - Taint
   - More Analyses

3. Generating Inputs

4. Concolic Execution
   - Flip and Solve
   - Hybrid Fuzzing

5. In Practice

Concrete and Symbolic, Together

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

	concrete $(0, 0)$	**symbolic** (α, β)
`entry`	$\mathtt{x} = 0, \mathtt{y} = 0$	$\mathtt{x} \mapsto \alpha, \mathtt{y} \mapsto \beta$

- Run a **real** input, and keep the symbolic state of the **same path** next to it.

Concrete and Symbolic, Together

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

	concrete $(0, 0)$	**symbolic** (α, β)
`entry`	$\mathtt{x} = 0, \mathtt{y} = 0$	$\mathtt{x} \mapsto \alpha, \mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$

- Run a **real** input, and keep the symbolic state of the **same path** next to it.

Concrete and Symbolic, Together

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

	concrete $(0, 0)$	**symbolic** (α, β)
`entry`	$\mathtt{x} = 0, \mathtt{y} = 0$	$\mathtt{x} \mapsto \alpha, \mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$
`z = x + y`	$\mathtt{z} = 0$	$\mathtt{z} \mapsto \alpha + \beta$

- Run a **real** input, and keep the symbolic state of the **same path** next to it.

Concrete and Symbolic, Together

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

	concrete $(0, 0)$	**symbolic** (α, β)
`entry`	$\mathtt{x} = 0, \mathtt{y} = 0$	$\mathtt{x} \mapsto \alpha, \mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$
`z = x + y`	$\mathtt{z} = 0$	$\mathtt{z} \mapsto \alpha + \beta$
`10 < z`	`False`: skip	$\Phi = \neg(10 < \alpha + \beta)$

- Run a **real** input, and keep the symbolic state of the **same path** next to it.
- The real value picks the branch. The symbolic side writes down its condition.

Concrete and Symbolic, Together

```
def f(x, y):
    r = 0
    z = x + y
    if 10 < z:
        if x < 3:
            r = 1
            if y < 8:
                r = 2
    return r
```

	concrete $(0, 0)$	**symbolic** (α, β)
`entry`	$\mathtt{x} = 0, \mathtt{y} = 0$	$\mathtt{x} \mapsto \alpha, \mathtt{y} \mapsto \beta$
`r = 0`	$\mathtt{r} = 0$	$\mathtt{r} \mapsto 0$
`z = x + y`	$\mathtt{z} = 0$	$\mathtt{z} \mapsto \alpha + \beta$
`10 < z`	`False`: skip	$\Phi = \neg(10 < \alpha + \beta)$
`return r`	returns 0	

- Run a **real** input, and keep the symbolic state of the **same path** next to it.
- The real value picks the branch. The symbolic side writes down its condition.

Flip the Last Branch

run	input	path condition Φ	flip the last branch	solver
1	$(0, 0)$	$\neg(10 < \alpha + \beta)$	$10 < \alpha + \beta$	$(11, 0)$

1. Run an input, and collect its path condition.
2. Negate the last condition not yet tried, and ask the solver.
3. The model is the next input: same path up to there, then the **other** way.

Flip the Last Branch

run	input	path condition Φ	flip the last branch	solver
1	$(0, 0)$	$\neg(10 < \alpha + \beta)$	$10 < \alpha + \beta$	$(11, 0)$
2	$(11, 0)$	$10 < \alpha + \beta \wedge \neg(\alpha < 3)$	$10 < \alpha + \beta \wedge \alpha < 3$	$(2, 9)$

1. Run an input, and collect its path condition.
2. Negate the last condition not yet tried, and ask the solver.
3. The model is the next input: same path up to there, then the **other** way.

Flip the Last Branch

run	input	path condition Φ	flip the last branch	solver
1	$(0, 0)$	$\neg(10 < \alpha + \beta)$	$10 < \alpha + \beta$	$(11, 0)$
2	$(11, 0)$	$10 < \alpha + \beta \wedge \neg(\alpha < 3)$	$10 < \alpha + \beta \wedge \alpha < 3$	$(2, 9)$
3	$(2, 9)$	$\cdots \wedge \alpha < 3 \wedge \neg(\beta < 8)$	$\cdots \wedge \alpha < 3 \wedge \beta < 8$	unsat

1. Run an input, and collect its path condition.
2. Negate the last condition not yet tried, and ask the solver.
3. The model is the next input: same path up to there, then the **other** way.

Flip the Last Branch

run	input	path condition Φ	flip the last branch	solver
1	$(0, 0)$	$\neg(10 < \alpha + \beta)$	$10 < \alpha + \beta$	$(11, 0)$
2	$(11, 0)$	$10 < \alpha + \beta \wedge \neg(\alpha < 3)$	$10 < \alpha + \beta \wedge \alpha < 3$	$(2, 9)$
3	$(2, 9)$	$\cdots \wedge \alpha < 3 \wedge \neg(\beta < 8)$	$\cdots \wedge \alpha < 3 \wedge \beta < 8$	unsat

1. Run an input, and collect its path condition.
2. Negate the last condition not yet tried, and ask the solver.
3. The model is the next input: same path up to there, then the **other** way.

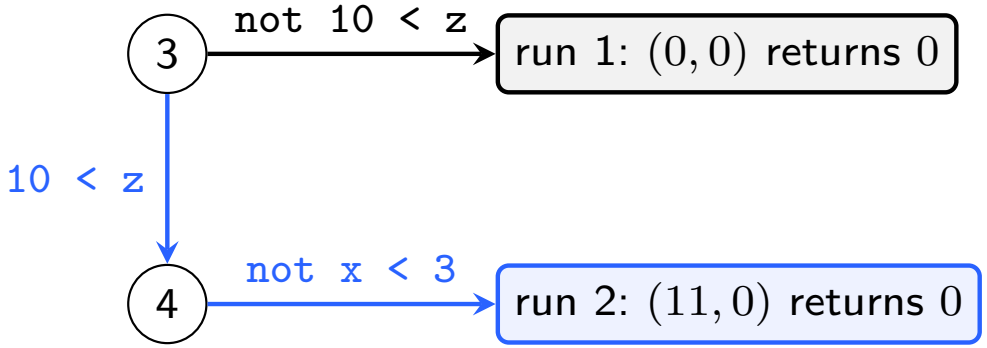
- Three runs: every branch but `y < 8` taken, and `y < 8` proved impossible.

The Same Tree, One Run at a Time



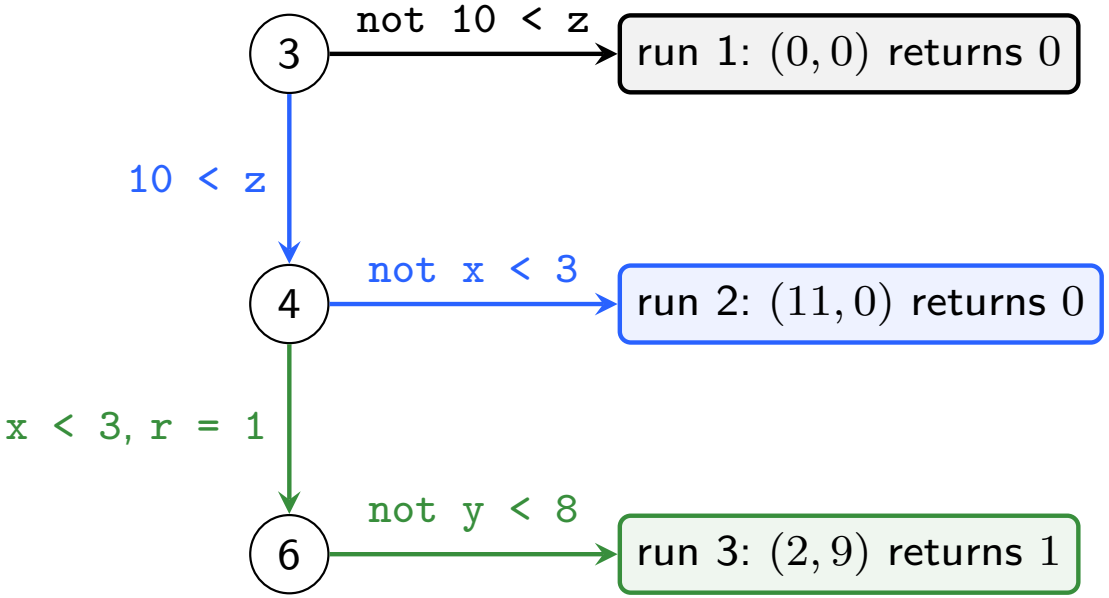
- Last time, symbolic execution built this whole tree at once.

The Same Tree, One Run at a Time



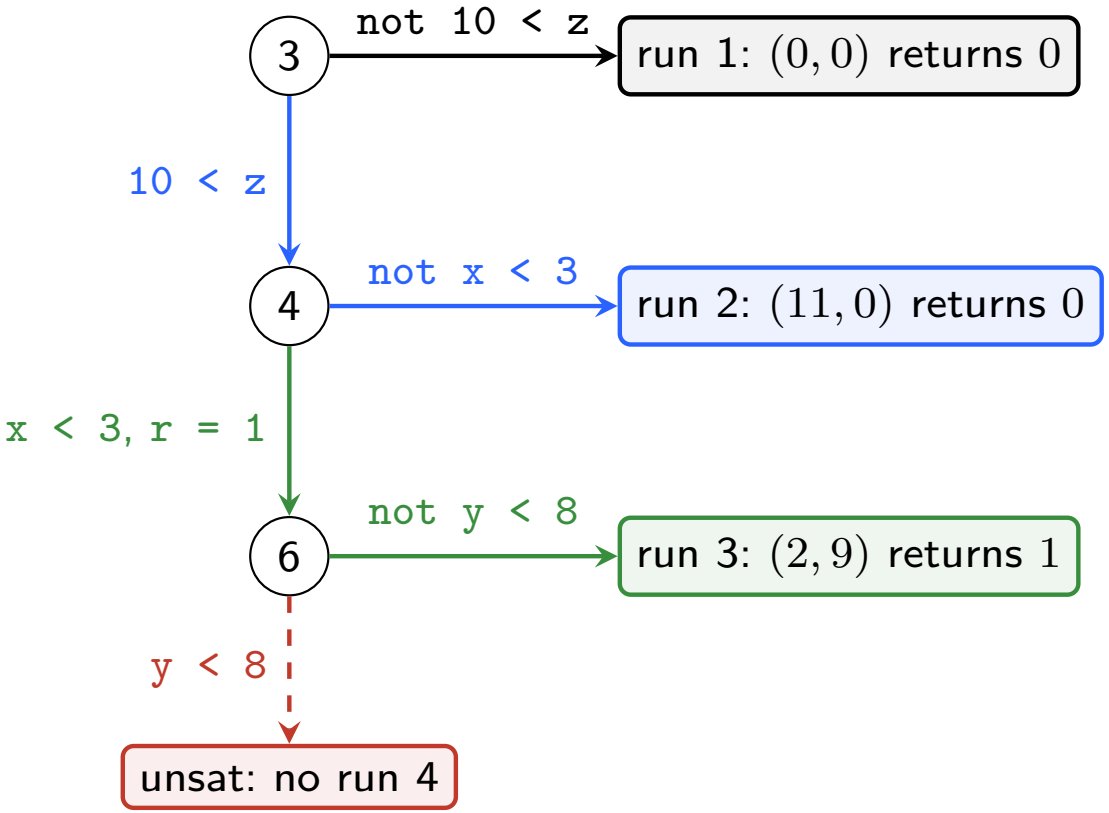
- Last time, symbolic execution built this whole tree at once.
- Concolic execution adds **one real path** per run.

The Same Tree, One Run at a Time



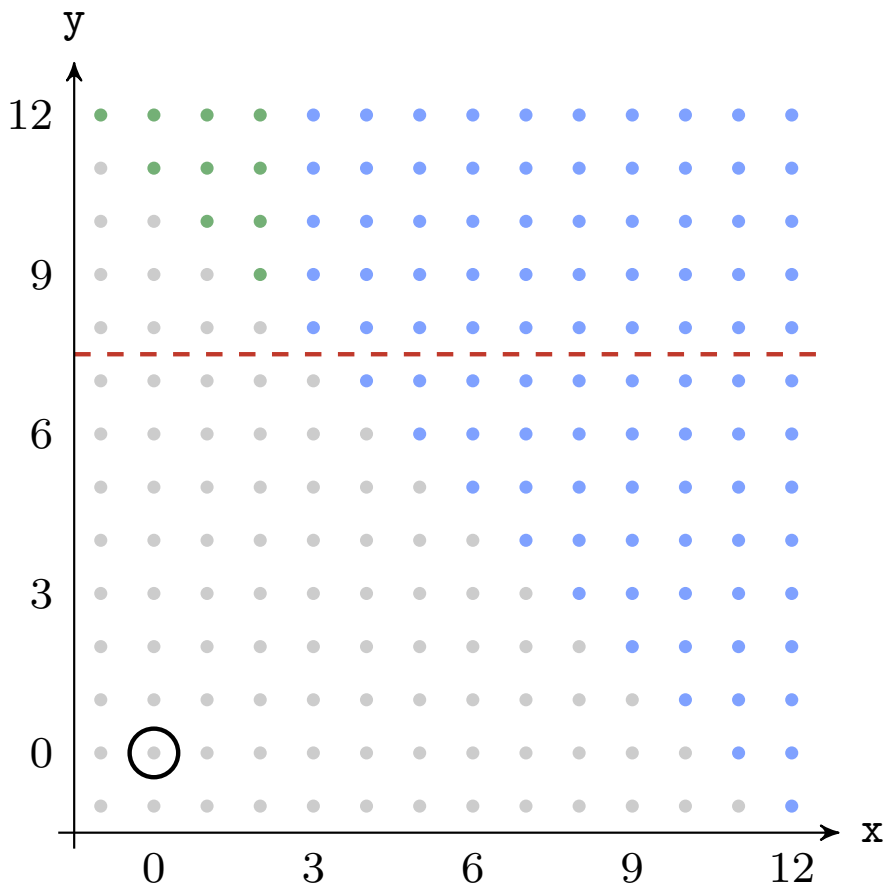
- Last time, symbolic execution built this whole tree at once.
- Concolic execution adds **one real path** per run.

The Same Tree, One Run at a Time



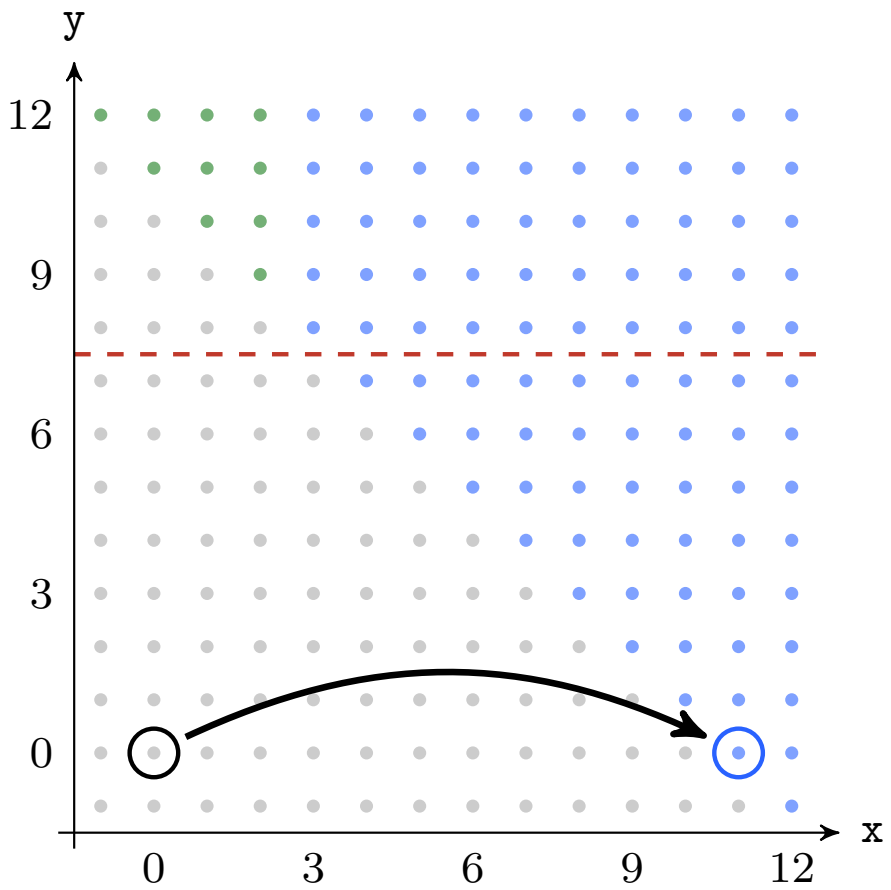
- Last time, symbolic execution built this whole tree at once.
- Concolic execution adds **one real path** per run.
- One solver call per flip.

Jumping Between Paths



- `not 10 < z`: returns 0
- `10 < z, not x < 3`: returns 0
- `10 < z, x < 3`: returns 1
- -- below it, `y < 8`

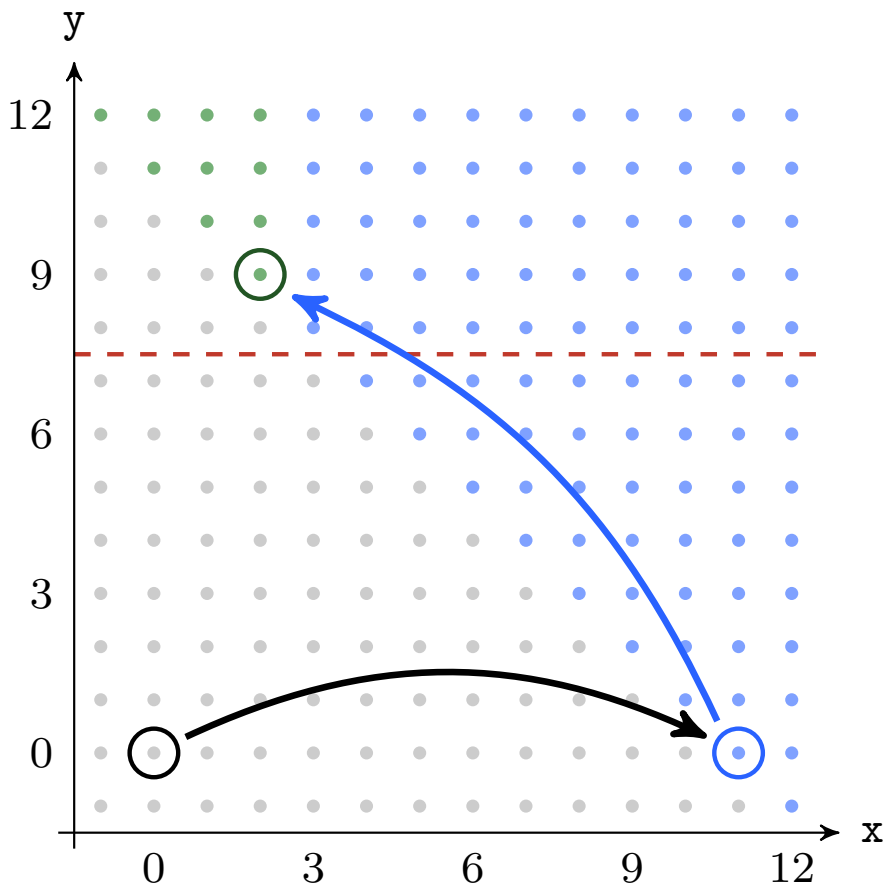
Jumping Between Paths



- `not 10 < z`: returns 0
- `10 < z, not x < 3`: returns 0
- `10 < z, x < 3`: returns 1
- -- below it, `y < 8`

- Each run lands in a **new** colour: a new path.

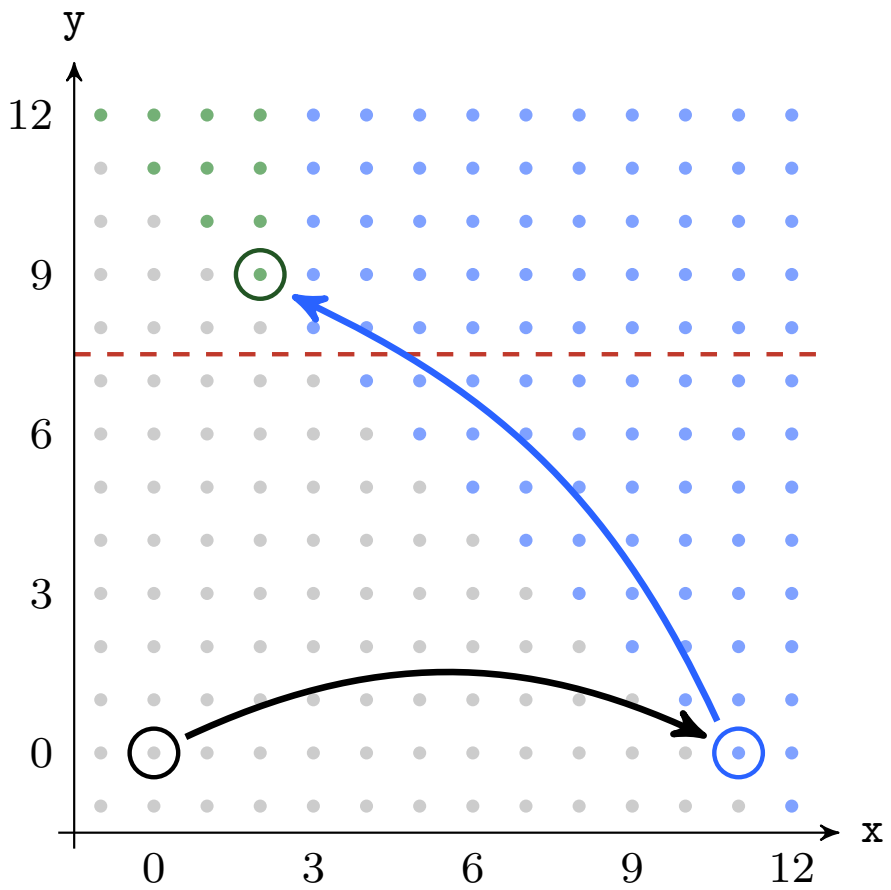
Jumping Between Paths



- `not 10 < z`: returns 0
- `10 < z, not x < 3`: returns 0
- `10 < z, x < 3`: returns 1
- -- below it, `y < 8`

- Each run lands in a **new** colour: a new path.
- No green point below the dashed line: no run can reach `r = 2`.

Jumping Between Paths



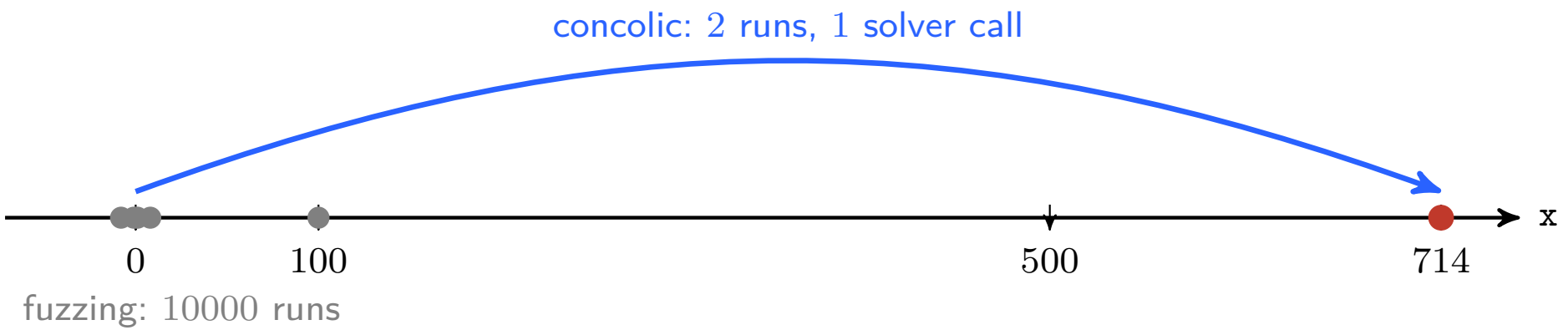
- `not 10 < z`: returns 0
- `10 < z, not x < 3`: returns 0
- `10 < z, x < 3`: returns 1
- below it, `y < 8`

- Each run lands in a **new** colour: a new path.
- No green point below the dashed line: no run can reach `r = 2`.
- A fuzzer moves by small steps. Concolic execution jumps.

The Magic Number, Found

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

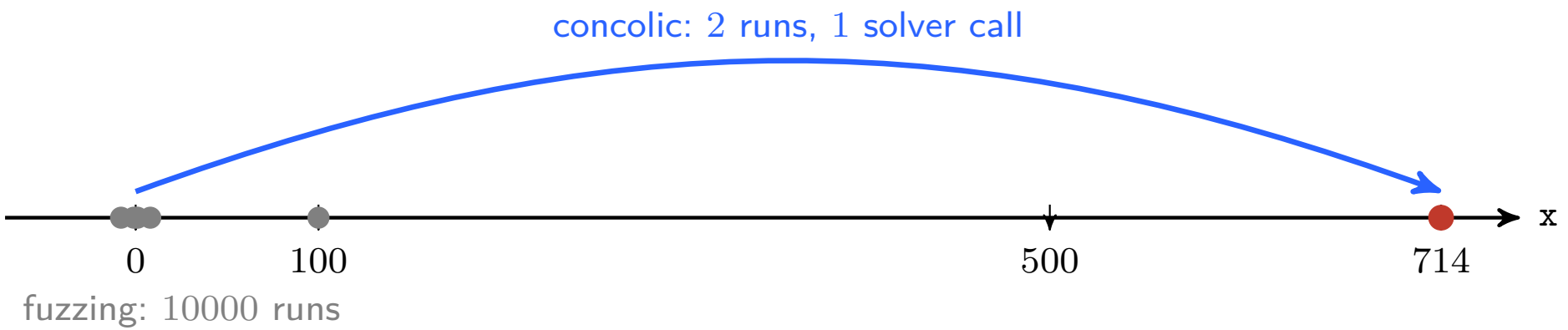
run	input	flip	solver
1	0	$\alpha \times 7 + 3 = 5001$	714



The Magic Number, Found

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

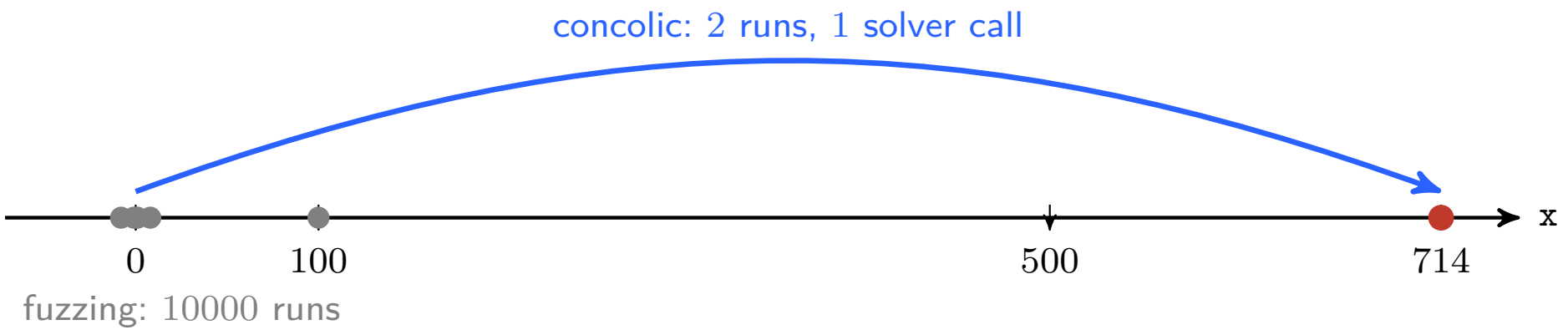
run	input	flip	solver
1	0	$\alpha \times 7 + 3 = 5001$	714
2	714	`r = 1` reached	



The Magic Number, Found

```
def magic(x):
    r = 0
    if x * 7 + 3 == 5001:
        r = 1
    return r
```

run	input	flip	solver
1	0	$\alpha \times 7 + 3 = 5001$	714
2	714	`r = 1` reached	



- The fuzzer guesses. The solver reads the condition.

When the Solver Cannot Follow

`checksum`: a library function the solver cannot see inside.

```
if y == checksum(x):
    ...                    # here
```

```
if checksum(x) == 1234:
    ...                    # here
```

When the Solver Cannot Follow

`checksum`: a library function the solver cannot see inside.

```
if y == checksum(x):
    ...                    # here
```

```
if checksum(x) == 1234:
    ...                    # here
```

- Symbolic: **stuck**.

When the Solver Cannot Follow

`checksum`: a library function the solver cannot see inside.

```
if y == checksum(x):
    ...                    # here
```

```
if checksum(x) == 1234:
    ...                    # here
```

- Symbolic: **stuck**.
- Concolic: run $(5, 0)$, and get `checksum(5)` $= c$ for real.

When the Solver Cannot Follow

`checksum`: a library function the solver cannot see inside.

```
if y == checksum(x):
    ...                    # here
```

```
if checksum(x) == 1234:
    ...                    # here
```

- Symbolic: **stuck**.
- Concolic: run $(5, 0)$, and get `checksum(5)` $= c$ for real.
- Solve $\beta = c$: the next run, $(5, c)$, gets **here**.

When the Solver Cannot Follow

`checksum`: a library function the solver cannot see inside.

```
if y == checksum(x):
    ...                    # here
```

- Symbolic: **stuck**.
- Concolic: run $(5, 0)$, and get `checksum(5)` $= c$ for real.
- Solve $\beta = c$: the next run, $(5, c)$, gets **here**.

```
if checksum(x) == 1234:
    ...                    # here
```

- Concolic: with `x` $= 5$, the condition is just $c = 1234$. `False`, and **nothing** to solve.

When the Solver Cannot Follow

`checksum`: a library function the solver cannot see inside.

```
if y == checksum(x):
    ...                    # here
```

- Symbolic: **stuck**.
- Concolic: run $(5, 0)$, and get `checksum(5)` $= c$ for real.
- Solve $\beta = c$: the next run, $(5, c)$, gets **here**.

```
if checksum(x) == 1234:
    ...                    # here
```

- Concolic: with `x` $= 5$, the condition is just $c = 1234$. `False`, and **nothing** to solve.
- Only luck gets **here**.

When the Solver Cannot Follow

`checksum`: a library function the solver cannot see inside.

```
if y == checksum(x):
    ...                    # here
```

- Symbolic: **stuck**.
- Concolic: run $(5, 0)$, and get `checksum(5)` $= c$ for real.
- Solve $\beta = c$: the next run, $(5, c)$, gets **here**.

```
if checksum(x) == 1234:
    ...                    # here
```

- Concolic: with `x` $= 5$, the condition is just $c = 1234$. `False`, and **nothing** to solve.
- Only luck gets **here**.

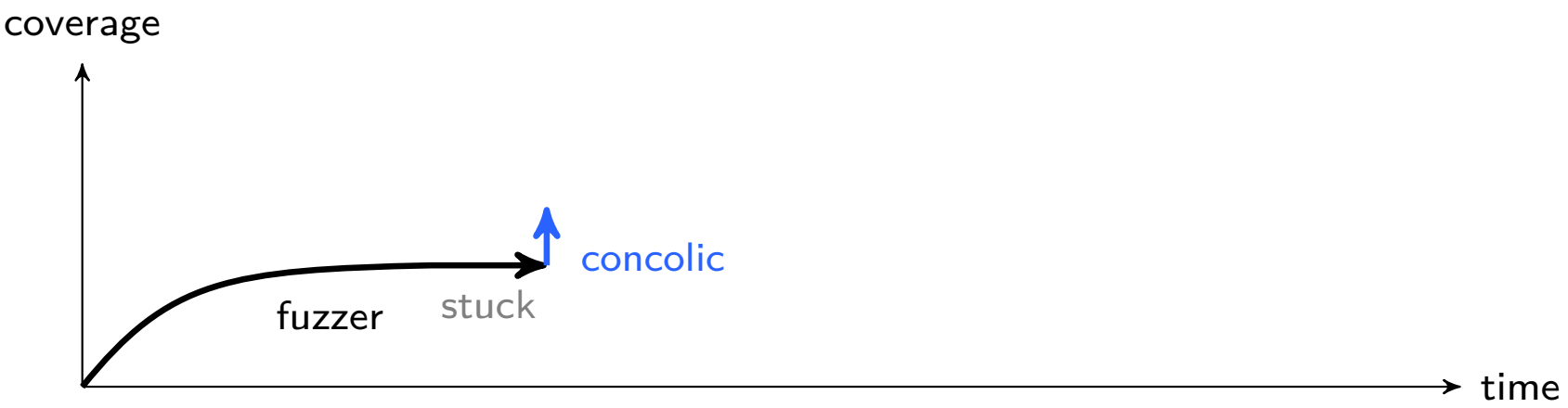
- Where the solver cannot follow, use the **real value**. Paths behind it may be missed.

Hybrid Fuzzing



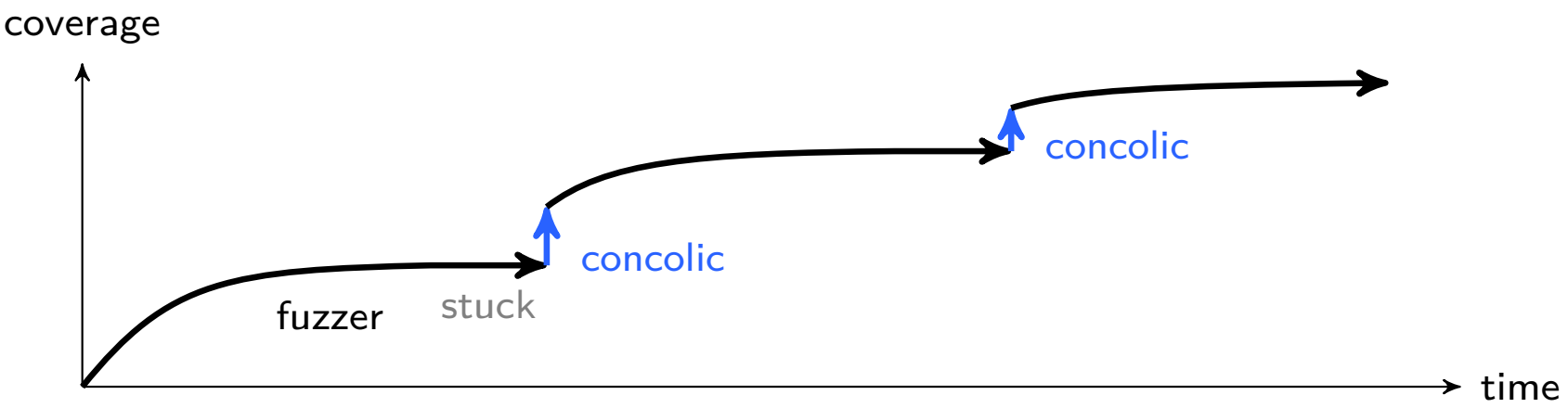
- The fuzzer is fast, and takes the easy branches.

Hybrid Fuzzing



- The fuzzer is fast, and takes the easy branches.
- When it is stuck, concolic execution solves the hard branch and hands back an input.

Hybrid Fuzzing



- The fuzzer is fast, and takes the easy branches.
- When it is stuck, concolic execution solves the hard branch and hands back an input.
- Then the fuzzer goes on. Driller (2016) and QSYM (2018) work this way.

Contents

1. Watching a Run

2. What a Trace Tells
   - Coverage
   - Taint
   - More Analyses

3. Generating Inputs

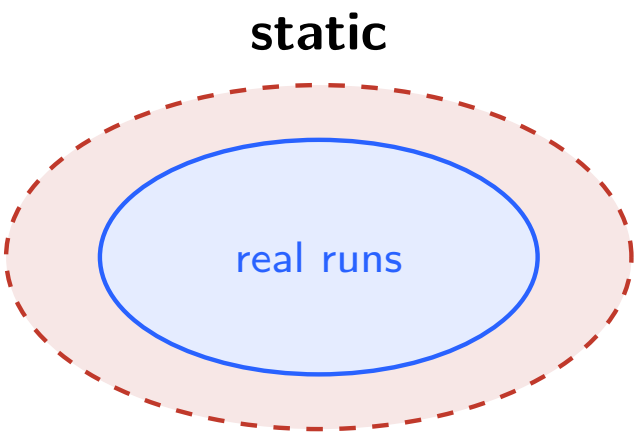
4. Concolic Execution
   - Flip and Solve
   - Hybrid Fuzzing

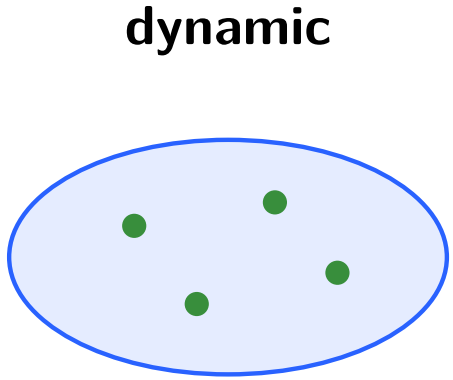
5. In Practice

Tools

tool	**what it does**
AFL++, libFuzzer	coverage-guided fuzzing
OSS-Fuzz	Google runs fuzzers on open-source projects, nonstop
AddressSanitizer	memory errors, by compiler instrumentation
Valgrind	memory errors in any binary, no recompiling
ThreadSanitizer	data races
SAGE, Driller, QSYM	concolic execution, hybrid fuzzing
TaintDroid	taint tracking on Android
Daikon	likely invariants

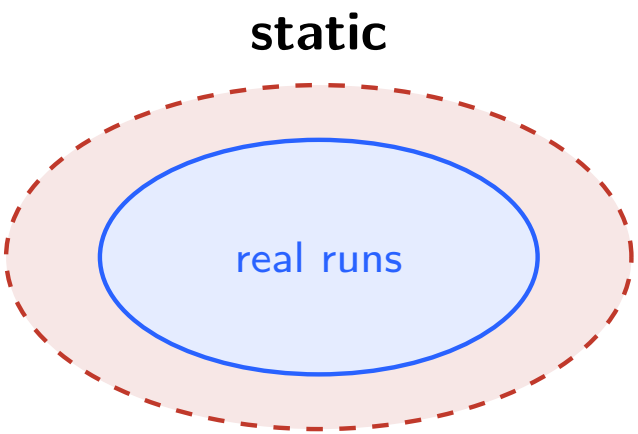
Static and Dynamic

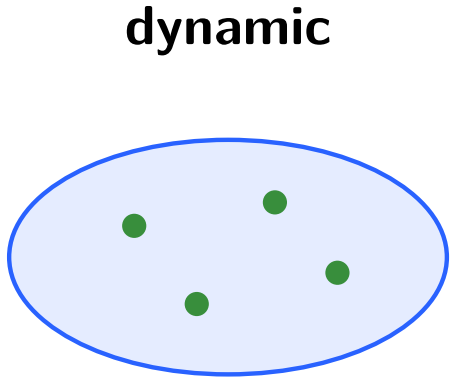




	static	**dynamic**
looks at	every run	the runs we made
a report	*may* be false	is always real
silence	means safe (if sound)	*may* be a miss

Static and Dynamic





	static	**dynamic**
looks at	every run	the runs we made
a report	*may* be false	is always real
silence	means safe (if sound)	*may* be a miss

- Concolic execution sits between: it runs, and lets the solver choose the next run.

Summary

PLRG

1. Watching a Run

2. What a Trace Tells
   - Coverage
   - Taint
   - More Analyses

3. Generating Inputs

4. Concolic Execution
   - Flip and Solve
   - Hybrid Fuzzing

5. In Practice

Next Lecture

- Real-World Tools and Practice

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr