CS 489 / 698: Software and Systems Security

Module 7: Bug Finding Tools and Practices
fuzz testing

Meng Xu *(University of Waterloo)*

Winter 2024

Outline

History: why do we call it "fuzzing"?

History: why do we call it "fuzzing"?

In 80's, someone remotely logged into a unix system over a dial-up
network link during a storm.

The rain caused a lot of random noise on the dial-up link.

And these noise caused applications that were using data off the
dial-up network line to crash.

History: why do we call it "fuzzing"?

In 80's, someone remotely logged into a unix system over a dial-up
network link during a storm.

The rain caused a lot of random noise on the dial-up link.

And these noise caused applications that were using data off the
dial-up network line to crash.

Gist of the story? — The rain tests the program way better than
human beings.

Intro
○○●○
Evolution
○○○○
Coverage
○○○○○○○○
Loop
○○○○○
Concolic
○○○○
Conclusion
○○○

The goal of fuzzing

Q: What is fuzzing doing essentially? Try to describe it in a way that is as abstract/general as possible.

The goal of fuzzing

Q: What is fuzzing doing essentially? Try to describe it in a way that is as abstract/general as possible.

A: To drive the execution of a system into desired states.

Intro
0000●
Evolution
0000
Coverage
00000000
Loop
00000
Concolic
0000
Conclusion
000

Elaborating on the definition

- What is special about the target system?
  - Do we know the source code?
  - Do we know the input format?
  - What are the challenges when executing the "system"?

- What do we mean by a state?
  - How can we tell that one state is different from another?

- What do we mean by desired?
  - New/unseen behavior?
  - Closeness to targeted execution points?

- What do we mean by driving the execution?
  - What can possibly be one mutation?
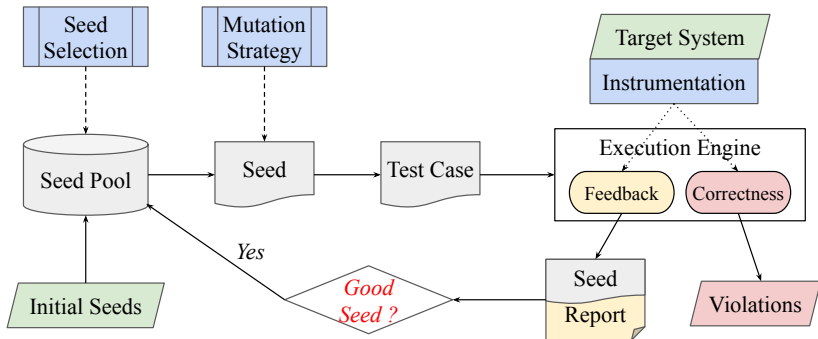 - How do you select the next mutation?

Outline

Intro
○○○○

Evolution
○●○○

Coverage
○○○○○○○○

Loop
○○○○○

Concolic
○○○○

Conclusion
○○○

Genetic algorithm

Training a program to play the snake game with genetic algorithm

Intro
oooo

Evolution
oo●o

Coverage
oooooooo

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Feedback-guided evolution process

Intro
○○○○

Evolution
○○●○

Coverage
○○○○○○○○

Loop
○○○○○

Concolic
○○○○

Conclusion
○○○

Feedback-guided evolution process
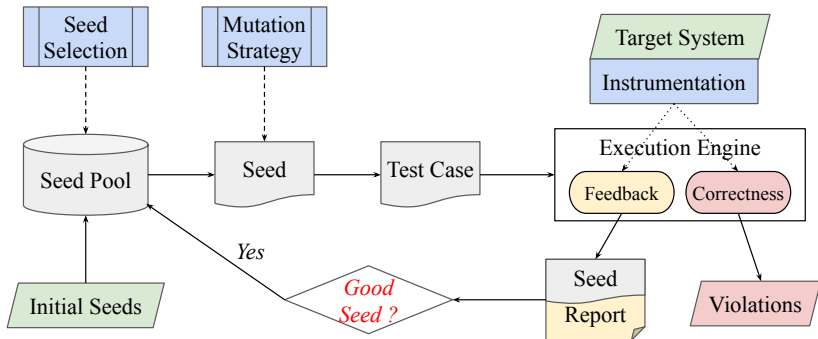
Natural selection — survival of the fittest

Intro
0000

Evolution
000●

Coverage
00000000

Loop
00000

Concolic
0000

Conclusion
000

Demo with AFL++

Acknowledgement: this demo is based on one of the examples used in the "Fuzzing with AFL" workshop by Michael Macnair.

Outline

1. **Introduction**

2. Evolution: from the rain-fuzzer to modern fuzzing

3. Program state coverage: "natural selection" in the fuzzing world

4. Loops: another trouble maker for branch coverage

5. Concolic execution: forced path exploration

6. Conclusion

Intro
oooo

Evolution
oooo

Coverage
o●oooooo

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Intuition: what makes a high-quality seed?

Intro
oooo

Evolution
oooo

Coverage
o●oooooo

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Intuition: what makes a high-quality seed?

```
1  pub fn foo(a: num, b: num) {
2      let c = if (a >= 0) {
3          1
4      } else {
5          2
6      };
7
8      // irrelevant operations
9
10     let d = if (b >= 0) {
11         2
12     } else {
13         3
14     };
15
16     // irrelevant operations
17
18     assert!(c != d);
19 }
```

Q: What is the testing plan?

Intro
oooo

Evolution
oooo

Coverage
o●oooooo

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Intuition: what makes a high-quality seed?

```
1  pub fn foo(a: num, b: num) {
2      let c = if (a >= 0) {
3          1
4      } else {
5          2
6      };
7
8      // irrelevant operations
9
10     let d = if (b >= 0) {
11         2
12     } else {
13         3
14     };
15
16     // irrelevant operations
17
18     assert!(c != d);
19 }
```

Q: What is the testing plan?

- Cover every line?
- Cover every if-else branch?
- Cover every exit status?
- Cover every path?

Intuition: what makes a high-quality seed?

```
1  pub fn foo(a: num, b: num) {
2      let c = if (a >= 0) {
3          1
4      } else {
5          2
6      };
7
8      // irrelevant operations
9
10     let d = if (b >= 0) {
11         2
12     } else {
13         3
14     };
15
16     // irrelevant operations
17
18     assert!(c != d);
19 }
```
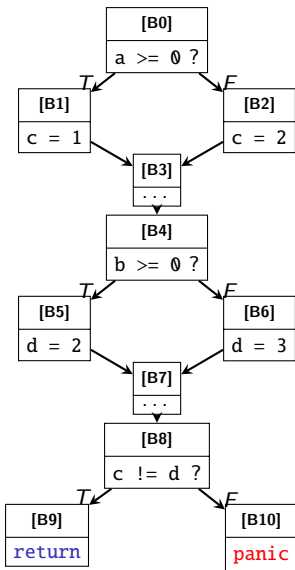
Q: What is the testing plan?

- Cover every line?
- Cover every if-else branch?
- Cover every exit status?
- Cover every path?

$\implies$ if the fuzzer generates an input that expands the coverage, that input is a good seed.

Intro
oooo

Evolution
oooo

Coverage
ooo●oooooo

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Illustration of different coverage metrics

```
1  pub fn foo(a: num, b: num) {
2      let c = if (a >= 0) {
3          1
4      } else {
5          2
6      };
7
8      // irrelevant operations
9
10     let d = if (b >= 0) {
11         2
12     } else {
13         3
14     };
15
16     // irrelevant operations
17
18     assert!(c != d);
19 }
```

Intro
○○○○

Evolution
○○○○

Coverage
○○○○●○○○○

Loop
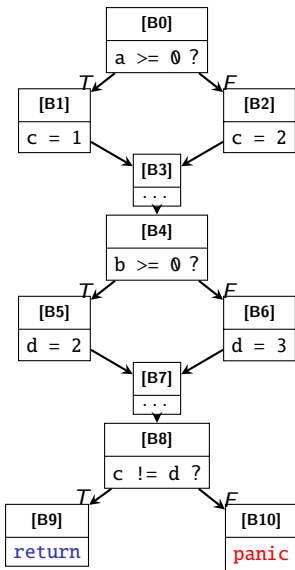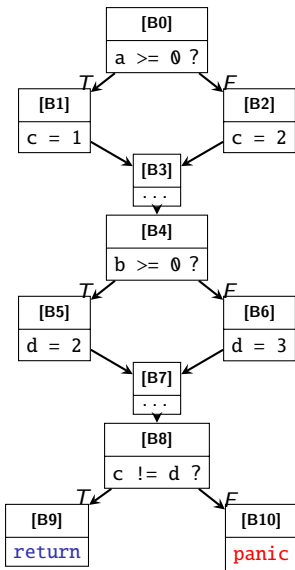○○○○○

Concolic
○○○○

Conclusion
○○○

Illustration of different coverage metrics

- Cover every line?
  - Block coverage

- Cover every if-else branch?
  - Branch coverage

- Cover every exit status?
  - Return coverage

- Cover every path?
  - Path coverage

Intro
○○○○

Evolution
○○○○

Coverage
○○○○●○○○○

Loop
○○○○○

Concolic
○○○○

Conclusion
○○○

Illustration of different coverage metrics

- Cover every line?
  - Block coverage

- Cover every if-else branch?
  - Branch coverage

- Cover every exit status?
  - Return coverage
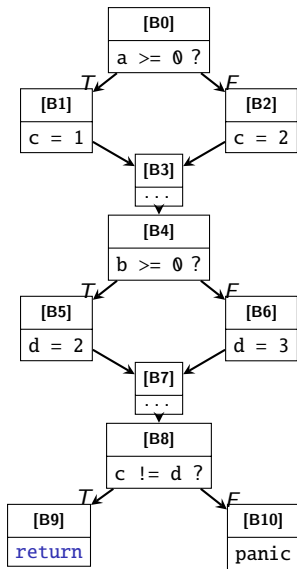
- Cover every path?
  - Path coverage

Path coverage: a theoretical optimum

Claim: A program is saturately tested if we obtain a set of inputs that covers every feasible path of the program CFG.
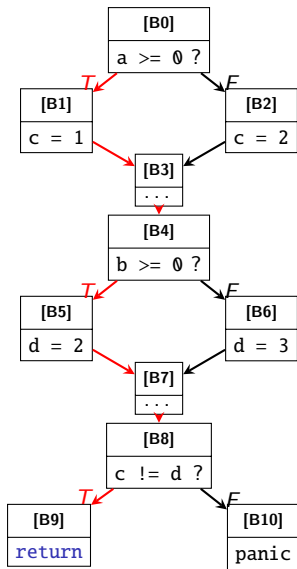
NOTE: feasible paths include paths that leads to explicit and implicit panics.

Intro
oooo

Evolution
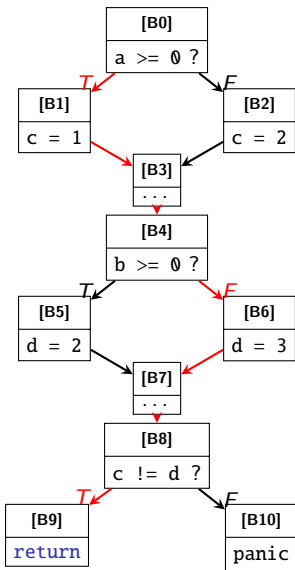oooo

Coverage
ooooo●oo

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Path coverage demo

Path coverage demo

- `a = 1, b = 1`

Intro
oooo
Evolution
oooo
Coverage
ooooo●oo
Loop
ooooo
Concolic
oooo
Conclusion
ooo

Path coverage demo

- `a = 1, b = 1`
- `a = 1, b = -1`

Intro
oooo

Evolution
oooo

Coverage
ooooo●oo

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Path coverage demo

- `a = 1, b = 1`
- `a = 1, b = -1`
- `a = -1, b = 1`

Intro
oooo

Evolution
oooo

Coverage
ooooo●oo

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Path coverage demo

- `a = 1, b = 1`
- `a = 1, b = -1`
- `a = -1, b = 1`
- `a = -1, b = -1`

Intro
0000
Evolution
0000
Coverage
00000●00
Loop
00000
Concolic
0000
Conclusion
000
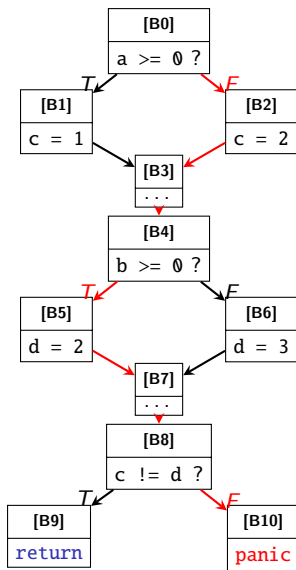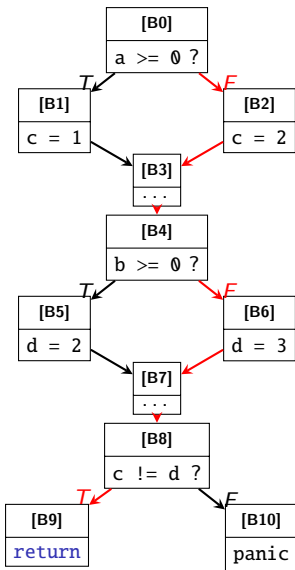
Path coverage demo

- a = 1, b = 1
- a = 1, b = -1
- a = -1, b = 1
- a = -1, b = -1

No new program behaviors can be discovered $\implies$ the program is saturately tested

```
[B0]
a >= 0 ?
```
T   F

```
[B1]        [B2]
c = 1       c = 2
```

```
[B3]
...
```

```
[B4]
b >= 0 ?
```
T   F

```
[B5]        [B6]
d = 2       d = 3
```

```
[B7]
...
```

```
[B8]
c != d ?
```
T   F

```
[B9]        [B10]
return      panic
```

Intro
0000

Evolution
0000

Coverage
0000000●0

Loop
00000

Concolic
0000

Conclusion
000

Why not path coverage in practice?

Intro
oooo
Evolution
oooo
Coverage
oooooooeo
Loop
ooooo
Concolic
oooo
Conclusion
ooo

Why not path coverage in practice?

Short answer: I don't know... AFL (American Fuzzy Lop) didn't adopt path coverage, so everyone follows suite...

Long answer:

- tracking block / branch coverage is stateless while tracking path coverage requires stateful instrumentations.
- different parts of the execution are not necessarily related, i.e., a new path does not necessarily mean interesting findings.
- it is hard to quantitatively measure the completeness of path coverage (because of infeasible paths). But by default, all branches should be somewhat feasible.
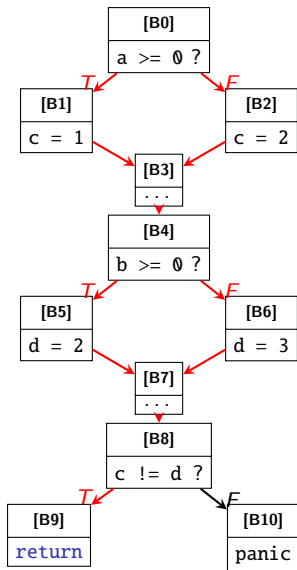
Intro
oooo

Evolution
oooo

Coverage
ooooooo●o

Loop
ooooo

Concolic
oooo

Conclusion
ooo

Why not path coverage in practice?

Short answer: I don't know... AFL (American Fuzzy Lop) didn't adopt path coverage, so everyone follows suite...

Long answer:

- tracking block / branch coverage is stateless while tracking path coverage requires stateful instrumentations.
- different parts of the execution are not necessarily related, i.e., a new path does not necessarily mean interesting findings.
- it is hard to quantitatively measure the completeness of path coverage (because of infeasible paths). But by default, all branches should be somewhat feasible.

In practice, branch coverage hits a nice balance between effectiveness and easiness of instrumentation.

Intro
oooo

Evolution
oooo

Coverage
ooooooo●

Loop
ooooo

Concolic
oooo

Conclusion
ooo

What's wrong with branch coverage?

Intro
oooo

Evolution
oooo

Coverage
ooooooo●

Loop
ooooo

Concolic
oooo

Conclusion
ooo

What's wrong with branch coverage?

- `a = 1, b = 1`
- `a = -1, b = -1`

Two seeds already covered most of the branches.

Intro
oooo

Evolution
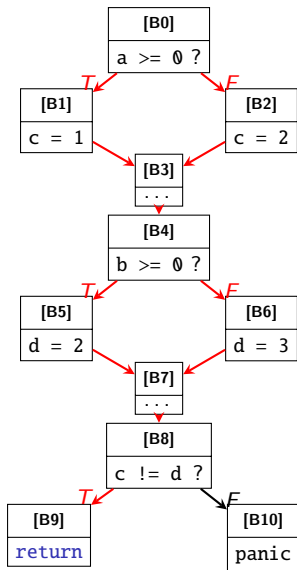oooo

Coverage
ooooooo●

Loop
ooooo

Concolic
oooo

Conclusion
ooo

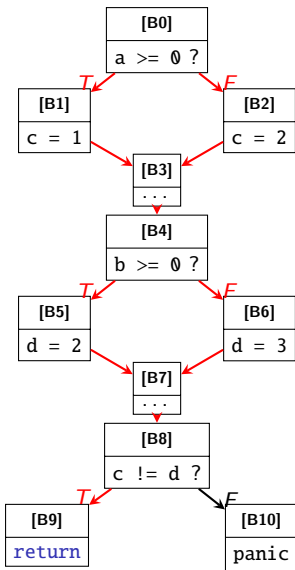# What's wrong with branch coverage?

- `a = 1, b = 1`
- `a = -1, b = -1`

Two seeds already covered most of the branches.

- `a = 1, b = -1`

A seed that yields new path but is considered as a bad seed as it yields no new branch coverage.

Intro
0000

Evolution
0000

Coverage
0000000●

Loop
00000

Concolic
0000

Conclusion
000

What's wrong with branch coverage?

- `a = 1, b = 1`
- `a = -1, b = -1`

Two seeds already covered most of the branches.

- `a = 1, b = -1`

A seed that yields new path but is considered as a bad seed as it yields no new branch coverage.

$\implies$ fuzzer is not rewarded by mutating a and b, hence, lowering their priorities and the panic case may never be found,

Intro
0000

Evolution
0000

Coverage
000000●

Loop
00000

Concolic
0000

Conclusion
000

What's wrong with branch coverage?

- a = 1, b = 1
- a = -1, b = -1

Two seeds already covered most of the branches.

- a = 1, b = -1

A seed that yields new path but is considered as a bad seed as it yields no new branch coverage.

$\implies$ fuzzer is not rewarded by mutating a and b, hence, lowering their priorities and the panic case may never be found, especially when fuzzing complex CFGs
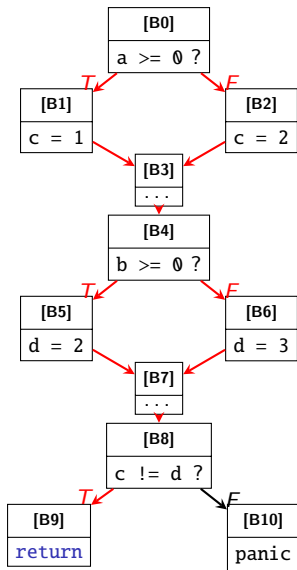
# Outline

1. **Introduction**

2. **Evolution: from the rain-fuzzer to modern fuzzing**

3. **Program state coverage: "natural selection" in the fuzzing world**

4. **Loops: another trouble maker for branch coverage**

5. **Concolic execution: forced path exploration**

6. **Conclusion**

Intro
oooo

Evolution
oooo

Coverage
oooooooo

Loop
oooooo

Concolic
oooo

Conclusion
ooo

Looping example

```rust
1  pub fn looping(
2    x: num,
3    y: num,
4    n: num
5  ) {
6    let i = 0;
7    while (x < n) {
8      if (y > x) {
9        x++;
10     }
11     else {
12       y++;
13     }
14     i++;
15   }
16   assert!(i != 7);
17 }
```
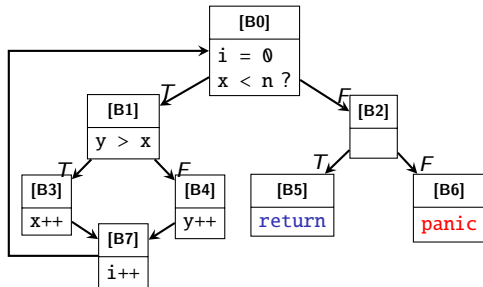
Looping example

```
1  pub fn looping(
2    x: num,
3    y: num,
4    n: num
5  ) {
6    let i = 0;
7    while (x < n) {
8      if (y > x) {
9        x++;
10     }
11     else {
12       y++;
13     }
14     i++;
15   }
16   assert!(i != 7);
17 }
```
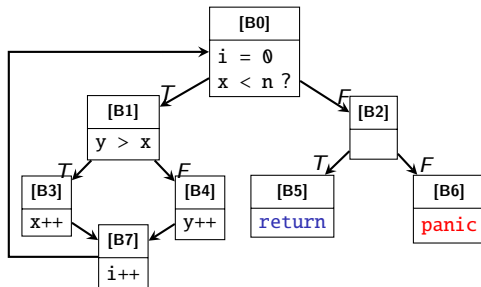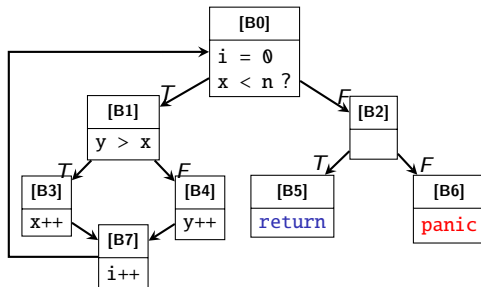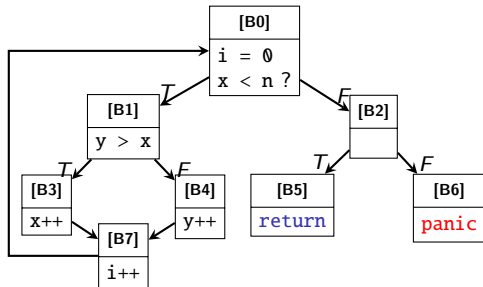


```
y <= x < n
```
1 y++ until y == x
2 y++; x++ until x == n

Looping example

```
1  pub fn looping(
2    x: num,
3    y: num,
4    n: num
5  ) {
6    let i = 0;
7    while (x < n) {
8      if (y > x) {
9        x++;
10     }
11     else {
12       y++;
13     }
14     i++;
15   }
16   assert!(i != 7);
17 }
```


y <= x < n
1 y++ until y == x
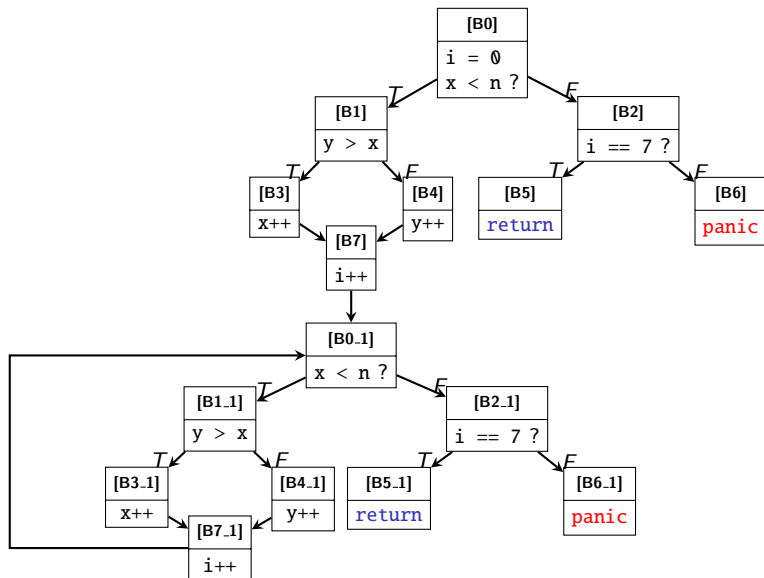2 y++; x++ until x == n

x < y <= n
1 x++ until x == y
2 y++; x++ until x == n

Intro
oooo

Evolution
oooo

Coverage
ooooooooo

Loop
oooooo

Concolic
oooo

Conclusion
ooo

Looping example

```
1  pub fn looping(
2    x: num,
3    y: num,
4    n: num
5  ) {
6    let i = 0;
7    while (x < n) {
8      if (y > x) {
9        x++;
10     }
11     else {
12       y++;
13     }
14     i++;
15   }
16   assert!(i != 7);
17 }
```


| [B0] |
| i = 0 |
| x < n ? |

[B1]
y > x

[B2]

[B3]
x++

[B4]
y++

[B5]
return

[B6]
panic

[B7]
i++

y <= x < n
1 y++ until y == x
2 y++; x++ until x == n

x < y <= n
1 x++ until x == y
2 y++; x++ until x == n

x < n <= y
1 x++ until x == n

Intro
oooo

Evolution
oooo

Coverage
ooooooooo

Loop
oooeoo

Concolic
oooo

Conclusion
ooo

Solution: bounded loop unrolling

Solution: bounded loop unrolling
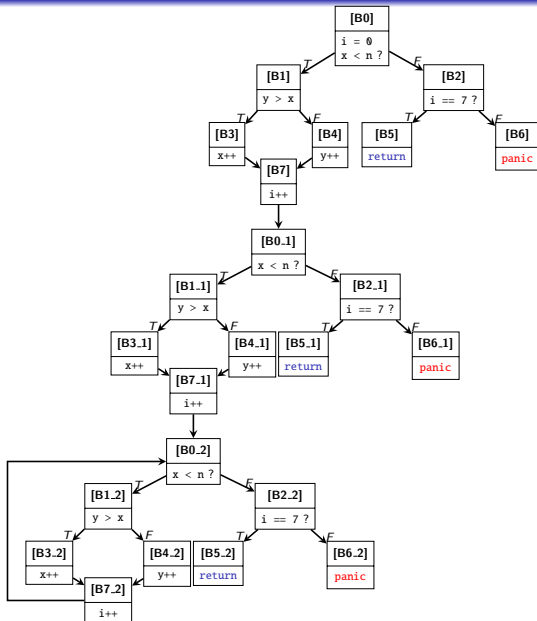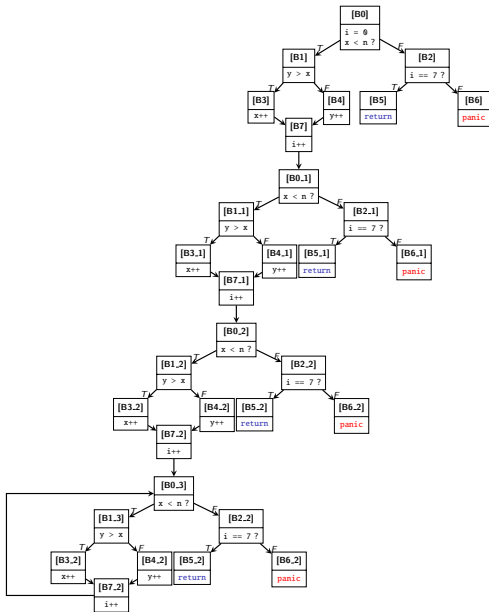
Intro
0000

Evolution
0000

Coverage
00000000

Loop
00000

Concolic
0000

Conclusion
000

Solution: bounded loop unrolling

Intro
0000

Evolution
0000

Coverage
00000000

Loop
00000

Concolic
●000

Conclusion
000

Outline

1. **Introduction**

2. Evolution: from the rain-fuzzer to modern fuzzing

3. Program state coverage: "natural selection" in the fuzzing world

4. Loops: another trouble maker for branch coverage

5. Concolic execution: forced path exploration

6. Conclusion

Intro
0000

Evolution
0000

Coverage
00000000

Loop
00000

Concolic
0●00

Conclusion
000

Narrow-range constraints

Random input generation is not suitable for passing narrow-ranged constraints. For example:

```
1 fn foo(x: u64, y: u64) {
2    if x + y = 42 {
3      panic!();
4    }
5 }
```

If x and y are randomly generated u64, the chances that their sum equals 42 is extremely low.

Narrow-range constraints

Random input generation is not suitable for passing narrow-ranged
constraints. For example:

```
1 fn foo(x: u64, y: u64) {
2   if x + y = 42 {
3     panic!();
4   }
5 }
```

If x and y are randomly generated u64, the chances that their sum
equals 42 is extremely low.

On the other hand, this is much easier for SMT solvers to produce
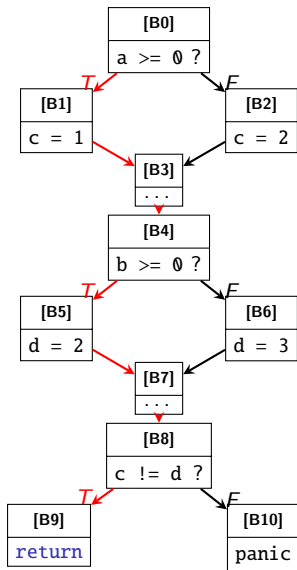valid values for x and y that satisfies this constraint.

The general intuition behind concolic execution

Let fuzzing do must of the state exploration. If the coverage saturates, i.e., the fuzzer is not able to make progress on finding new coverage, invoke the symbolic reasoning engine to breakthrough.

Intro
oooo

Evolution
oooo

Coverage
oooooooo

Loop
ooooo

Concolic
oooo●

Conclusion
ooo

Concolic execution demo

- a = 1, b = 1

We start with a sample input for the program, and execute the input concretely to obtain an execution trace.

Intro
○○○○

Evolution
○○○○

Coverage
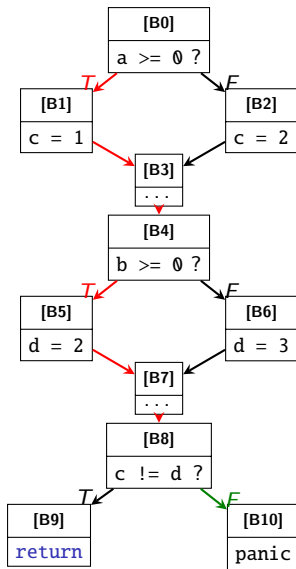○○○○○○○○

Loop
○○○○○

Concolic
○○○●

Conclusion
○○○

Concolic execution demo

- a = 1, b = 1

We start with a sample input for the program, and execute the input concretely to obtain an execution trace.

Query 1: given constraint $a \geq 0 \wedge b \geq 0$ and the program, can we toggle $c \neq d$?
$\implies$ unsat, infeasible path

Intro
0000

Evolution
0000

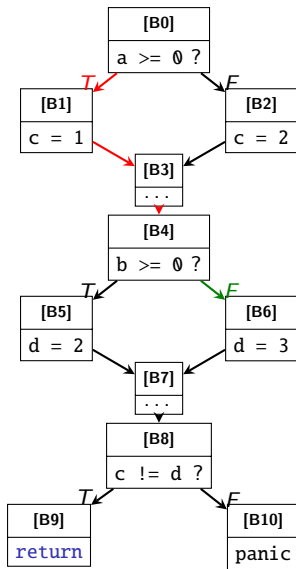Coverage
0000000

Loop
00000

Concolic
000●

Conclusion
000

Concolic execution demo

- a = 1, b = 1

We start with a sample input for the program, and execute the input concretely to obtain an execution trace.

Query 1: given constraint $a \geq 0 \wedge b \geq 0$ and the program, can we toggle $c \neq d$?
$\implies$ unsat, infeasible path

Query 2: given constraint $a \geq 0$ and the program, can we toggle $b \geq 0$?
$\implies$ sat, $a = 1$, $b = -1$

Intro
oooo

Evolution
oooo

Coverage
ooooooooo

Loop
ooooo

Concolic
ooo●

Conclusion
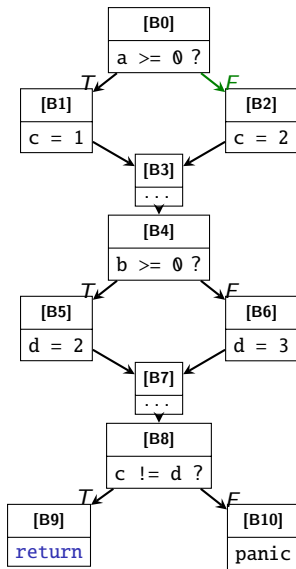ooo

Concolic execution demo

- a = 1, b = 1

We start with a sample input for the program, and execute the input concretely to obtain an execution trace.

Query 1: given constraint $a \geq 0 \wedge b \geq 0$ and the program, can we toggle $c \neq d$?
$\implies$ unsat, infeasible path

Query 2: given constraint $a \geq 0$ and the program, can we toggle $b \geq 0$?
$\implies$ sat, $a = 1, b = -1$

Query 3: given constraint *true* and the program, can we toggle $a \geq 0$?
$\implies$ sat, $a = -1$

Intro
0000

Evolution
0000

Coverage
00000000

Loop
00000

Concolic
0000

Conclusion
●○○

Outline

Intro
oooo

Evolution
oooo

Coverage
ooooooooo

Loop
ooooo

Concolic
oooo

Conclusion
o●o

A comprehensive survey of current works

Fuzzing Family Tree

Intro
oooo

Evolution
oooo

Coverage
oooooooo

Loop
ooooo

Concolic
oooo

Conclusion
oo●

⟨ **End** ⟩