CS 489 / 698: Software and Systems Security

Module 2: Program Security (Defenses)
fuzz testing

Meng Xu *(University of Waterloo)*

Spring 2023

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

History: why we call it “fuzzing”?

History: why 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 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.

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.

Elaborating the definition

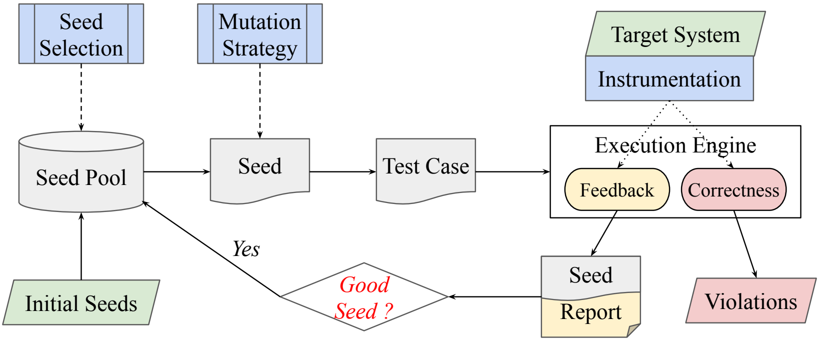
- What is special about the target system?
  - Do we know the source code?
  - Do we know the input format?
  - What are the challenges when executing the "system"?
- What do we mean by a state?
  - How can we tell that one state is different from another?
- What do we mean by desired?
  - New/unseen 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

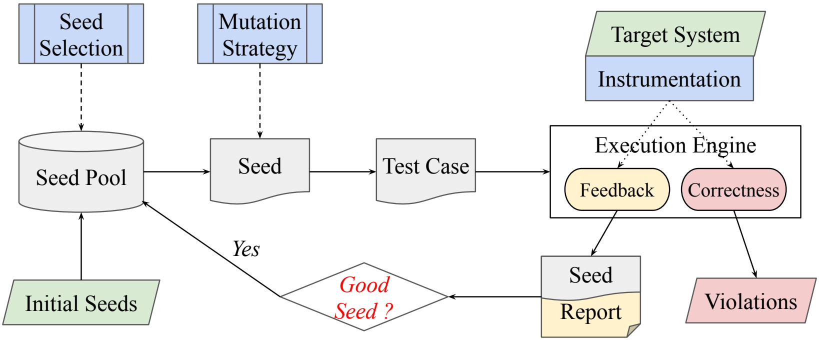
Genetic algorithm

Training a program to play the snake game with genetic algorithm

Feedback-guided evolution process



Feedback-guided evolution process



Natural selection — survival of the fittest

Demo with AFL++

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

Outline

Intuition: what makes a high-quality seed?

Intuition: what makes a high-quality seed?

```
pub fn foo(a: num, b: num) {
    let c = if (a >= 0) {
        1
    } else {
        2
    };

    // irrelevant operations

    let d = if (b >= 0) {
        2
    } else {
        3
    };

    // irrelevant operations

    assert!(c != d);
}
```

Q: What is the testing plan?

Intuition: what makes a high-quality seed?

```
pub fn foo(a: num, b: num) {
    let c = if (a >= 0) {
        1
    } else {
        2
    };

    // irrelevant operations

    let d = if (b >= 0) {
        2
    } else {
        3
    };

    // irrelevant operations

    assert!(c != d);
}
```

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?

```
pub fn foo(a: num, b: num) {
    let c = if (a >= 0) {
        1
    } else {
        2
    };

    // irrelevant operations

    let d = if (b >= 0) {
        2
    } else {
        3
    };

    // irrelevant operations

    assert!(c != d);
}
```

Q: What is the testing plan?

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

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

Illustration of different coverage metrics

```
pub fn foo(a: num, b: num) {
    let c = if (a >= 0) {
        1
    } else {
        2
    };

    // irrelevant operations

    let d = if (b >= 0) {
        2
    } else {
        3
    };

    // irrelevant operations

    assert!(c != d);
}
```

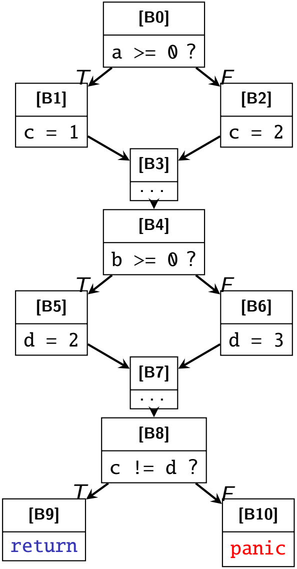


Illustration of different coverage metrics

- Cover every line?
  - Block coverage
- Cover every if-else branch?
  - Branch coverage
- Cover every exit status?
  - Return coverage
- Cover every path?
  - Path coverage

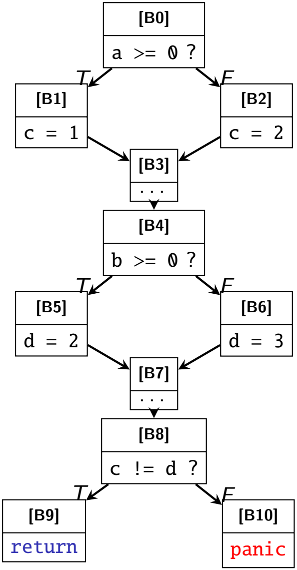
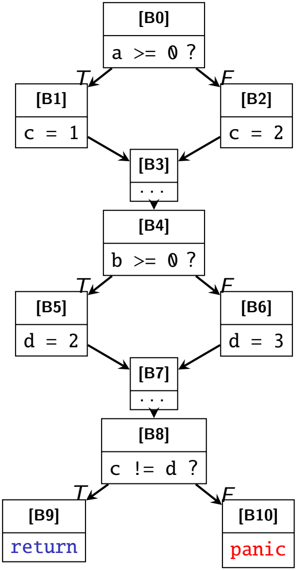


Illustration of different coverage metrics

- Cover every line?
  - Block coverage
- Cover every if-else branch?
  - Branch coverage
- Cover every exit status?
  - Return coverage
- Cover every path?
  - Path coverage

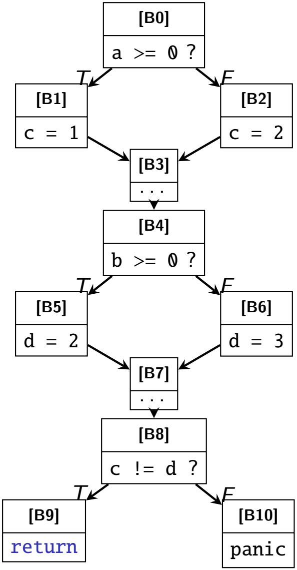


Path coverage: a theoretical optimum

Claim: A program is saturately tested if we obtain a set of inputs that covers every feasible path of the program CFG.

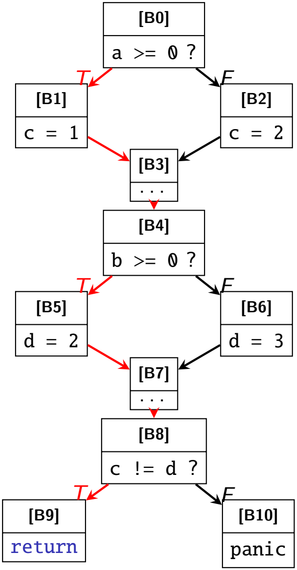
NOTE: feasible paths include paths that leads to explicit and implicit panics.

Path coverage demo



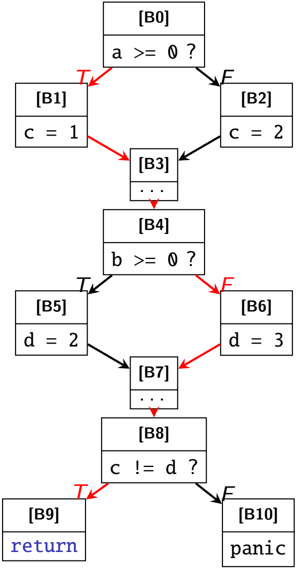
Path coverage demo

- a = 1, b = 1



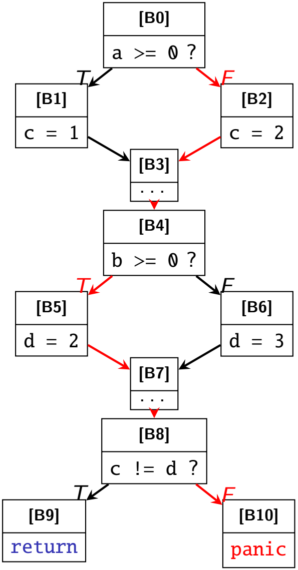
Path coverage demo

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



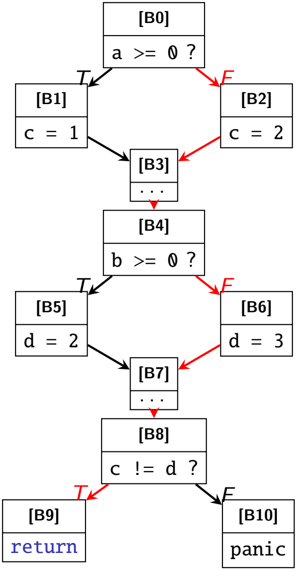
Path coverage demo

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



Path coverage demo

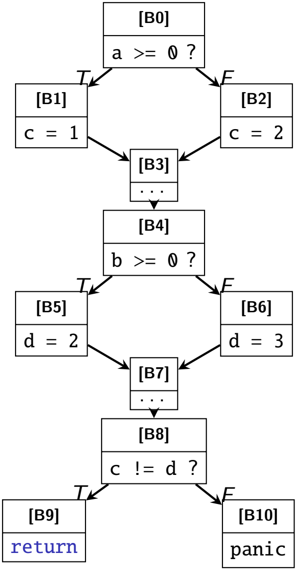
- a = 1, b = 1
- a = 1, b = -1
- a = -1, b = 1
- a = -1, b = -1



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 $\Longrightarrow$ the program is saturately tested



Why not path coverage in practice?

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.

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.

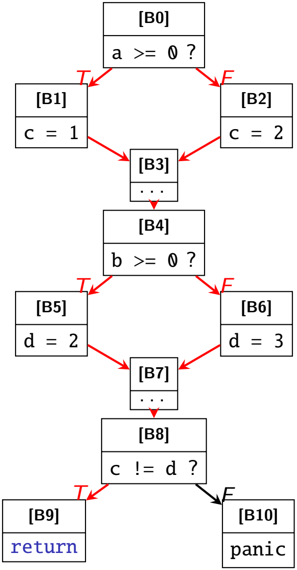
In practice, branch coverage hits a nice balance between effectiveness and easiness of instrumentation.

What's wrong with branch coverage?

What's wrong with branch coverage?

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

Two seeds already covered most of the branches.



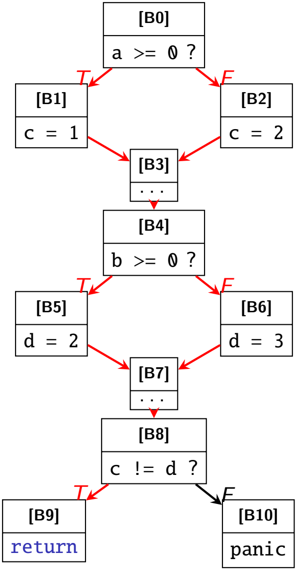
What's wrong with branch coverage?

- 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.



What's wrong with branch coverage?

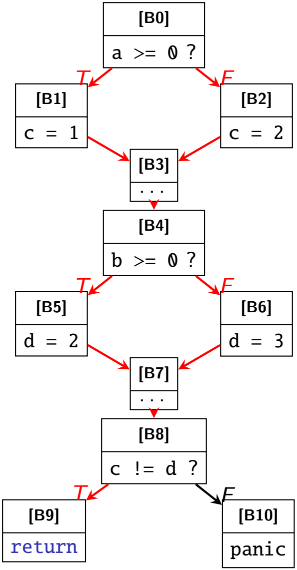
- `a = 1, b = 1`
- `a = -1, b = -1`

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,



What's wrong with branch coverage?

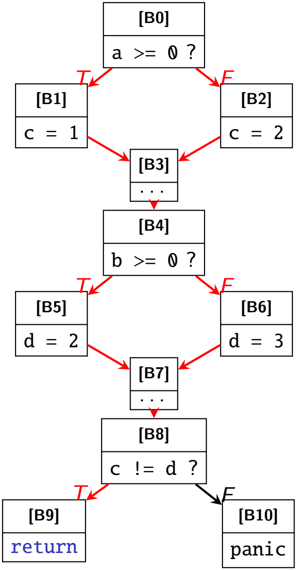
- `a = 1, b = 1`
- `a = -1, b = -1`

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.

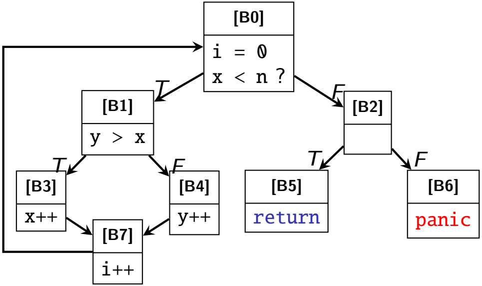
$\Longrightarrow$ fuzzer is not rewarded by mutating `a` and `b`, hence, lowering their priorities and the panic case may never be found, especially when fuzzing complex CFGs



Outline

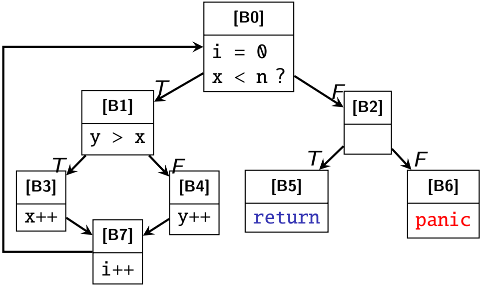
Looping example

```
pub fn looping(
  x: num,
  y: num,
  n: num
) {
  let i = 0;
  while (x < n) {
    if (y > x) {
      x++;
    }
    else {
      y++;
    }
    i++;
  }
  assert!(i != 7);
}
```



Looping example

```
pub fn looping(
  x: num,
  y: num,
  n: num
) {
  let i = 0;
  while (x < n) {
    if (y > x) {
      x++;
    }
    else {
      y++;
    }
    i++;
  }
  assert!(i != 7);
}
```

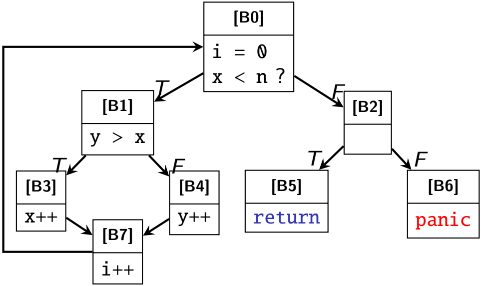


y <= x < n

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

Looping example

```
pub fn looping(
  x: num,
  y: num,
  n: num
) {
  let i = 0;
  while (x < n) {
    if (y > x) {
      x++;
    }
    else {
      y++;
    }
    i++;
  }
  assert!(i != 7);
}
```



`y <= x < n`

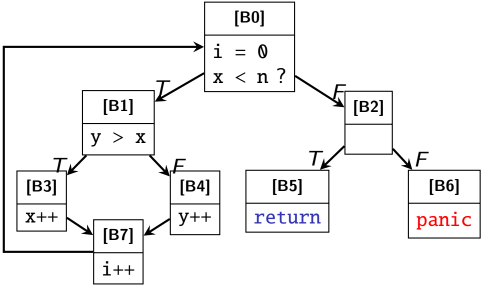
1. `y++` until `y == x`
2. `y++; x++` until `x == n`

`x < y <= n`

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

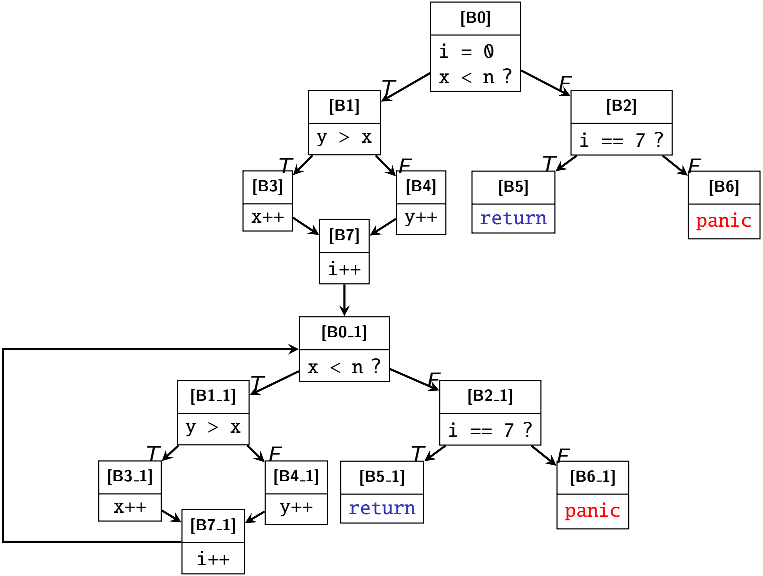
Looping example

```
pub fn looping(
  x: num,
  y: num,
  n: num
) {
  let i = 0;
  while (x < n) {
    if (y > x) {
      x++;
    }
    else {
      y++;
    }
    i++;
  }
  assert!(i != 7);
}
```

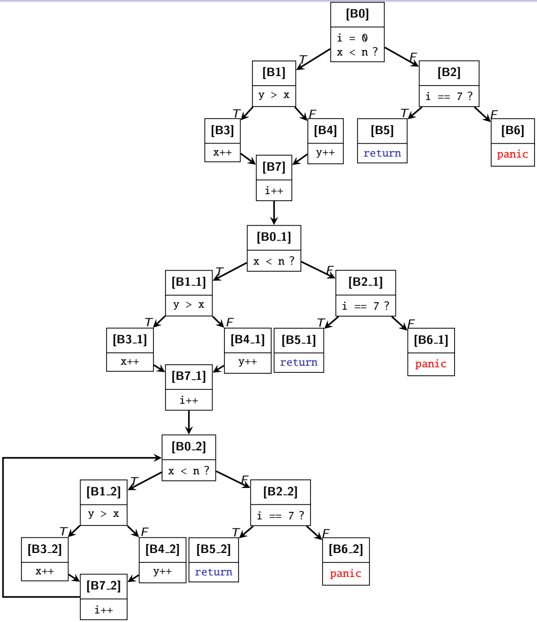


y <= x < n	x < y <= n	x < n <= y
1 y++ until y == x	1 x++ until x == y	1 x++ until x == n
2 y++; x++ until x == n	2 y++; x++ until x == n	

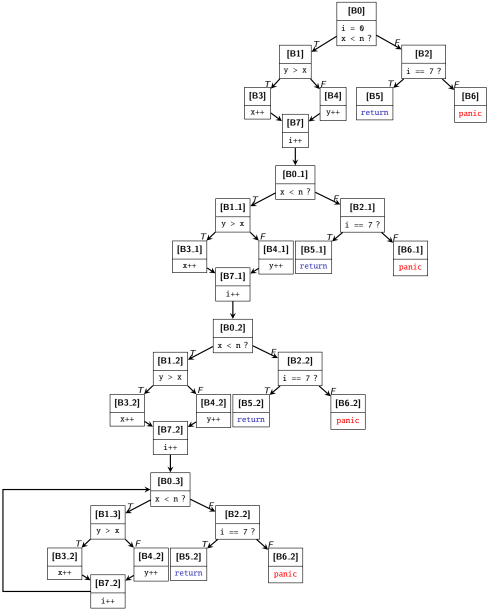
Solution: bounded loop unrolling



Solution: bounded loop unrolling



Solution: bounded loop unrolling



Outline

Narrow-range constraints

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

```
fn foo(x: u64, y: u64) {
  if x + y = 42 {
    panic!();
  }
}
```

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:

```
fn foo(x: u64, y: u64) {
  if x + y = 42 {
    panic!();
  }
}
```

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.

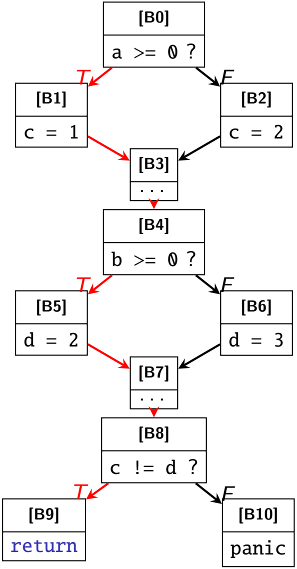
The general intuition behind concolic execution

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.

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.

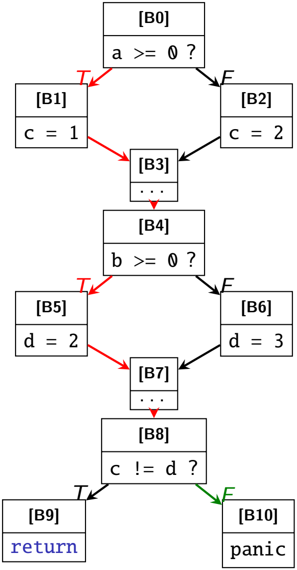


Concolic execution demo

- `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



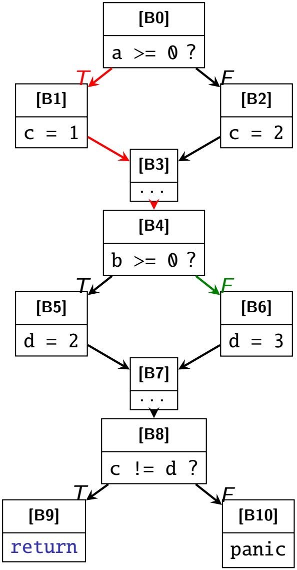
Concolic execution demo

- `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, $a = 1,\ b = -1$



Concolic execution demo

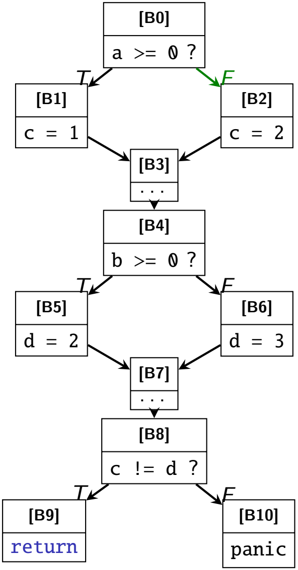
- a = 1, b = 1

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, $a = -1$



Outline

A comprehensive survey of current works

Fuzzing Family Tree

$\langle$ **End** $\rangle$