CS 453/698: Software and Systems Security

Module: Bug Finding Tools and Practices
Lecture: Fuzz testing (a.k.a., fuzzing)

Meng Xu *(University of Waterloo)*

Winter 2025

Outline

Program assurance

An eternal question behind all software developers' mind:
how do I know that my code is correct and secure?

Program assurance

An eternal question behind all software developers' mind:
how do I know that my code is correct and secure?

- Existing practice: testing with manual effort
  - a.k.a., unit tests, E2E tests, quality assurance, etc.

Program assurance

An eternal question behind all software developers' mind:
how do I know that my code is correct and secure?

- Existing practice: testing with manual effort
  - a.k.a., unit tests, E2E tests, quality assurance, etc.
- Emerging trend in practice: coverage-guided fuzzing
  - i.e., automated, evolutionary, and random generation of test cases

Program assurance

An eternal question behind all software developers' mind:
how do I know that my code is correct and secure?

- Existing practice: testing with manual effort
  - a.k.a., unit tests, E2E tests, quality assurance, etc.
- Emerging trend in practice: coverage-guided fuzzing
  - i.e., automated, evolutionary, and random generation of test cases
- In research pipeline: symbolic execution
  - i.e., automated, systematic, and deterministic exploration of search space

Program assurance

An eternal question behind all software developers' mind:
how do I know that my code is correct and secure?

- Existing practice: testing with manual effort
  - a.k.a., unit tests, E2E tests, quality assurance, etc.
- Emerging trend in practice: coverage-guided fuzzing
  - i.e., automated, evolutionary, and random generation of test cases
- In research pipeline: symbolic execution
  - i.e., automated, systematic, and deterministic exploration of search space
- Latest development: **concolic execution**
  - i.e., automated, efficient, and practical exploration of search space

Running example

Running example

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
    /*
     *
     * code omitted for now
     *
     */
}
```

calc is the function we want to check for vulnerabilities.

Running example

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
    /*
     *
     * code omitted for now
     *
     */
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  // use the results
  // ......
}
```

calc is the function we want to check for vulnerabilities.

We know how to use calc (as shown in the main function).

Running example

```
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
    /*
     *
     * code omitted for now
     *
     */
}
```

```
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

calc is the function we want to check for vulnerabilities.

We know how to use calc (as shown in the `main` function).

More importantly, we know when calc is wrong (by the assertion the `main` function).

Running example

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
    /*
     *
     * code omitted for now
     *
     */
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

calc is the function we want to check for vulnerabilities.

We know how to use calc (as shown in the `main` function).

More importantly, we know when calc is wrong (by the assertion the `main` function).

Real-world software is of course more complicated than this simple example.

Running example: open-box analysis

Running example: open-box analysis

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

Q: Now we have the source code of the `calc` function, how do we know it is correct?

Running example: open-box analysis

```
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

Q: Now we have the source code of the calc function, how do we know it is correct?

A: Just try it with some concrete inputs and see if the results match with our expectations.

Outline

Testing the running example

```
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

Testing the running example

```
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

- x=0, y=1, n=2 → a=2, b=2, i=3

Testing the running example

```
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

- x=0, y=1, n=2 → a=2, b=2, i=3
- x=1, y=0, n=2 → a=2, b=2, i=3

Testing the running example

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

- x=0, y=1, n=2 $\rightarrow$ a=2, b=2, i=3
- x=1, y=0, n=2 $\rightarrow$ a=2, b=2, i=3
- x=0, y=2, n=1 $\rightarrow$ a=1, b=2, i=1
- x=1, y=2, n=0 $\rightarrow$ a=1, b=2, i=0

Testing the running example

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

- x=0, y=1, n=2 → a=2, b=2, i=3
- x=1, y=0, n=2 → a=2, b=2, i=3
- x=0, y=2, n=1 → a=1, b=2, i=1
- x=1, y=2, n=0 → a=1, b=2, i=0
- x=2, y=0, n=1 → a=2, b=0, i=0
- x=2, y=1, n=0 → a=2, b=1, i=0
- ......

Testing the running example

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

- x=0, y=1, n=2 → a=2, b=2, i=3
- x=1, y=0, n=2 → a=2, b=2, i=3
- x=0, y=2, n=1 → a=1, b=2, i=1
- x=1, y=2, n=0 → a=1, b=2, i=0
- x=2, y=0, n=1 → a=2, b=0, i=0
- x=2, y=1, n=0 → a=2, b=1, i=0
- ......

Q: When should the testing end?

Testing the running example

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

- x=0, y=1, n=2 → a=2, b=2, i=3
- x=1, y=0, n=2 → a=2, b=2, i=3
- x=0, y=2, n=1 → a=1, b=2, i=1
- x=1, y=2, n=0 → a=1, b=2, i=0
- x=2, y=0, n=1 → a=2, b=0, i=0
- x=2, y=1, n=0 → a=2, b=1, i=0
- ......

Q: When should the testing end?

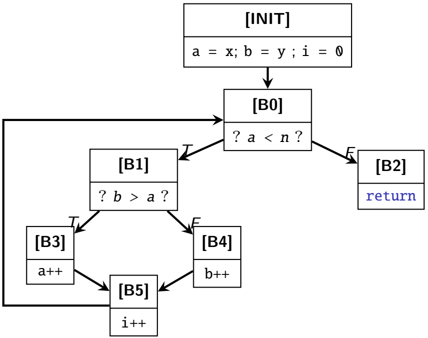
A: The *de facto* answer is: when achieved 100% code coverage.

CFG and code coverage

```
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

Figure: the control-flow graph (CFG) of function `calc(..)`

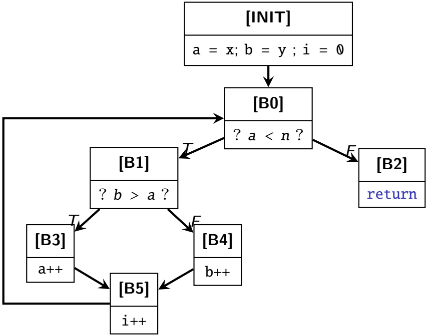


CFG and code coverage

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

Figure: the control-flow graph (CFG) of function `calc(..)`



100% code coverage usually means:

- all nodes in the CFG, or
- all edges in the CFG

100% coverage does not imply a worry-free program

```rust
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

```rust
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

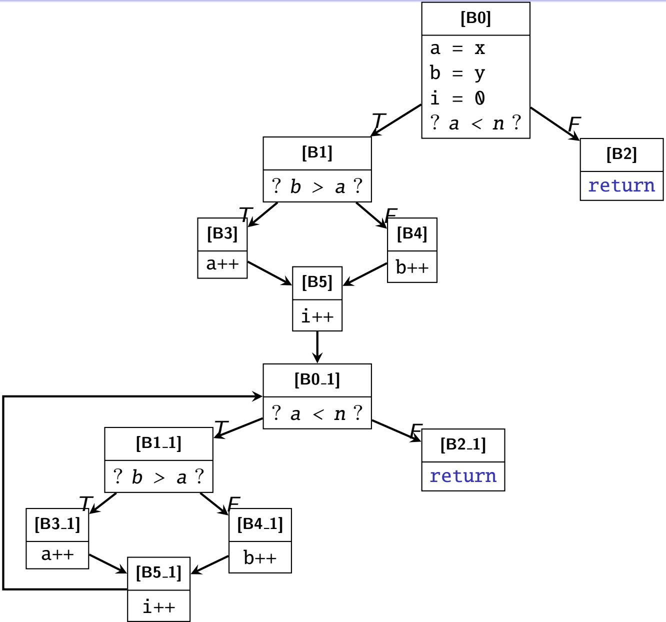
100% coverage does not imply a worry-free program

```
// implementation of `calc`
fn calc(
  x: u64, y: u64, n: u64
) -> (u64, u64, u64) {
  let a = x, b = y, i = 0;
  while (a < n) {
    if (b > a) {
      a++;
    } else {
      b++;
    }
    i++;
  }
  return (a, b, i);
}
```

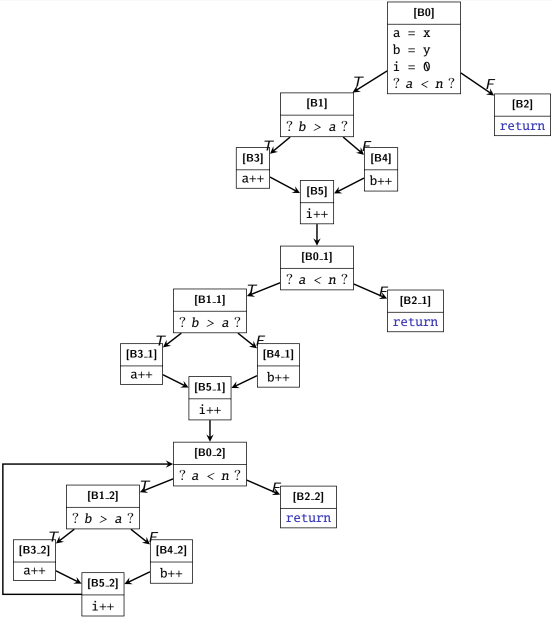
```
// use the `calc` function
pub fn main() {
  let (x, y, n) = /* input */;
  let (a, b, i) = calc(x, y, n);
  assert!(n-a-b+i != 42);

}
```

- $x=0$, $y=1$, $n=2 \rightarrow a=2$, $b=2$, $i=3$
- $x=1$, $y=0$, $n=2 \rightarrow a=2$, $b=2$, $i=3$
- $x=0$, $y=2$, $n=1 \rightarrow a=1$, $b=2$, $i=1$
- $x=1$, $y=2$, $n=0 \rightarrow a=1$, $b=2$, $i=0$
- $x=2$, $y=0$, $n=1 \rightarrow a=2$, $b=0$, $i=0$
- $x=2$, $y=1$, $n=0 \rightarrow a=2$, $b=1$, $i=0$

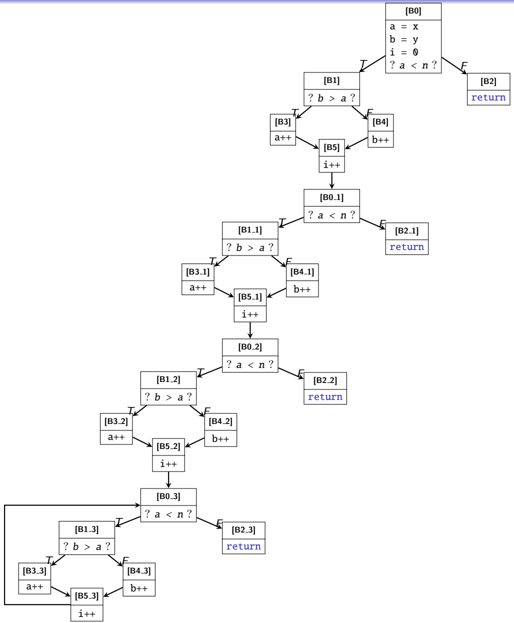
Reason: loop unrolling yields new components in CFG



Reason: loop unrolling yields new components in CFG



Reason: loop unrolling yields new components in CFG



Manual enumeration is impossible

In reality, it is infeasible to manually construct test cases that cover every new node / edge in the loop-unrolled CFG.

Manual enumeration is impossible

In reality, it is infeasible to manually construct test cases that cover every new node / edge in the loop-unrolled CFG.

A genetic programming solution: coverage-guided fuzzing.

Manual enumeration is impossible

In reality, it is infeasible to manually construct test cases that cover every new node / edge in the loop-unrolled CFG.

A genetic programming solution: coverage-guided fuzzing.

1. Randomly selects a seed s from a pool of test cases (seed pool)
   - e.g., seed test case $s = \{\ \mathtt{x}{=}0,\ \mathtt{y}{=}1,\ \mathtt{n}{=}2\ \}$
2. Mutate seed s to produce a new test case t
   - e.g., test case $t = \{\ \mathtt{x}{=}0,\ \mathtt{y}{=}42,\ \mathtt{n}{=}2\ \}$
3. Execute t and collect coverage on the CFG during execution
   - if t yields new coverage, save t to the seed pool
   - otherwise, discard t
4. Go back to step 1 — completing one round of evolution

Outline

1. Introduction
2. Testing with concrete inputs
3. Fuzzing overview
4. Program state coverage: “natural selection” in the fuzzing world
5. Conclusion

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.

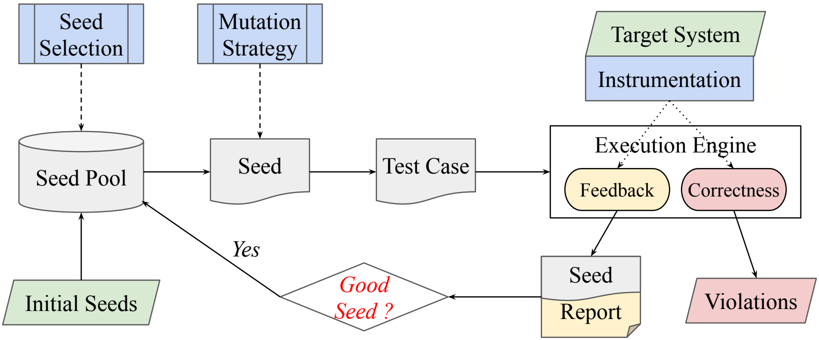
Evolution: from the rain-fuzzer to modern fuzzing

Evolution: from the rain-fuzzer to modern fuzzing

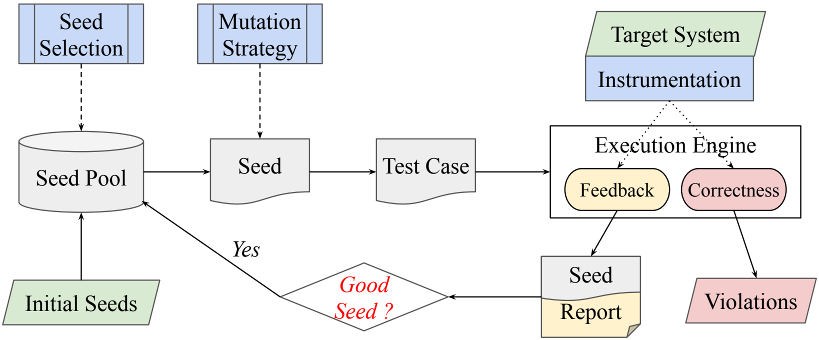
The key is **genetic algorithm**.

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?

$\implies$ 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);
}
```

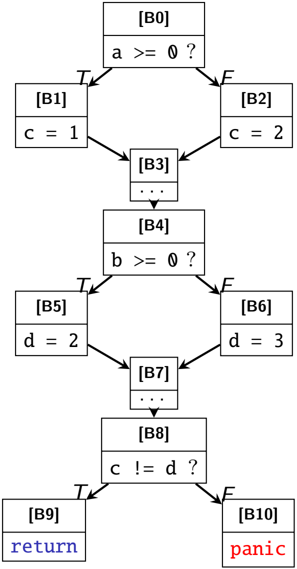


Illustration of different coverage metrics

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

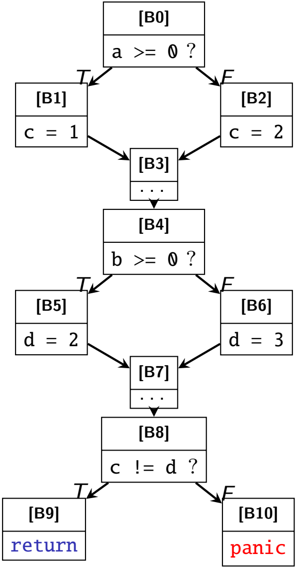
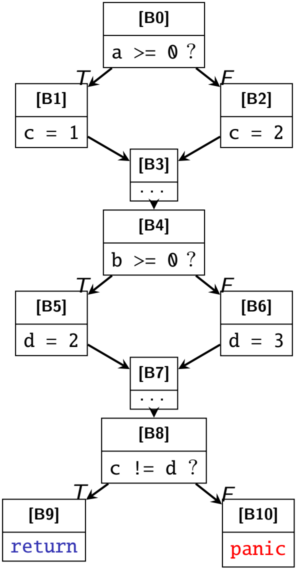


Illustration of different coverage metrics

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

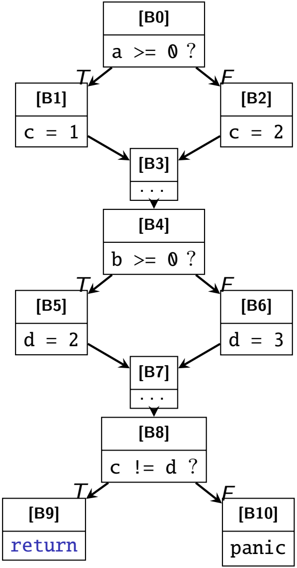


Path coverage: a theoretical optimum

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

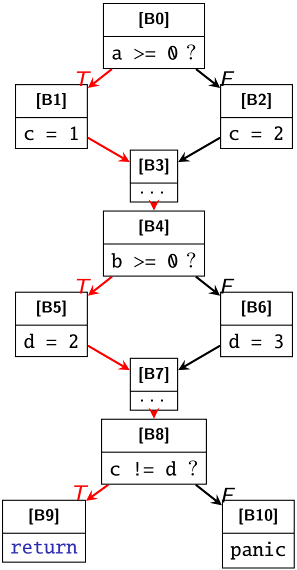
NOTE: feasible paths include paths that leads to explicit and implicit panics.

Path coverage demo



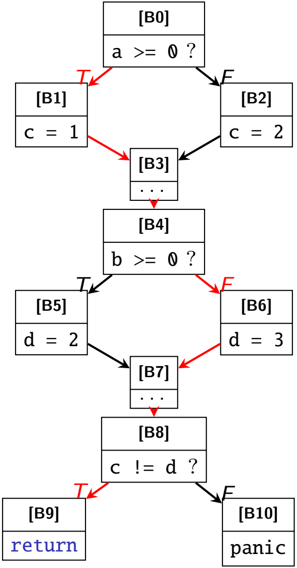
Path coverage demo

- `a = 1, b = 1`



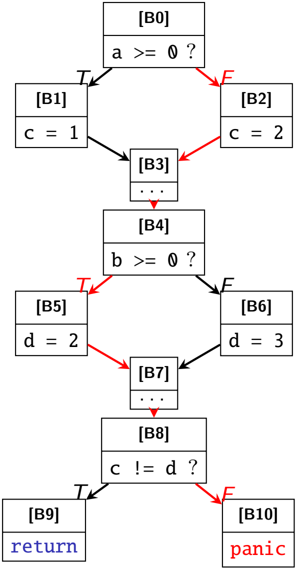
Path coverage demo

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



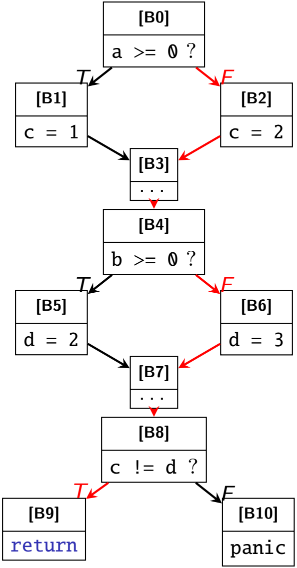
Path coverage demo

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



Path coverage demo

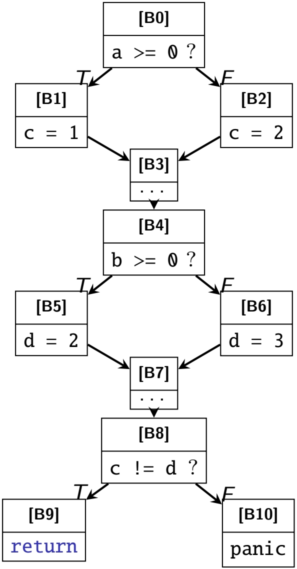
- a = 1, b = 1
- a = 1, b = -1
- a = -1, b = 1
- a = -1, b = -1



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 $\implies$ 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.

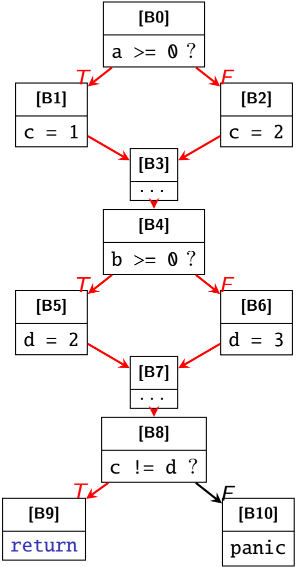
In practice, branch coverage hits a nice balance between effectiveness and easiness of instrumentation.

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.



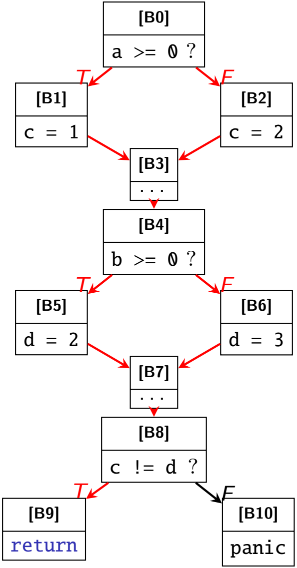
What's wrong with branch coverage?

- `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?

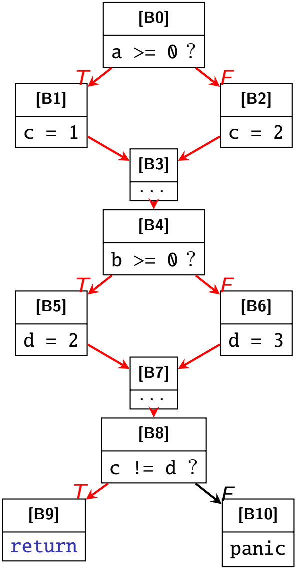
- `a = 1, b = 1`
- `a = -1, b = -1`

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.

$\Longrightarrow$ 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?

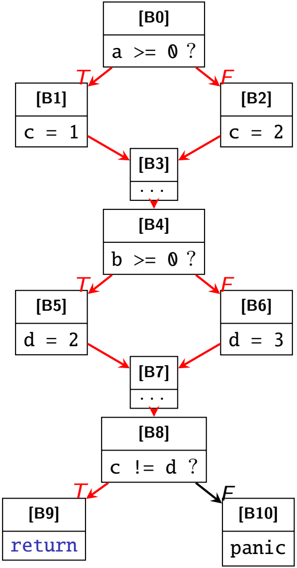
- `a = 1, b = 1`
- `a = -1, b = -1`

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

1. Introduction
2. Testing with concrete inputs
3. Fuzzing overview
4. Program state coverage: “natural selection” in the fuzzing world
5. 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.

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?

⟨ End ⟩