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”? 

In the 80s, 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 noises 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.
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.
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

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
Genetic algorithm

Training a program to play the snake game with genetic algorithm
Feedback-guided evolution process

- Seed Pool
- Seed Selection
- Mutation Strategy
- Seed
- Test Case
- Target System
- Instrumentation
- Execution Engine
  - Feedback
  - Correctness
- Initial Seeds
- Good Seed?
- Yes
- Seed Report
- Violations
- Good Seed?
Feedback-guided evolution process

- Seed Pool
- Mutation Strategy
- Target System
- Instrumentation
- Execution Engine
- Seed
- Good Seed?
- Violations
- Seed Report
- Correctness
- Feedback
- Initial Seeds

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

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
Intuition: what makes a high-quality seed?
Intuition: what makes a high-quality seed?

Q: What is the testing plan?

```rust
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);
}
```
Intuition: what makes a high-quality seed?

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?

```rust
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?

⇒ if the fuzzer generates an input that expands the coverage, that input is a good seed.
Illustration of different coverage metrics

```rust
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 lead to explicit and implicit panics.
Path coverage demo

\[ a = 1,\quad b = 1 \]
\[ a = 1,\quad b = -1 \]
\[ a = -1,\quad b = 1 \]
\[ a = -1,\quad b = -1 \]

No new program behaviors can be discovered \( \Rightarrow \) the program is saturately tested

\[
\begin{align*}
[B0] & \quad a \geq 0 \ ? \\
[B1] & \quad c = 1 \\
[B2] & \quad c = 2 \\
[B4] & \quad b \geq 0 \ ? \\
[B5] & \quad d = 2 \\
[B6] & \quad d = 3 \\
[B8] & \quad c \neq d \ ?
\end{align*}
\]

\[ T \quad F \quad T \quad F \quad T \quad F \quad T \quad F \]

15 / 29
Path coverage demo

- a = 1, b = 1
Path coverage demo

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

\[ a \geq 0 \quad ? \\
[B1] \\
c = 1 \quad \rightarrow \\
[B2] \\
c = 2 \quad \rightarrow \\
[B3] \\
\ldots \quad \rightarrow \\
[B4] \\
b \geq 0 \quad ? \\
[B5] \\
d = 2 \quad \rightarrow \\
[B6] \\
d = 3 \quad \rightarrow \\
[B7] \\
\ldots \quad \rightarrow \\
[B8] \\
c \neq d \quad ? \\
[B9] \\
return \quad \rightarrow \\
[B10] \\
panic

\[ T \quad F \quad T \quad F \quad T \quad F \quad T \quad F \quad 15 / 29 \]
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?

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.
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?

a = 1, b = 1
a = -1, b = -1
Two seeds already covered most of the branches.

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

\[ a \geq 0 \rightarrow \] \[ b \geq 0 \rightarrow \] 
\[ c = 1 \]
\[ c = 2 \]
\[ \ldots \]
\[ b \neq c \rightarrow \]
\[ \text{return} \]

T
F
T
F
T
F
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.

\[\Rightarrow\] 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
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
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     }  
15     i++;  
16     assert!(i != 7);  
17 }
Looping example

```rust
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);
}
```

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.    } else {
11.        y++;
12.    }
13.    i++;
14. }
15. `assert!(i != 7);`
16. `}`

1. `y <= x < n`
2. `y++ until y == x`
3. `y++; x++ until x == n`
Looping example

```rust
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);
}
```

The diagram illustrates the control flow of the `looping` function. The loop condition `(x < n)` is checked, and the function increments `i` if `y > x`. If `y <= x`, the function increments `y` and continues the loop. The loop ends when `i` is not equal to 7, and an assertion is made to ensure the loop termination condition is met.
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         } else {
11             y++;
12         }
13         i++;
14     }
15     assert!(i != 7);
16 }

17

y <= x < n
17 y++ until y == x
18 y++; x++ until x == n
x < y <= n
18 x++ until x == y
19 y++; x++ until x == n
x < n <= y
19 x++ until x == n

y > x
[B0] i = 0
x < n ?
[F]

x++
[B3]
T

F

y++
[B4]

[B5] return

panic
[B6]

T

F

i++
[B7]
Solution: bounded loop unrolling

```
i = 0
x < n ?

y > x

i == 7 ?

x++
y++
return

panic
```

```
i++

x < n ?

y > x

i == 7 ?

x++
y++
return

panic
```
Solution: bounded loop unrolling
Solution: bounded loop unrolling

```
[B0] i = 0
    x < n?
        [B1] y > x
            [B3] x++
            [B4] y++
            [B7] i++
                [B0.1] x < n?
                    [B1.1] y > x
                        [B3.1] x++
                        [B4.1] y++
                        [B7.1] i++
                            [B0.2] x < n?
                                [B1.2] y > x
                                    [B3.2] x++
                                    [B4.2] y++
                                    [B7.2] i++
                                        [B0.3] x < n?
                                            [B1.3] y > x
                                                [B3.3] x++
                                                [B4.3] y++
                                                [B7.3] i++
                                                    [B0.4] x < n?
                                                        [B1.4] y > x
                                                            [B3.4] x++
                                                            [B4.4] y++
                                                            [B7.4] i++
```

"panic"
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
Narrow-range constraints

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

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

```rust
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 most 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.
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 \(a \geq 0 \land b \geq 0\) and the program, can we toggle \(c \neq d\)?

\[\Rightarrow \text{unsat, infeasible path}\]
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 $a \geq 0 \land b \geq 0$ 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, $a = 1, b = -1$
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 \( a \geq 0 \land b \geq 0 \) and the program, can we toggle \( c \neq d \)? 
\[ \implies \text{unsat, infeasible path} \]

**Query 2**: given constraint \( a \geq 0 \) and the program, can we toggle \( b \geq 0 \)? 
\[ \implies \text{sat, } a = 1, b = -1 \]

**Query 3**: given constraint \( \text{true} \) and the program, can we toggle \( a \geq 0 \)? 
\[ \implies \text{sat, } a = -1 \]
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
A comprehensive survey of current works

Fuzzing Family Tree
〈 End 〉