COMP 587: AFL and KLEE

Kyle Dewey

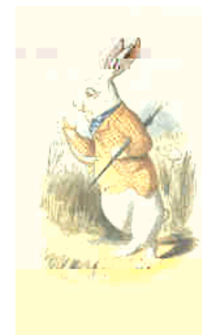
Fuzzing Approaches

- Generation-based: generate whole test inputs from scratch (what we've been doing)
- Mutation-based: generate new test inputs by modifying old ones
- Can do both, even simultaneously

AFL

- Very popular fully-automated fuzzer
- Mutation-based: make new tests by tweaking existing tests
- Tell it where the input is, and it does the rest

AFL - Basic Idea



-Flip bits
-Rearrange bits / bytes
-Randomly inject bits
-Look at code coverage information while this is happening to see if tests are getting into new areas - tests that hit new areas are selected more frequently for mutation

AFL Demo

AFL Highlights

- Fast
- Easy to use
- Needs a *seed corpus* (starting set of tests) to start mutations from
  - Has major impact on performance
- Good, but tends to plateau quickly compared to specialized fuzzers

Symbolic Execution
(Towards KLEE)

Symbolic Execution

- Basic idea: remember the conditions that led you to your current place in the code
  - This is called the *path condition*

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Example

Path Condition

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

`x > 0`

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

```
x > 0 &&
 y > 10
```

-So to print "a", x must be > 0 and y must be > 10

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

`x > 0`

-if, however, we went down the false branch...

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

```
x > 0 &&
!(y > 10)
```

–...then to print "b", it must be that x > 10 and NOT y > 10

Example

Path Condition

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

-Going back to the beginning, if the condition was false...

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

```
!(x > 0)
```

–...then it must be the case that x is not > 0

Concolic Execution

- Combines *concrete* (normal) execution and symbolic execution
- Basic idea: use the path condition to discover test inputs which explore different program paths

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

Tests

`x = 1, y = 1`

–Randomly choose inputs of x = 1 and y = 1

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

Tests

`x = 1, y = 1`

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

Tests

`x = 1, y = 1`

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

`x > 0`

Tests

`x = 1, y = 1`

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

`x > 0`

`x > 0` goes down this path, so `!(x > 0)` goes down another path

Tests

`x = 1, y = 1`

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

`x > 0`

`x > 0` goes down this path, so `!(x > 0)` goes down another path

Tests

`x = 1, y = 1`

`x = 0, y = 1`

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

```
x > 0 &&
!(y > 10)
```

Tests

```
x = 1, y = 1
x = 0, y = 1
```

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

`x > 0 && !(y > 10)`

To go down another path, need

`x > 0 && y > 10`

Tests

`x = 1, y = 1`

`x = 0, y = 1`

Example

```
def foo(x, y):
  if x > 0:
    if y > 10:
      print("a")
    else:
      print("b")
  else:
    print("c")
```

Path Condition

```
x > 0 &&
!(y > 10)
```

To go down another path, need

```
x > 0 && y > 10
```

Tests

```
x = 1, y = 1
x = 0, y = 1
x = 1, y = 11
```

Basic Idea

- Negate parts of the path condition to discover different program paths
- Find inputs which satisfy these negated paths to generate new test inputs
- Keep running generated test inputs and continue this process until all paths are explored

Finding Inputs Satisfying Constraints

- This is what SMT solvers do
- Best-case NP-Complete, worst-case undecidable
- Usually surprisingly fast in practice

`x > 0 && y > 10` → `x = 1, y = 11`

KLEE

- Tool which performs concolic execution
- Has a custom SMT solver internally for doing this quickly (STP)
- Been used to find bugs in tons of systems, including the Linux kernel

KLEE Demo

Concolic Execution Downside

Explores all program paths...whether you want to or not.

Concolic Execution Downside

Explores all program paths...whether you want to or not.

```
def bar(x):
  while x > 10:
    x = x - 1
```

Path Condition

Concolic Execution Downside

Explores all program paths...whether you want to or not.

```
def bar(x):
  while x > 10:
    x = x - 1
```

Path Condition

`!(x > 10)`

Tests

`x > 10; x = 11`

Concolic Execution Downside

Explores all program paths...whether you want to or not.

```
def bar(x):
  while x > 10:
    x = x - 1
```

Path Condition

Tests

`x > 10; x = 11`

Concolic Execution Downside

Explores all program paths...whether you want to or not.

```
def bar(x):
  while x > 10:
    x = x - 1
```

Path Condition

`x0 > 10`

Tests

`x > 10; x = 11`

Concolic Execution Downside

Explores all program paths...whether you want to or not.

```
def bar(x):
  while x > 10:
    x = x - 1
```

Path Condition

`x0 > 10`

Tests

`x > 10; x = 11`

Concolic Execution Downside

Explores all program paths...whether you want to or not.

```
def bar(x):
  while x > 10:
    x = x - 1
```

Path Condition

`x0 > 10 &&`
`!(x1 > 10)`

Tests

`x > 10; x = 11`

-Key point: each iteration introduces a new variable into the path constraint
-There are ways around this in specific cases, but loops can trip up symbolic execution systems

Concolic Execution Overall

- Great for code dealing with specific conditions which are unlikely to hit otherwise
- Can get tripped up on loops