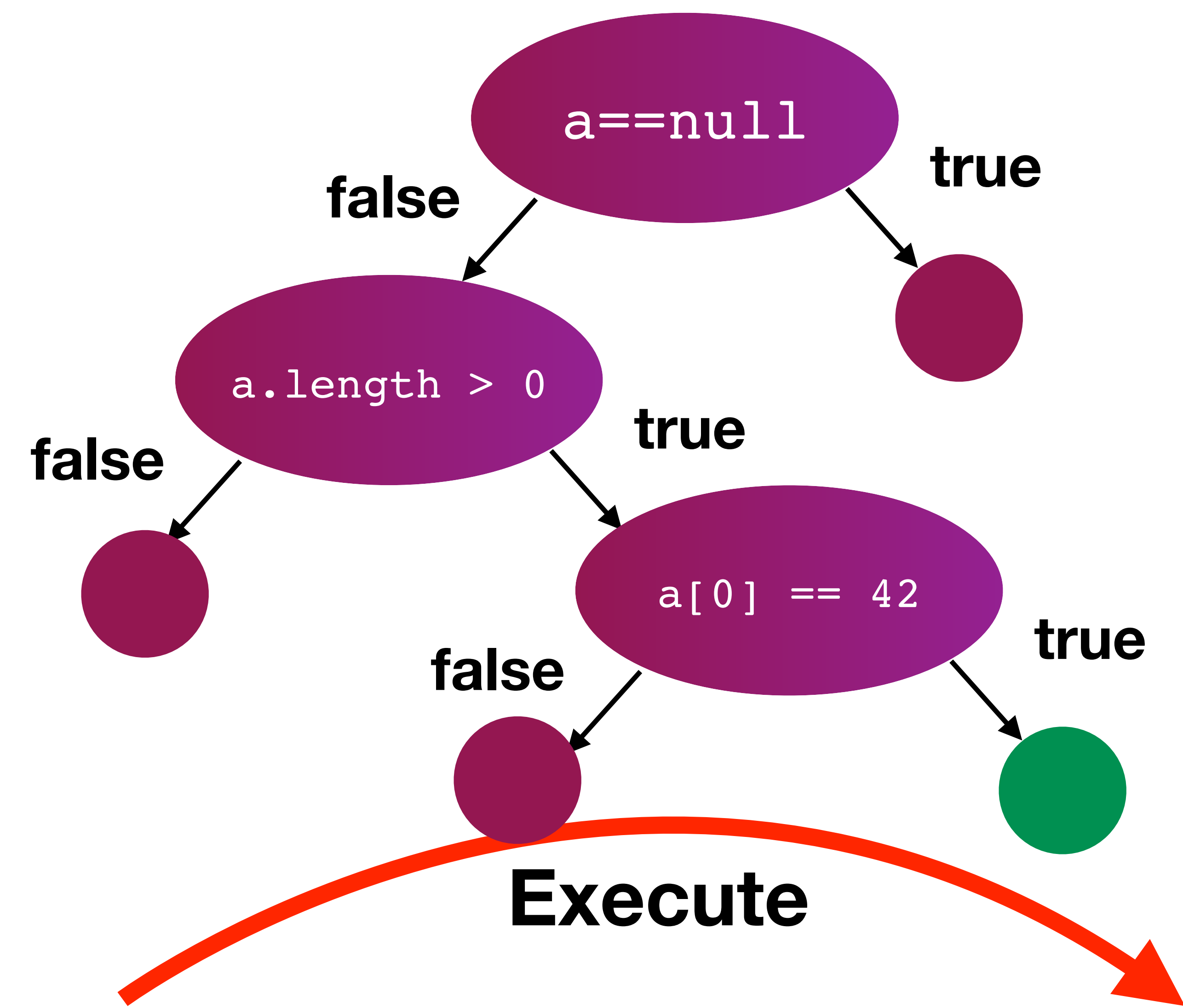
Lightweghted Concolic Execution Hands-on

CS453 Automated Software Testing

Shin Yoo

```
void testme(int[] a)
{
  if(a == null) return;
  if(a.length > 0)
  {
   if(a[0] == 42)
    throw new Exception("bug");
  }
}
```



Solve

Execute

Constraints to Solve	Data	Observed Path Condition
	null	a==null
a!=null	{}	a!=null && !(a.length > 0)
a!=null && a.length > 0	{0}	a!=null && a.length > 0 && a[0] != 42
a!=null && a.length > 0 && a[0] == 42	{42}	No more path!

Negate last condition and choose another path

Let's build a small concolic engine

…for a small subset of Python

- We will handle a single function only
- The function only takes integers
- Only if statements, no loops
- Only a single comparison between arguments
- No assignments, no local variables

```
def foo(x, y):
 if x - y > 5:
  print("foo")
  if x == y * 10 + 2:
   print("bar")
  else:
   print("zoo")
```

Primer: Z3 SMT Solver

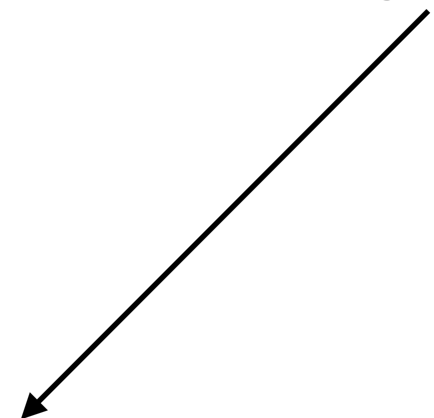
- Install the solver
  - `$ brew install z3`
  - `$ sudo apt install z3`
- Install the python wrapper
  - `$ pip install z3-solver`

```python
from z3 import *

x = Int("x")
y = Int("y")

s = Solver()

s.add(x + 45 < y)
s.add(x * 3 == y)

s.check()

print("x =", s.model()[x].as_long())
print("y =", s.model()[y].as_long())
```

Implementing Symbolic Execution

- You can modify existing interpreter or compiler.
- You can instrument the entire program, so that you rewrite and inject the symbolic layer
    - For example, change `x = y + 1` to `assign(x, add(y, 1))` and implement `assign()` and `add()` to handle the symbols
- You can (at least partially) exploit the dynamic dispatching of languages like Python 👈 (today)

Python does not really care whether x and y are int or not…



```
def foo(x, y):
  if x - y > 5:
    print("foo")
    if x == y * 10 + 2:
      print("bar")
    else:
      print("zoo")
```

What happens if we use a fake int?

- The fake int should carry the symbols.
- But it should also carry concrete values, because we want the vanilla Python code to run based on these fake int values.

```
class SymInt():
 def __init__(self, symb, conc):
  self.symbolic = symb
  self.concrete = conc
```

What about operations?

- We need to override magic methods so that symbolic semantic can be recorded.
    - `SymInt(“x”, 1) + SymInt(“y”, 4)` should be?
    - And this can be implemented where? 🤠

Where do we capture path constraints?

- Since we do not expect any loops, we can simply record whenever we actually compute a Boolean predicate.
  - Symbolic comparison
  - `__bool__`

Finally, the concolic algorithm

- We will not do any book-keeping of executed paths.
- Instead, let's record all solved path constraints - if we start repeating ourselves, it means that we have solved everything.