Driller: Augmenting Fuzzing through Symbolic Execution

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna

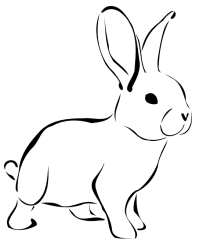
Motivation

- Large number of memory corruption bugs

- Problems with testcase generation techniques
  - Fuzzing
  - Symbolic Execution

Fuzzing

```
x = int(input())
if x > 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



Let's fuzz it!

1 ⇒ "You lose!"

593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

498 ⇒ "You lose!"

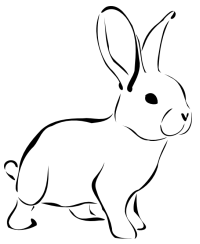
48 ⇒ "You win!"

Catching Bugs

- Monitors program for crashes

```
x = int(input())
if x > 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Let's fuzz it!



1 ⇒ "You lose!"

593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

498 ⇒ "You lose!"

42 ⇒ "You lose!"

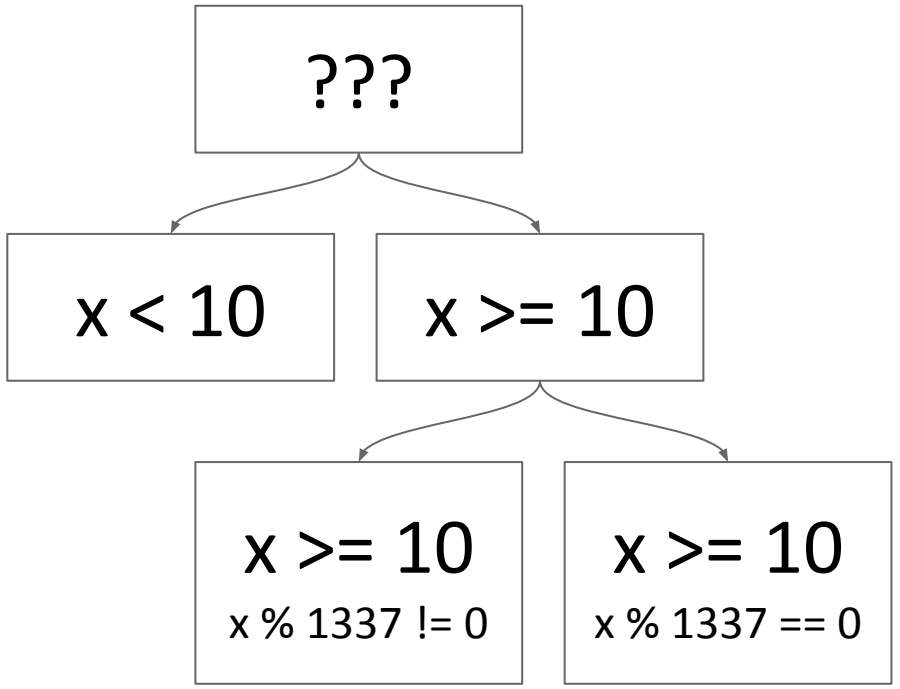
3 ⇒ "You lose!"

.....

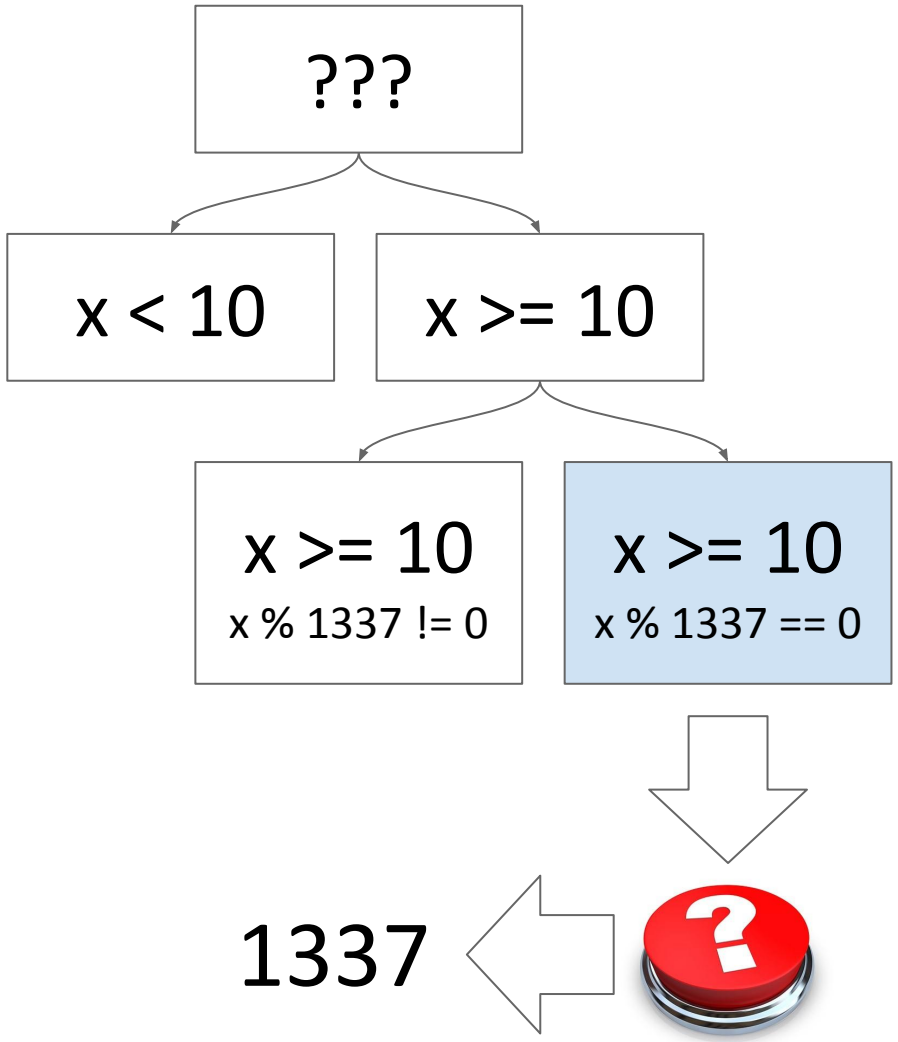
57 ⇒ "You lose!"

Symbolic Execution

```
x = input()
if x >= 10:
    if x % 1337 == 0:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



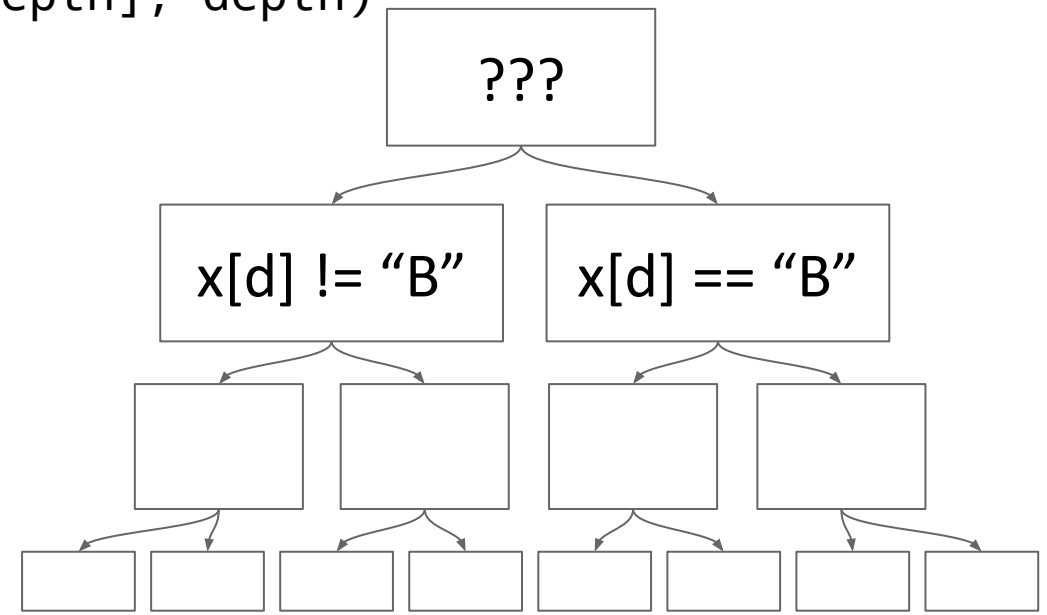
```
x = input()
if x >= 10:
    if x % 1337 == 0:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



Catching Bugs

- Checks each state for safety violations
  - symbolic program counter
  - writes/reads from symbolic address

```
x = input()

def recurse(x, depth):
  if depth == 2000
    return 0
  else {
    r = 0;
    if x[depth] == “B”:
      r = 1
    return r + recurse(x[depth], depth)

if recurse(x, 0) == 1:
  print “You win!”
```



Different Approaches

Fuzzing

- Good at finding solutions for general conditions
- Bad at finding solutions for specific conditions

Symbolic Execution

- Good at finding solutions for specific conditions
- Spends too much time iterating over general conditions

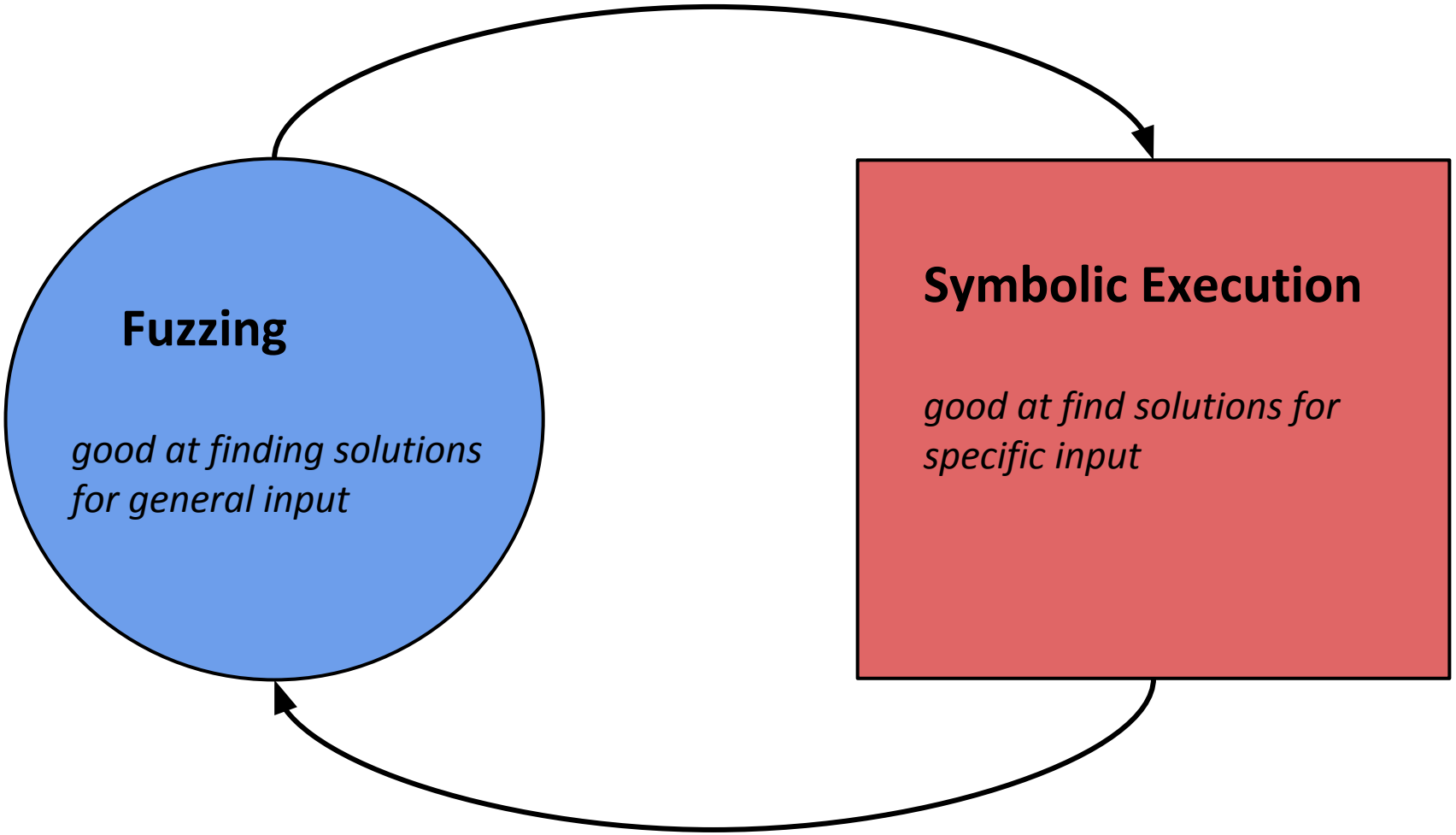
Fuzzing vs. Symbolic Execution

```
x = input()

def recurse(x, depth):
  if depth == 2000
    return 0
  else {
    r = 0;
    if x[depth] == “B”:
      r = 1
    return r + recurse(x
[depth], depth)

if recurse(x, 0) == 1:
  print “You win!”
```

Fuzzing Wins

```
x = int(input())
if x >= 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

Symbolic Execution Wins


Fuzzing
good at finding solutions for general input
Symbolic Execution
good at find solutions for specific input

American Fuzzy Lop + angr

AFL

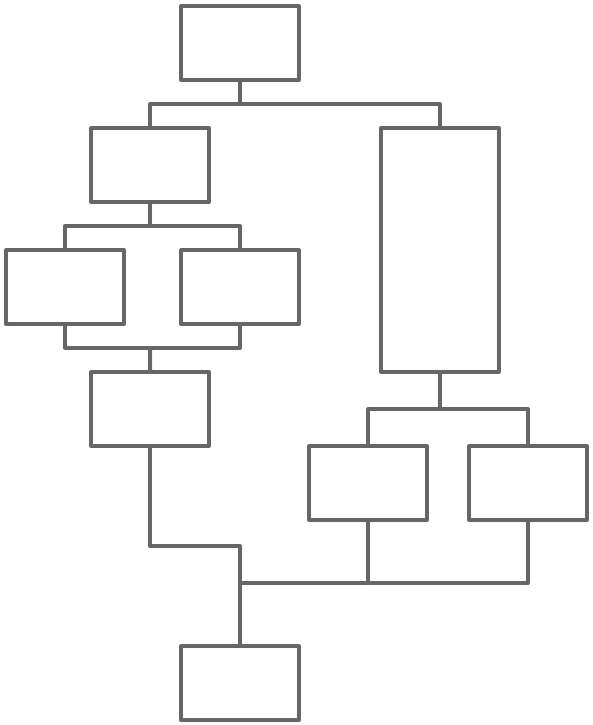
- state-of-the-art instrumented fuzzer
- path uniqueness tracking
- genetic mutations
- open source

angr

- binary analysis platform
- implements symbolic execution engine
- influenced by Mayhem
- works on binary code
- available on github

Combining the Two (High-level)

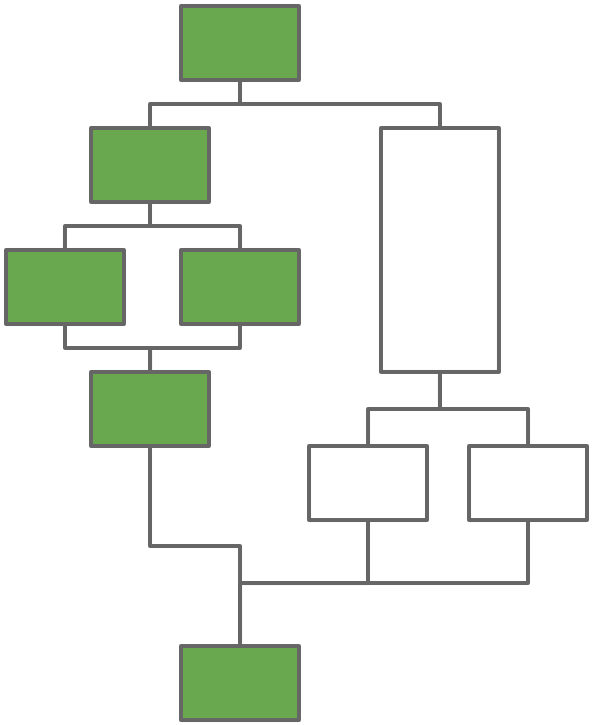
Test Cases



Control Flow Graph

Combining the Two

"Cheap" fuzzing coverage



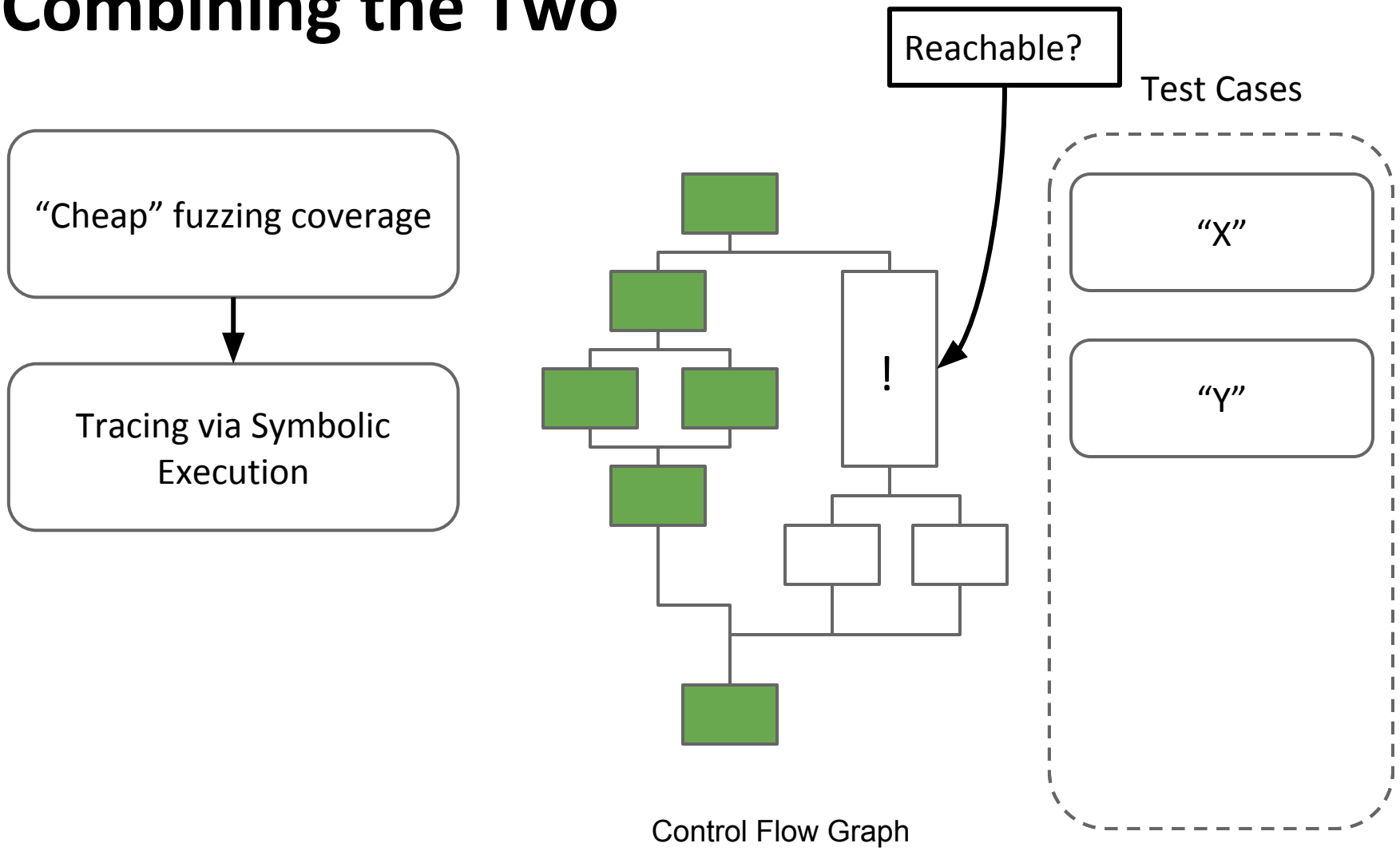
Control Flow Graph

Test Cases

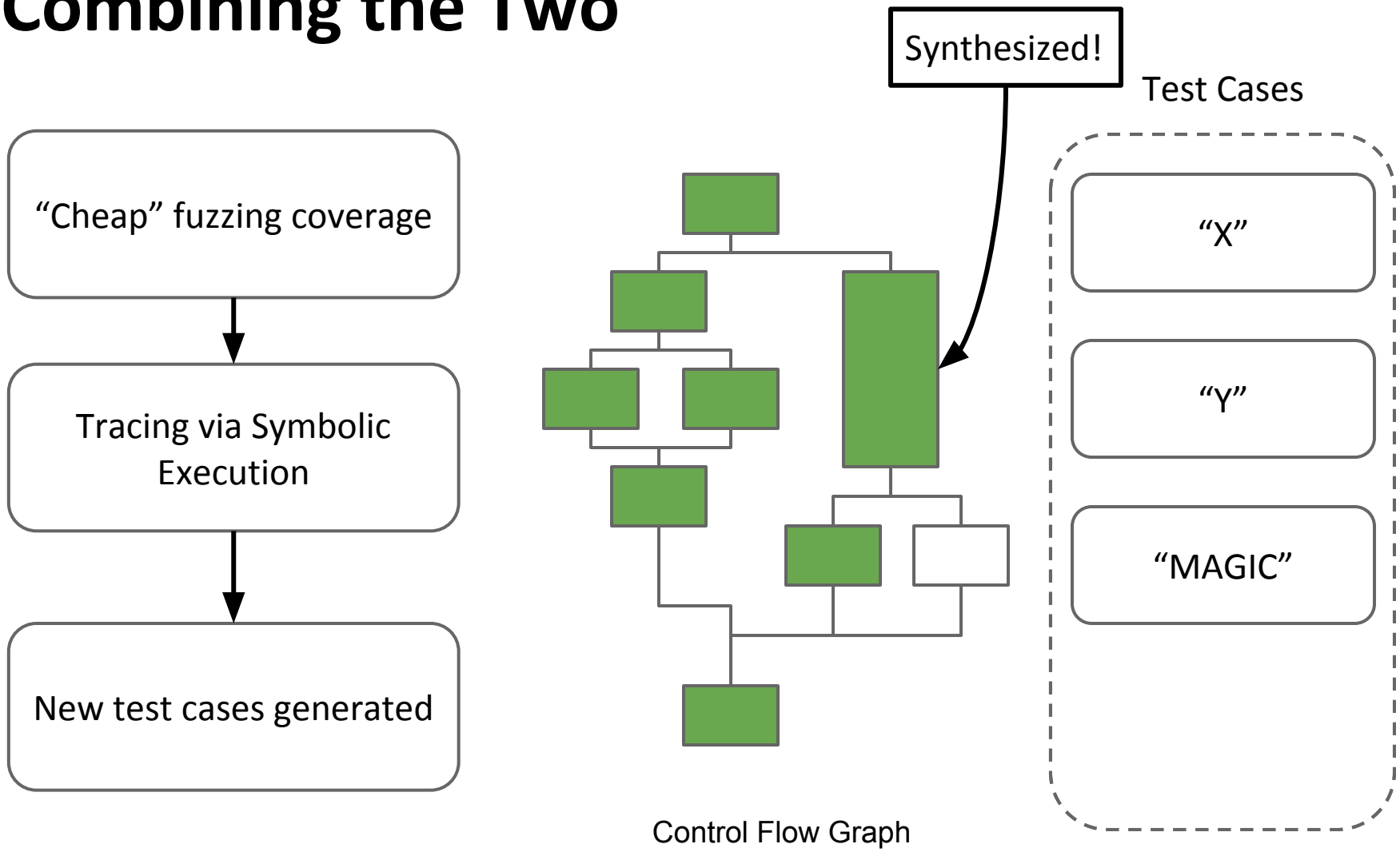
"X"

"Y"

Combining the Two

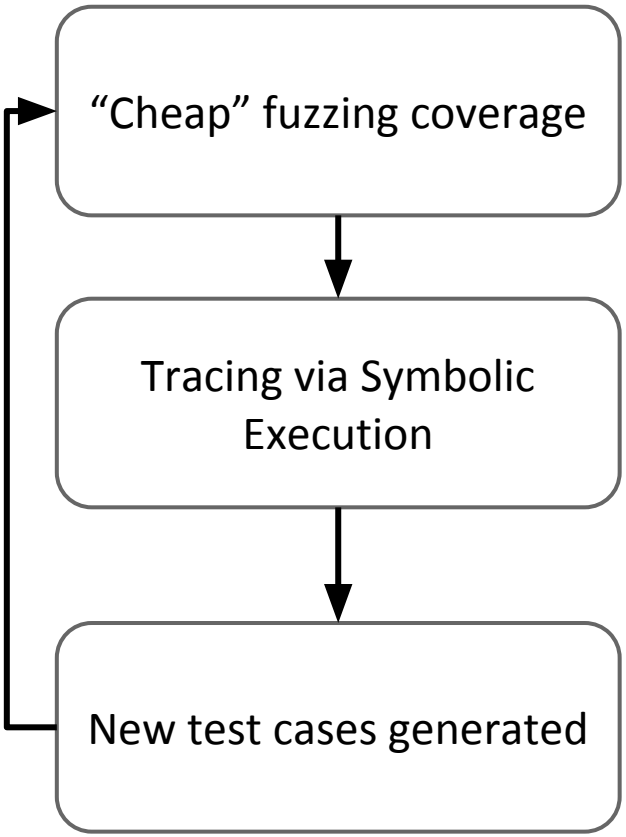


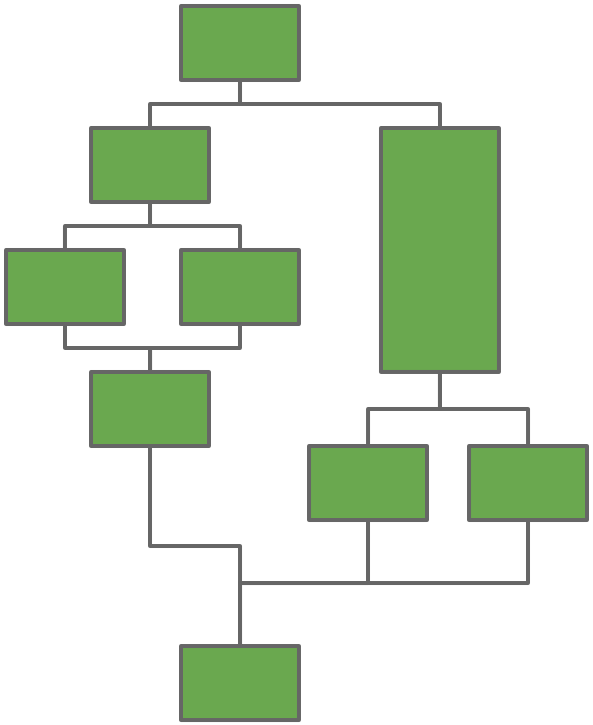
Combining the Two



Combining the Two

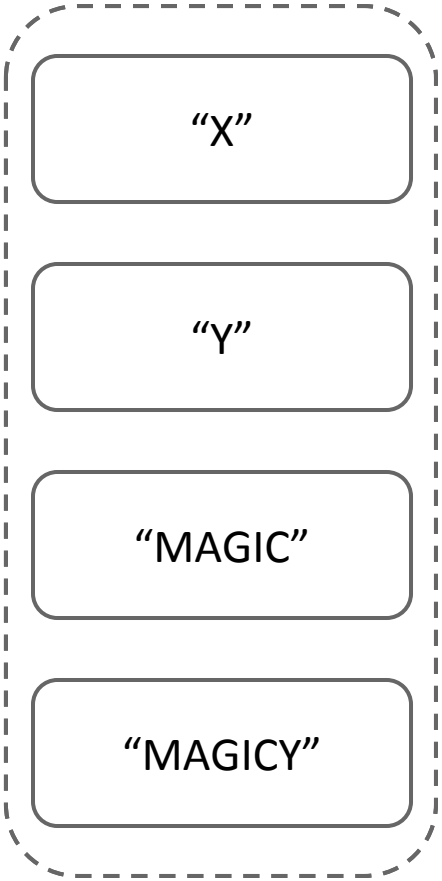
Towards completer code coverage!





Control Flow Graph

Test Cases



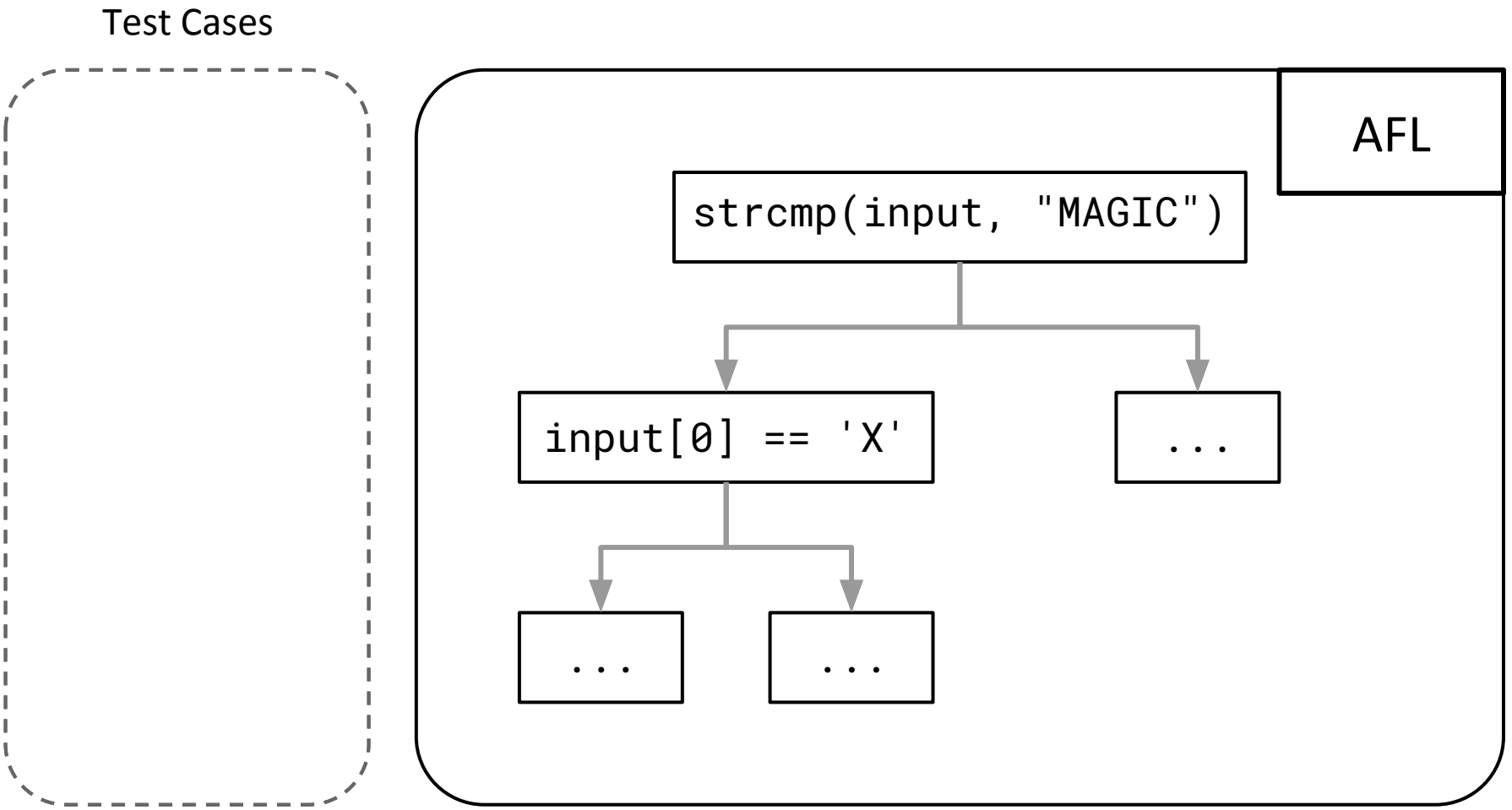
AFL's Path Selection

- Tracks state-transitions on each program run
  - Basic Block A -> Basic Block B

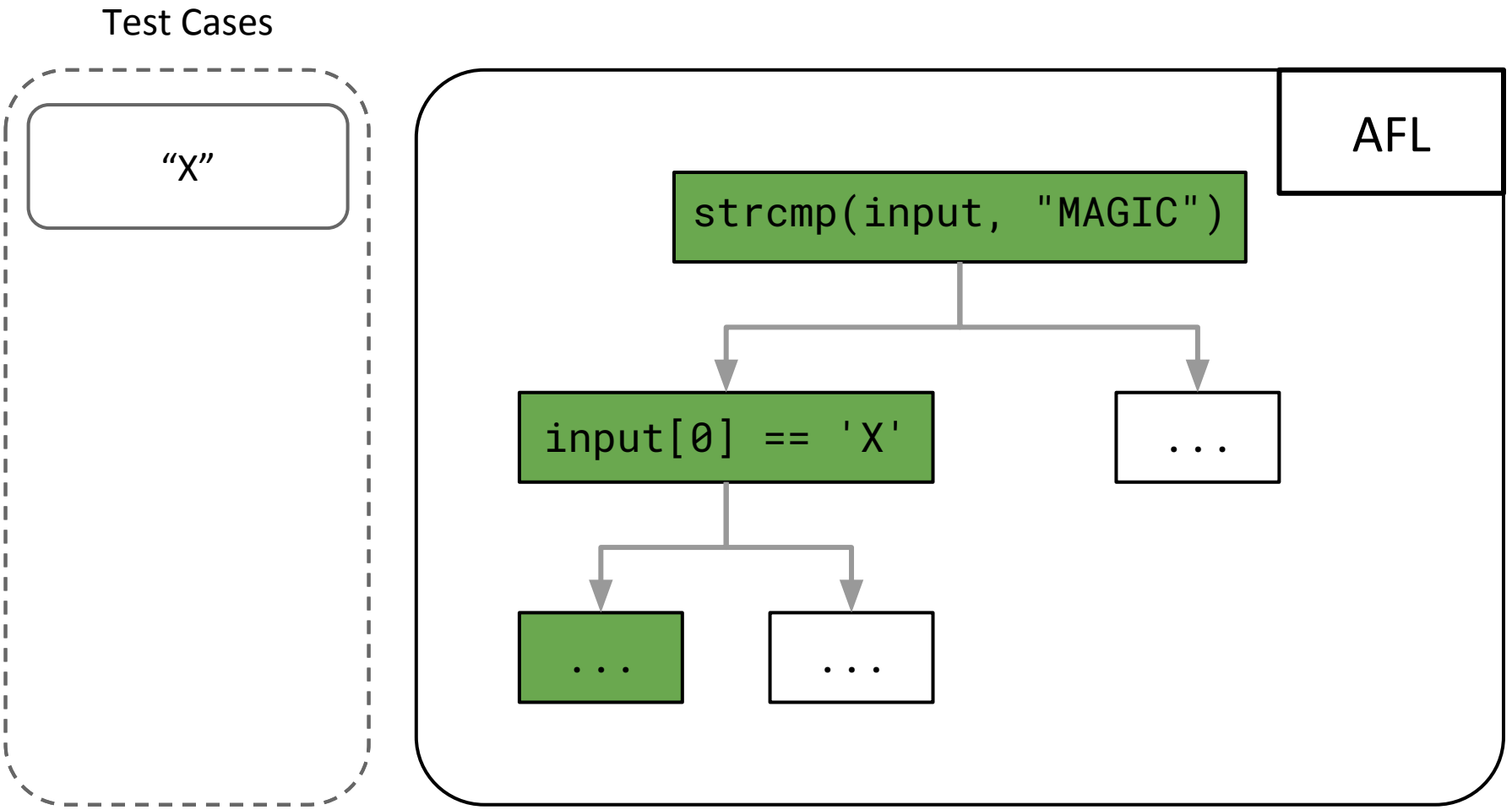
- Path uniqueness = Set of state-trans uniqueness

- Input generation is still primitive mutations

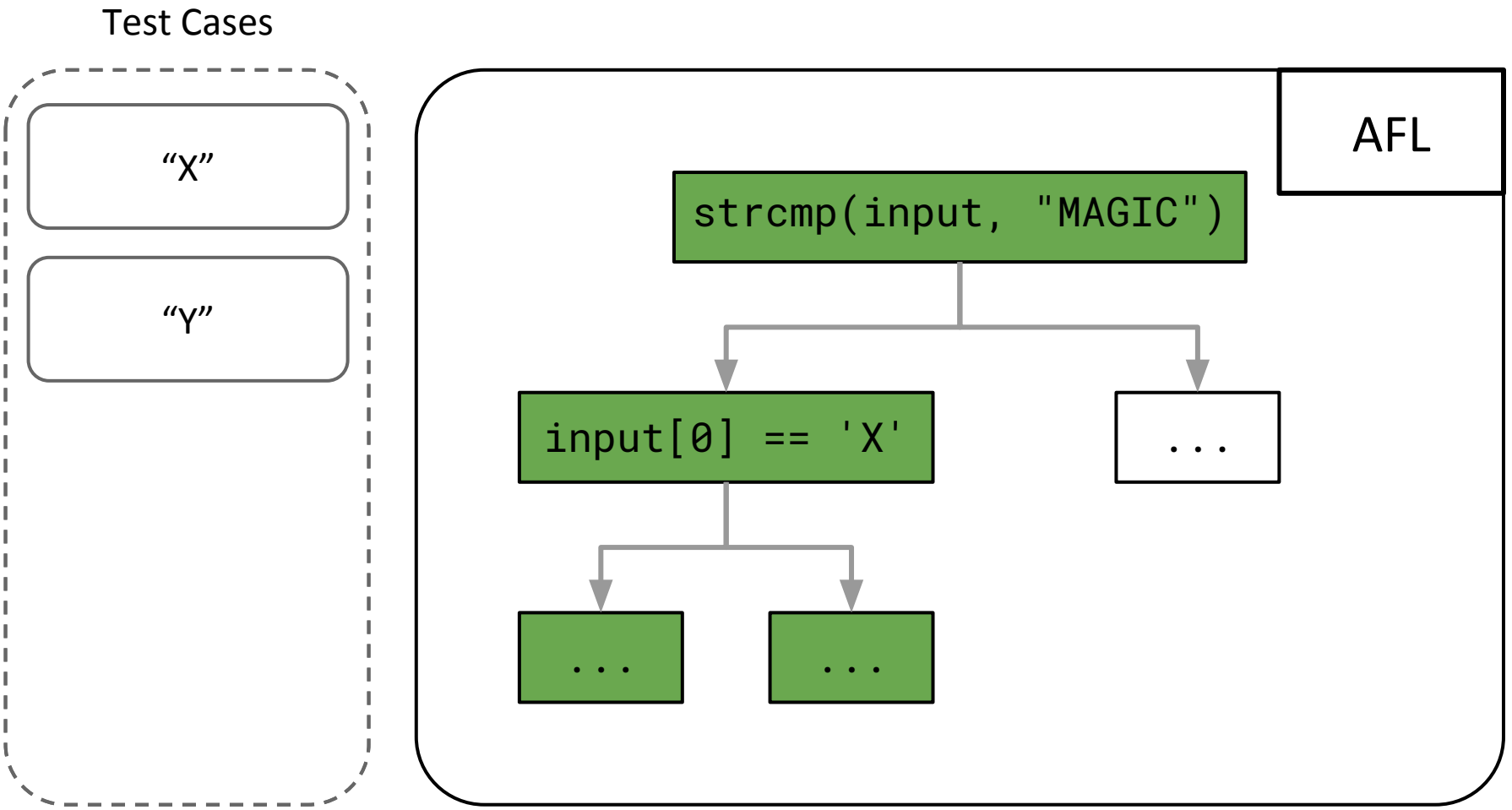
Improving Path Selection with angr



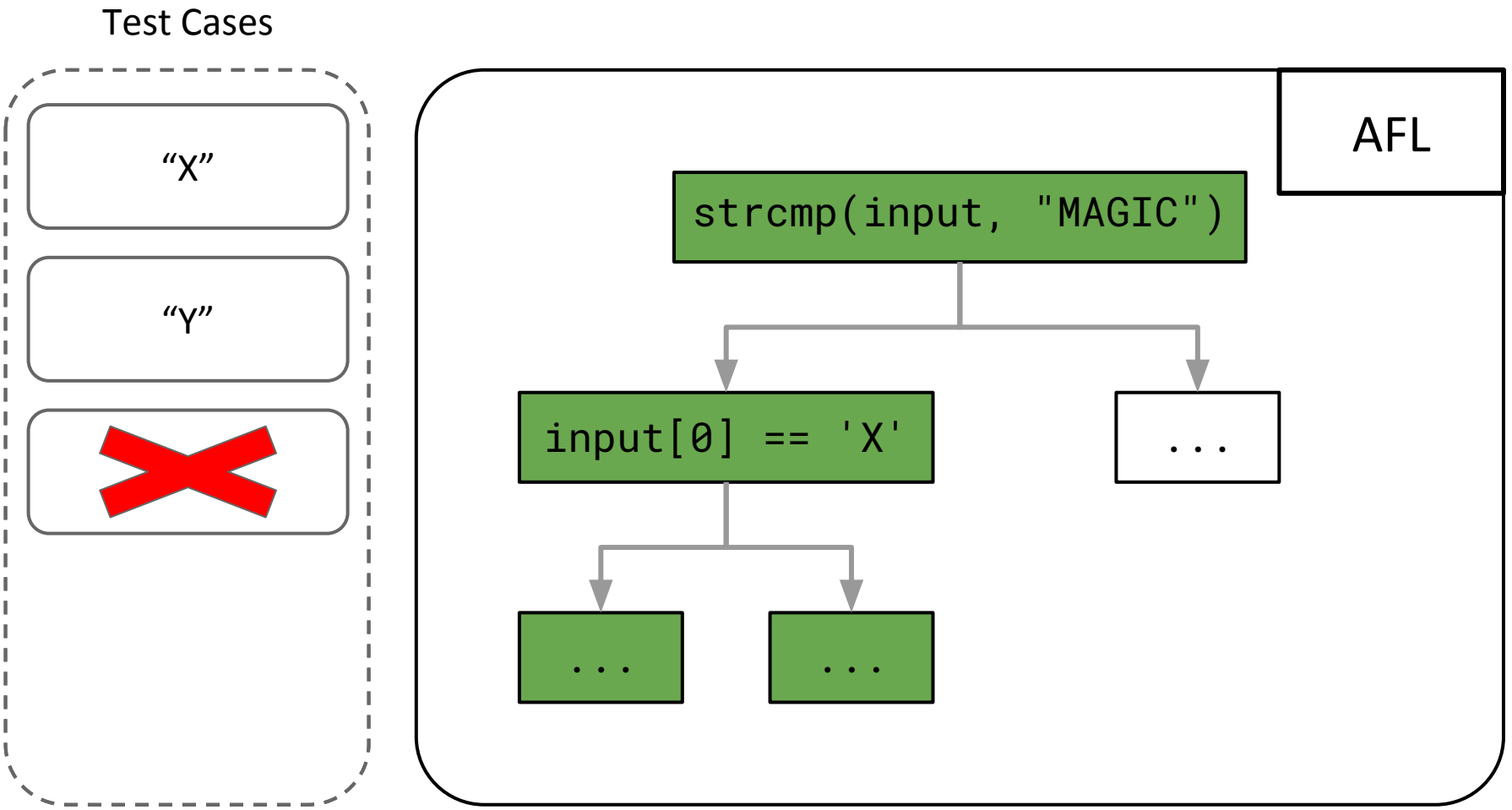
Improving Path Selection with angr



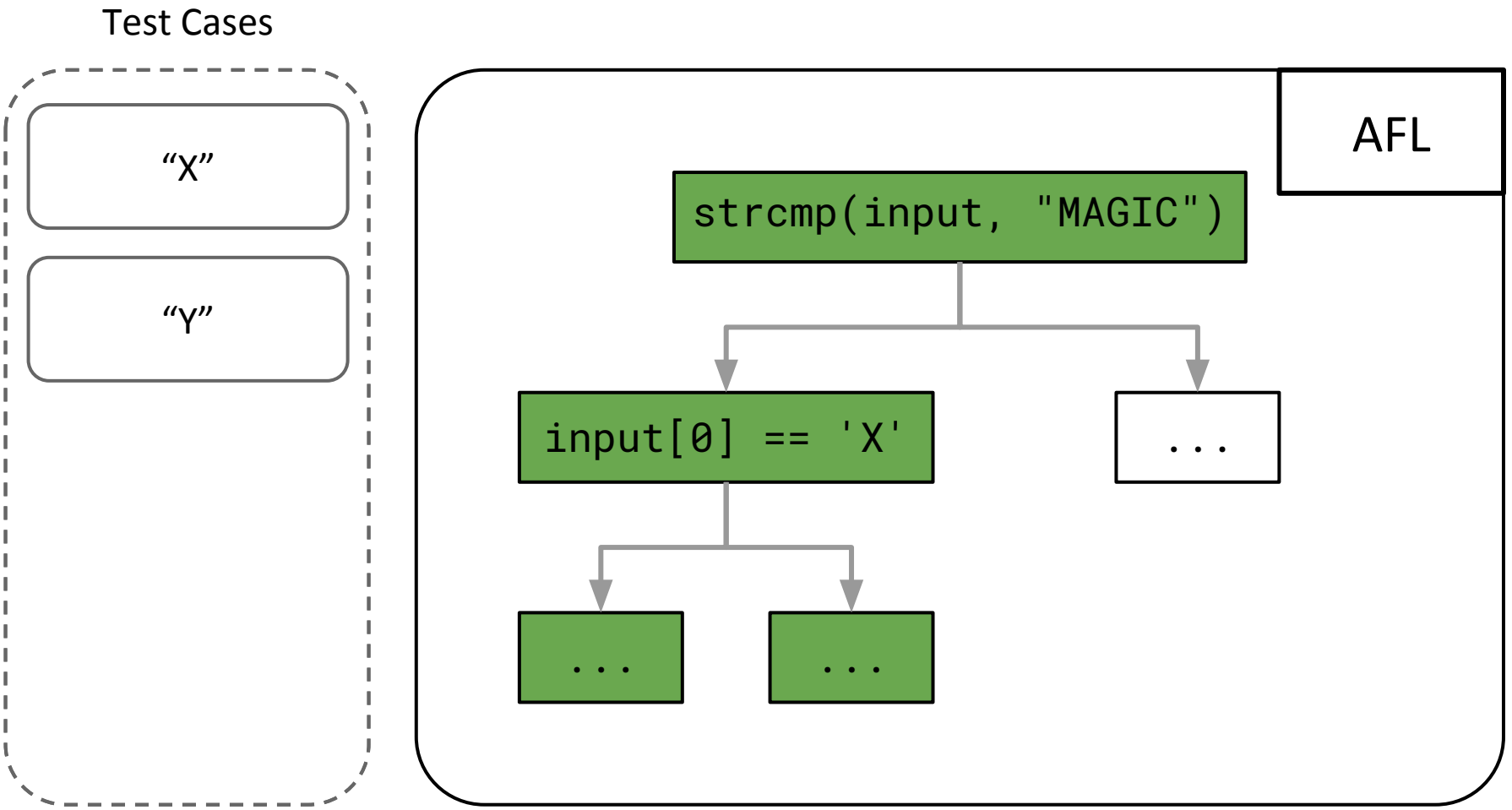
Improving Path Selection with angr



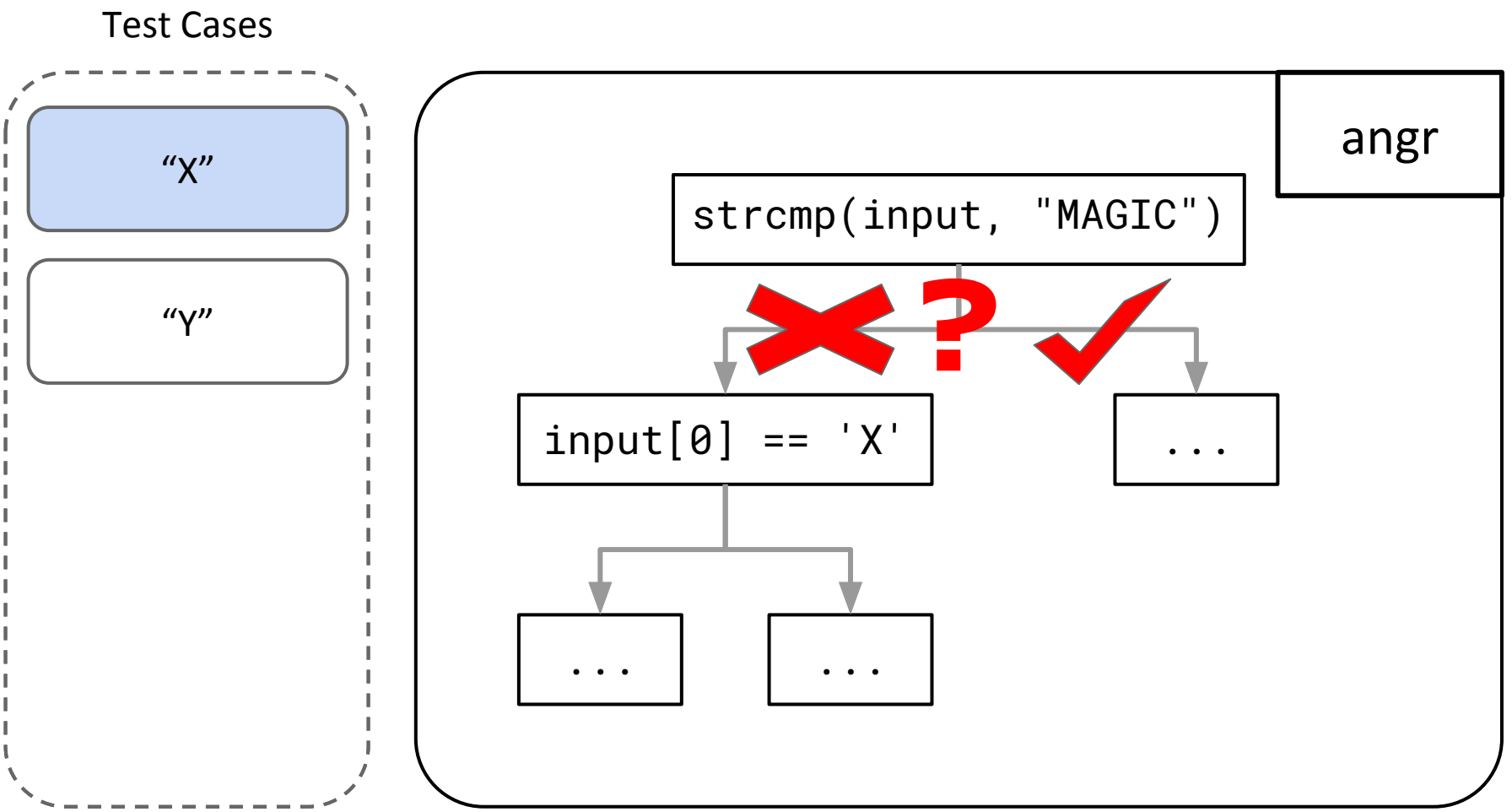
Improving Path Selection with angr



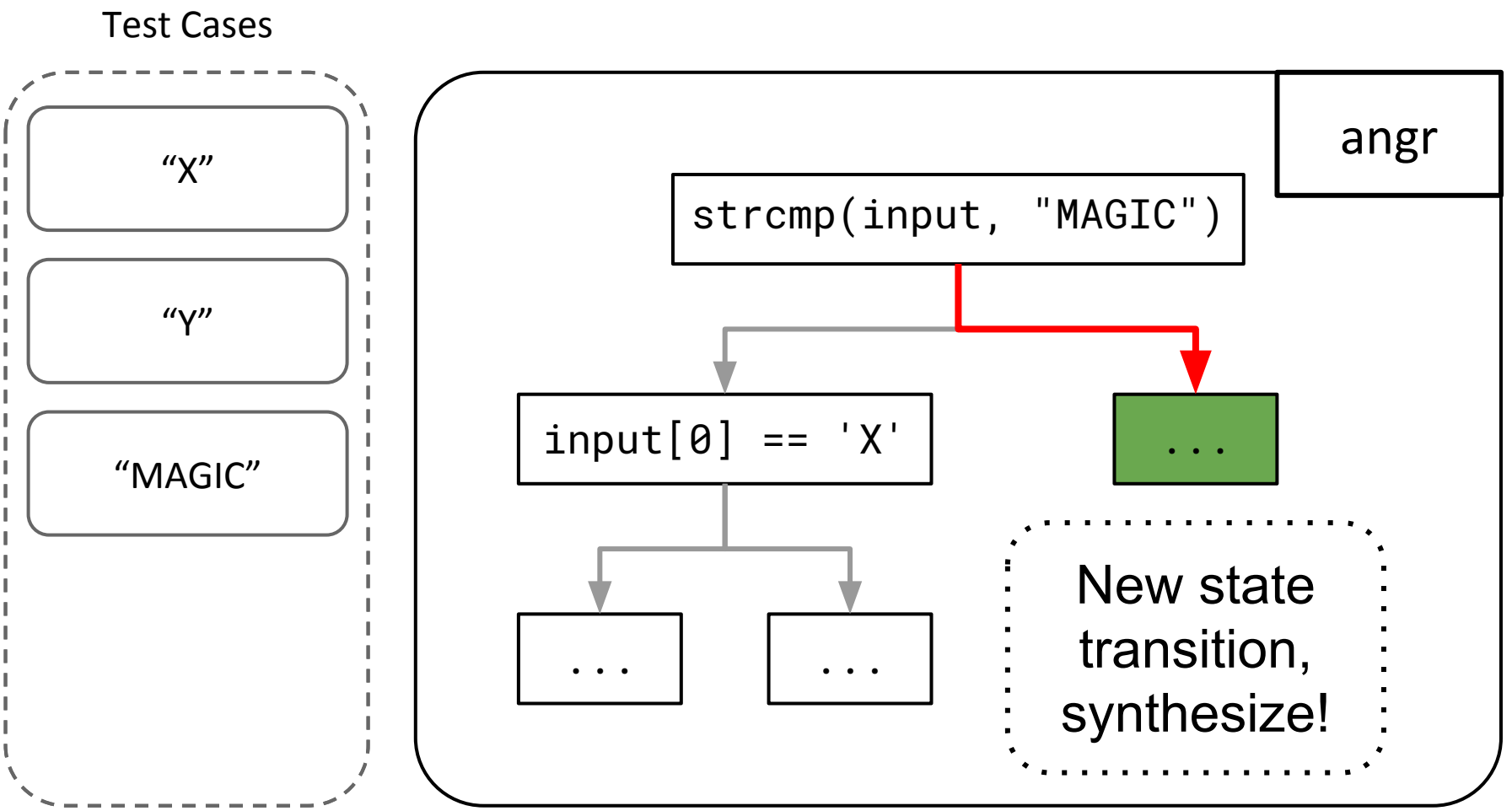
Improving Path Selection with angr



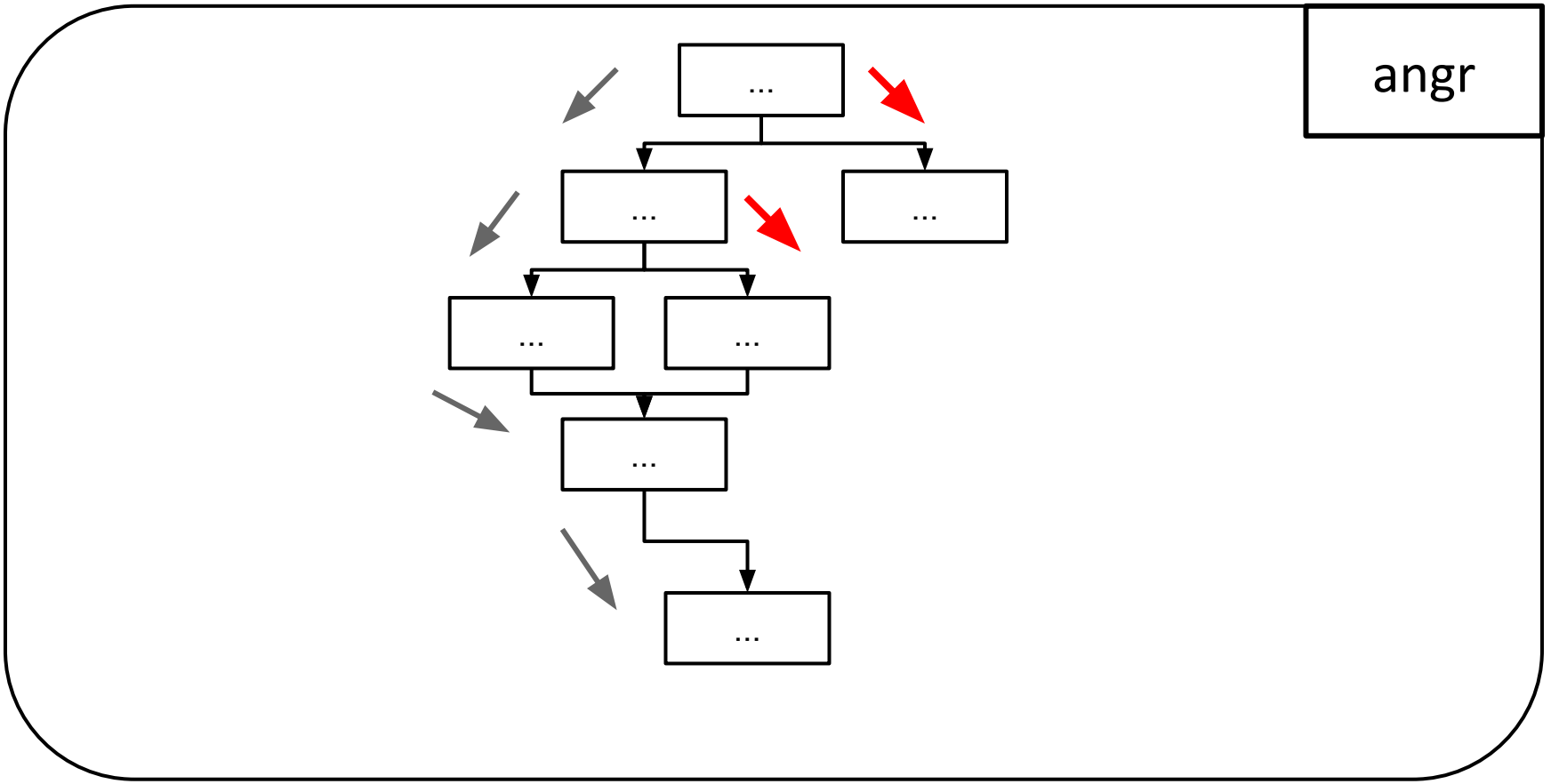
Improving Path Selection with angr



Improving Path Selection with angr



Improving Path Selection with angr



Continue following “X”’s original path until completion, deviating when possible.

State Space Reduction

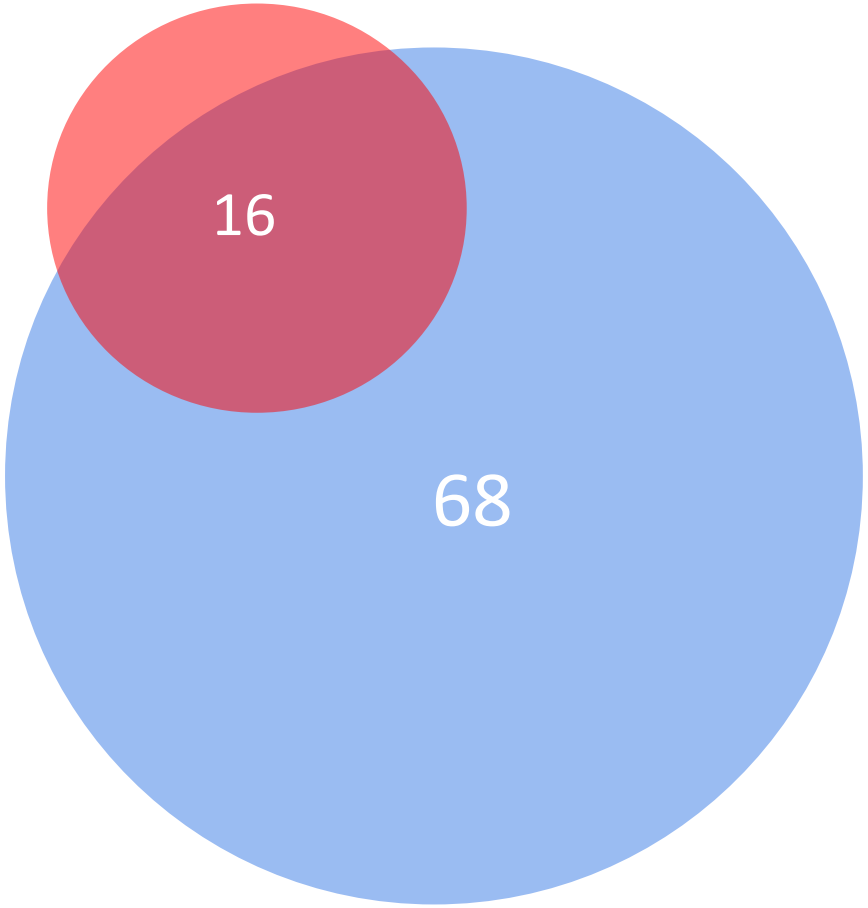
- Symbolic Execution's state-space is reduced to AFL's

- Reduces path explosion

Symbolic Execution (angr) - 16 total

Fuzzing (AFL) - 68 total

S & F Shared - 13 total



71 / 128 binaries

Binary Crashes per Technique

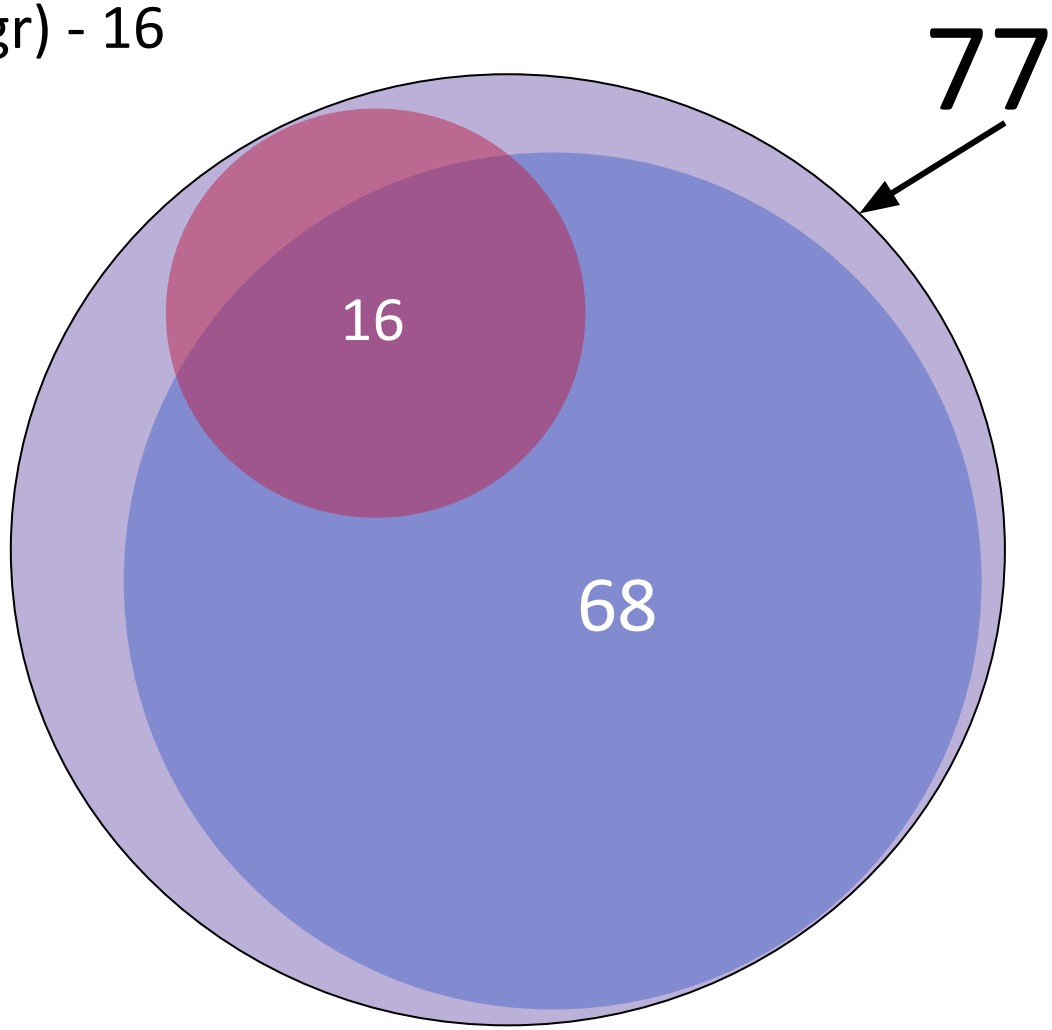
Symbolic Execution (angr) - 16

Fuzzing (AFL) - 68

S & F Shared - 13 total

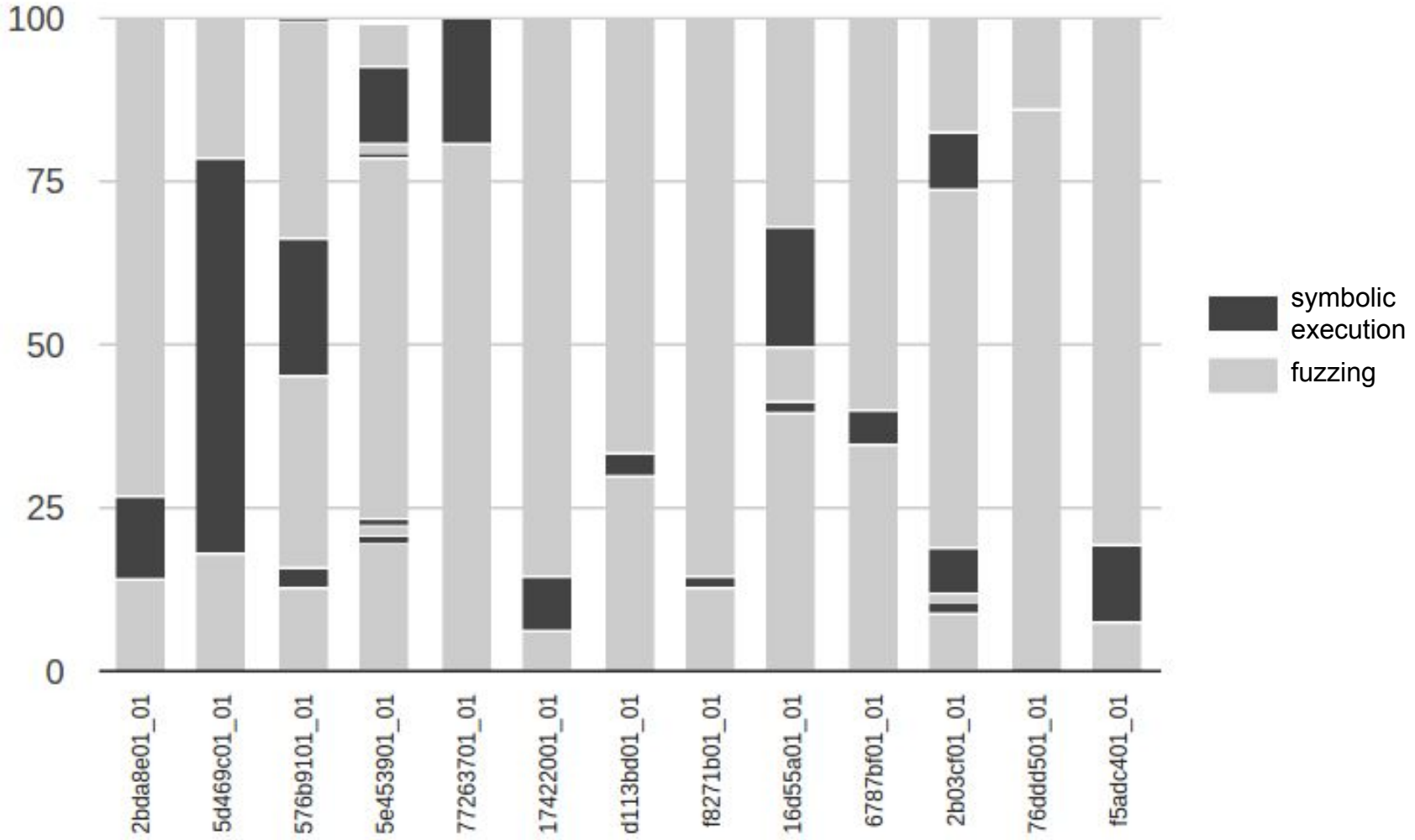
Driller - 77

77 / 128 binaries



Binary Crashes per Technique

Distribution of Transitions Found as Iterations of Symbolic Execution and Fuzzing



Limitations

```
int main(void) {
    char data[100];
    char *computed_hash;
    char hash[16];

    read(0, data, sizeof data);

    computed_hash = hash(data);

    read(0, hash, sizeof hash);

    if (memcmp(hash, computed_hash, 16) != 0) {
        // `data` processed here
        // code susceptible to fuzzing
    }
}
```

Fuzzing beyond the hash is still problematic!

Conclusion

- Driller is greater than the sum of its parts
- Offers a >10% increase in crashes over pure AFL
- Driller curbs path explosion