Systems and Internet Infrastructure Security

Network and Security Research Center
Department of Computer Science and Engineering
Pennsylvania State University, University Park PA

CMPSC 447
Symbolic Execution

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

Our Goal

PENNSTATE 1855

- In this course, we want to develop techniques to detect vulnerabilities before they are exploited automatically
  - What's a vulnerability?
  - How to find them?

Static vs. Dynamic

- Dynamic
  - Depends on concrete inputs
  - Must run the program
  - Impractical to run all possible executions in most cases
- Static
  - Overapproximates possible input values (sound)
  - Assesses all possible runs of the program at once
  - Setting up static analysis is somewhat of an art form
- Is there something that combines best of both?

Best of Both?

- What would be the best of both?

Best of Both?

- What would be the best of both?
  - Run over lots of inputs at once (static)
  - Easy to setup (dynamic)
  - Run all paths (static)
  - Identify concrete values that lead to problems (dynamic)
- Can't quite achieve all these, but can come closer

Symbolic Execution

- Symbolic execution is a method for emulating the execution of a program to learn constraints
  - Assign variables to symbolic values instead of concrete values
  - Symbolic execution tells you what values are possible for symbolic variables at any particular point in your program
- Like dynamic analysis (fuzzing) in that the program is executed in a way – albeit on symbolic inputs
- Like static analysis in that one start of the program tells you what values may reach a particular state

Symbolic Execution

- What's a symbolic value?
- Remember in AFL fuzzing, you provide a candidate concrete input to identify the format
  - And the fuzzer produces lots of variants of this input
- In symbolic execution, you don't provide a concrete input, but rather identify which value(s) you want to assess – just say an input is "symbolic"
  - Then the symbolic execution tells you the possible values of that input to reach particular points in the program

EXE & KLEE

Slides by Yoni Leibowitz

Example

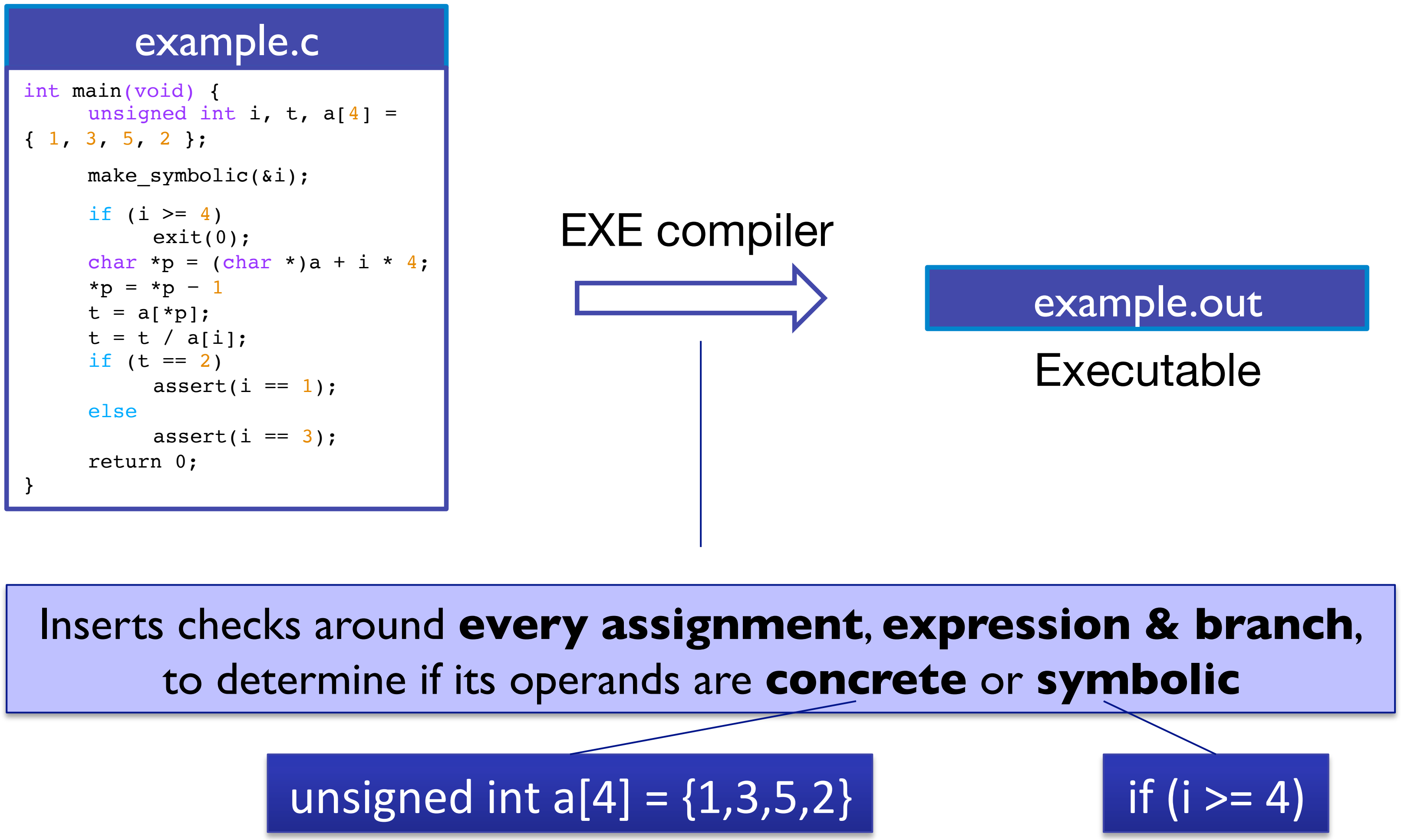
```
int main(void) {
   unsigned int i, t, a[4] = { 1, 3, 5, 2 };


   if (i >= 4)
      exit(0);
   char *p = (char *)a + i * 4;
   *p = *p - 1
   t = a[*p];
   t = t / a[i];
   if (t == 2)
      assert(i == 1);
   else
      assert(i == 3);
   return 0;
}
```

Marking Symbolic Data

```
int main(void) {
   unsigned int i, t, a[4] = { 1, 3, 5, 2 };
   make_symbolic(&i);

   if (i >= 4)
      exit(0);
   char *p = (char *)a + i * 4;
   *p = *p − 1
   t = a[*p];
   t = t / a[i];
   if (t == 2)
      assert(i == 1);
   else
      assert(i == 3);
   return 0;
}
```

Marks the 4 bytes associated with 32-bit variable 'i' as **symbolic**

Compiling...



```
int main(void) {
      unsigned int i, t, a[4] =
{ 1, 3, 5, 2 };
      make_symbolic(&i);
      if (i >= 4)
            exit(0);
      char *p = (char *)a + i * 4;
      *p = *p − 1
      t = a[*p];
      t = t / a[i];
      if (t == 2)
            assert(i == 1);
      else
            assert(i == 3);
      return 0;
}
```

Inserts checks around **every assignment**, **expression & branch**, to determine if its operands are **concrete** or **symbolic**

Compiling...

example.c

```
int main(void) {
      unsigned int i, t, a[4] =
{ 1, 3, 5, 2 };
      make_symbolic(&i);
      if (i >= 4)
            exit(0);
      char *p = (char *)a + i * 4;
      *p = *p − 1
      t = a[*p];
      t = t / a[i];
      if (t == 2)
            assert(i == 1);
      else
            assert(i == 3);
      return 0;
}
```

EXE compiler $\Longrightarrow$

example.out

Executable

Inserts checks around **every assignment**, **expression & branch**, to determine if its operands are **concrete** or **symbolic**

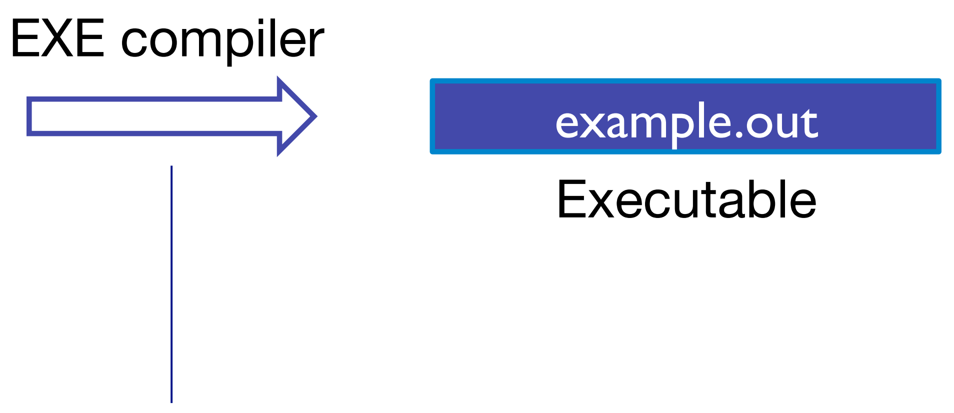
If any operand is **symbolic**, the operation is not performed, but is **added as a constraint** for the current path

Compiling...

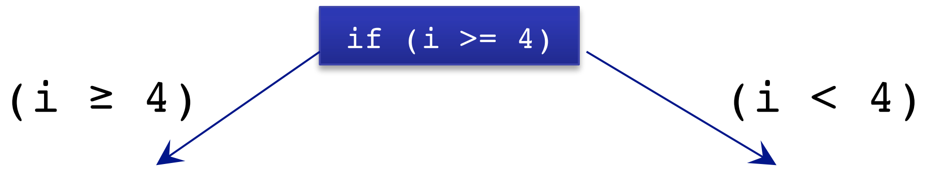
PENNSTATE

example.c

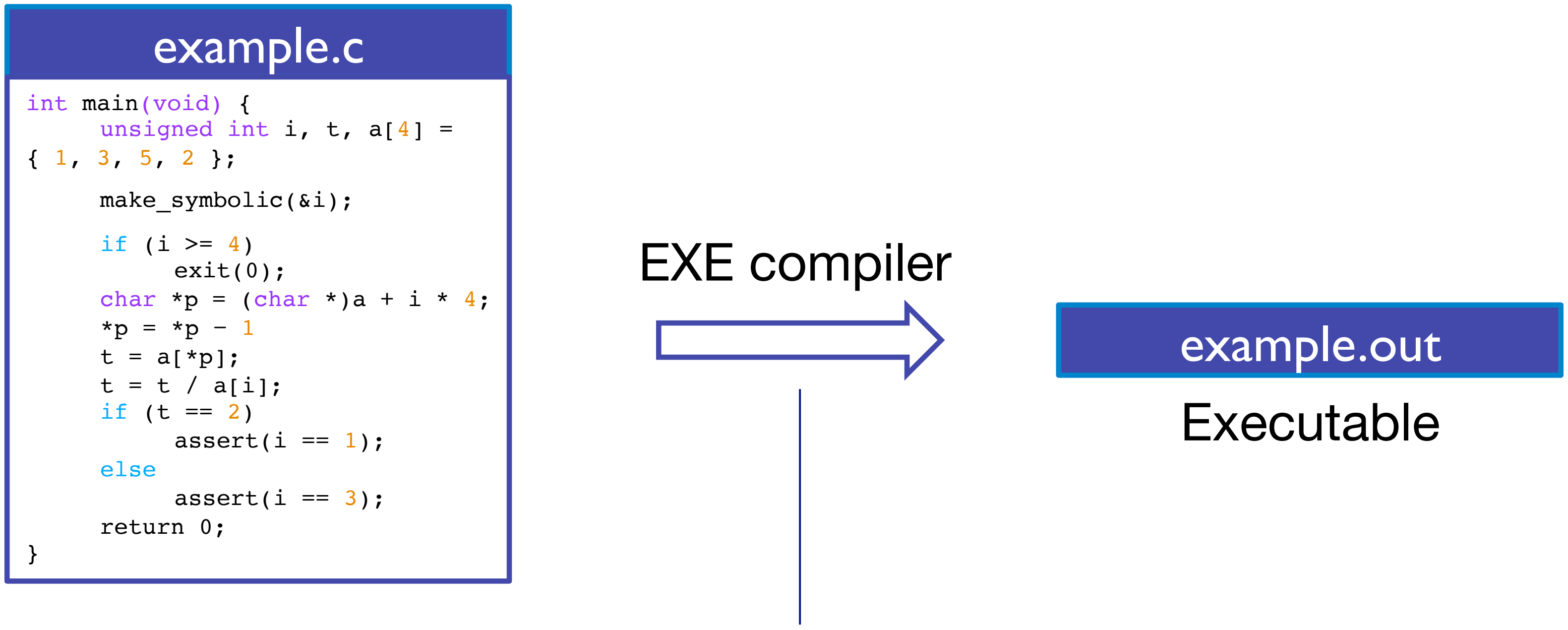
```
int main(void) {
      unsigned int i, t, a[4] =
{ 1, 3, 5, 2 };

      make_symbolic(&i);

      if (i >= 4)
            exit(0);
      char *p = (char *)a + i * 4;
      *p = *p − 1
      t = a[*p];
      t = t / a[i];
      if (t == 2)
            assert(i == 1);
      else
            assert(i == 3);
      return 0;
}
```



Inserts code to **fork** program execution when it reaches a **symbolic branch point**, so that it can explore **each possibility**



Compiling...



example.c

```
int main(void) {
      unsigned int i, t, a[4] =
{ 1, 3, 5, 2 };

      make_symbolic(&i);

      if (i >= 4)
            exit(0);
      char *p = (char *)a + i * 4;
      *p = *p − 1
      t = a[*p];
      t = t / a[i];
      if (t == 2)
            assert(i == 1);
      else
            assert(i == 3);
      return 0;
}
```

Inserts code to **fork** program execution when it reaches a **symbolic branch point**, so that it can explore **each possibility**

For each **branch constraint**, queries constraint solver for existence of **at least one solution for the current path**. If not – stops executing path

Compiling...

example.c

```c
int main(void) {
      unsigned int i, t, a[4] =
{ 1, 3, 5, 2 };

      make_symbolic(&i);

      if (i >= 4)
            exit(0);
      char *p = (char *)a + i * 4;
      *p = *p − 1
      t = a[*p];
      t = t / a[i];
      if (t == 2)
            assert(i == 1);
      else
            assert(i == 3);
      return 0;
}
```

EXE compiler

example.out

Executable

Inserts code for checking if a **symbolic expression** could have **any possible value** that could cause **errors**

```
t = t /
a[i]
```

Division by Zero?

Compiling...

example.c

```
int main(void) {
      unsigned int i, t, a[4] =
{ 1, 3, 5, 2 };
      make_symbolic(&i);
      if (i >= 4)
            exit(0);
      char *p = (char *)a + i * 4;
      *p = *p − 1
      t = a[*p];
      t = t / a[i];
      if (t == 2)
            assert(i == 1);
      else
            assert(i == 3);
      return 0;
}
```

$$\xrightarrow{\text{EXE compiler}}$$

example.out

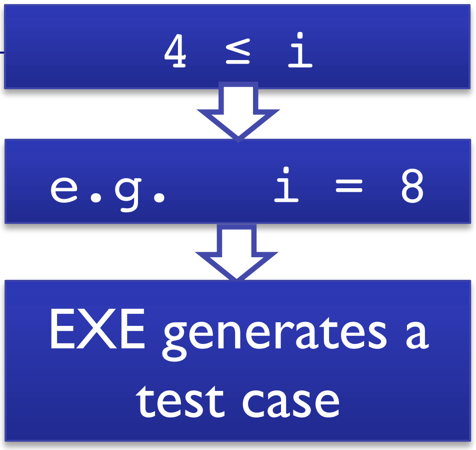
Executable

Inserts code for checking if a **symbolic expression** could have **any possible value** that could cause **errors**

If the check passes – the path has been **verified as safe under all possible input values** (relative to those checks)

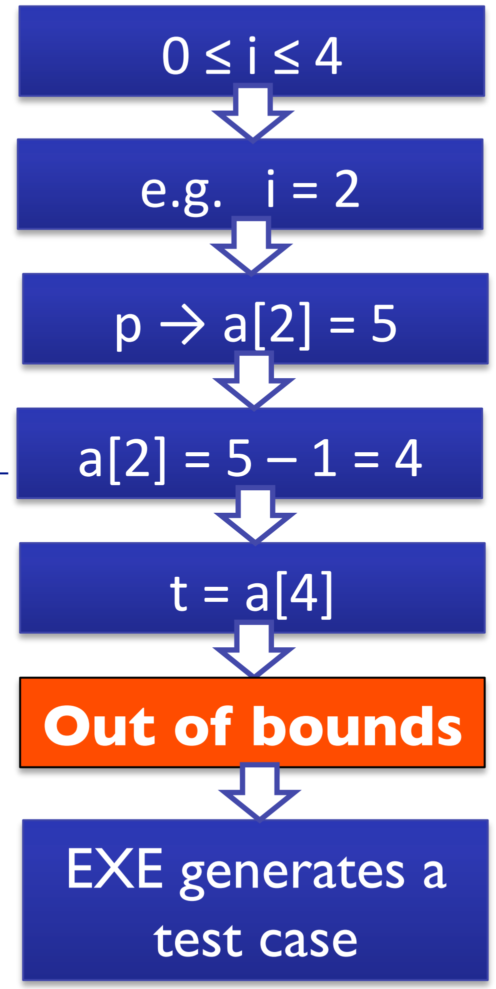
Running...

```
int main(void) {
   unsigned int i, t, a[4] = { 1, 3, 5, 2 };
   make_symbolic(&i);

   if (i >= 4)
      exit(0);
   char *p = (char *)a + i * 4;
   *p = *p – 1
   t = a[*p];
   t = t / a[i];
   if (t == 2)
      assert(i == 1);
   else
      assert(i == 3);
   return 0;
}
```



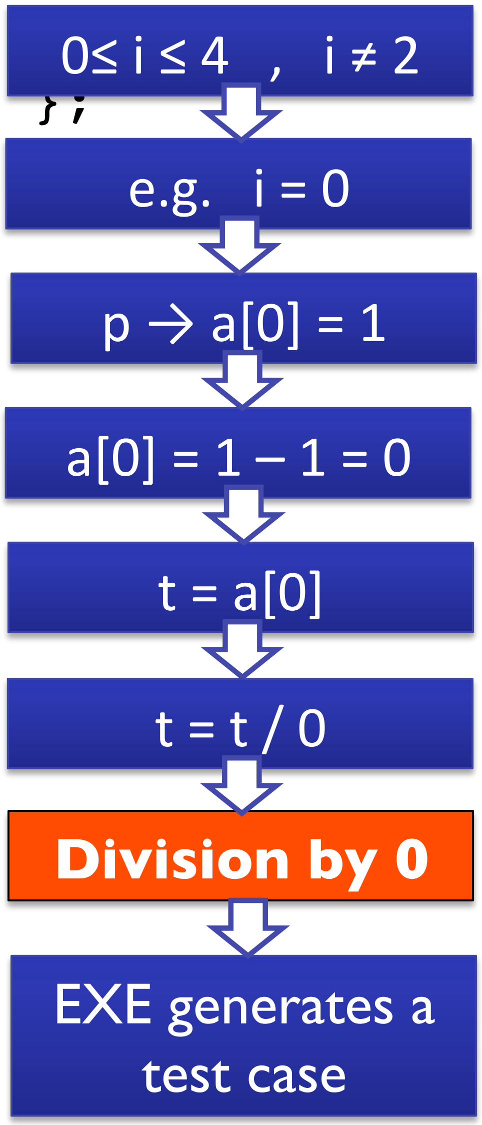
Running...

```
int main(void) {
   unsigned int i, t, a[4] = { 1, 3, 5, 2 };
   make_symbolic(&i);

   if (i >= 4)
      exit(0);
   char *p = (char *)a + i * 4;
   *p = *p - 1
   t = a[*p];
   t = t / a[i];
   if (t == 2)
      assert(i == 1);
   else
      assert(i == 3);
   return 0;
}
```



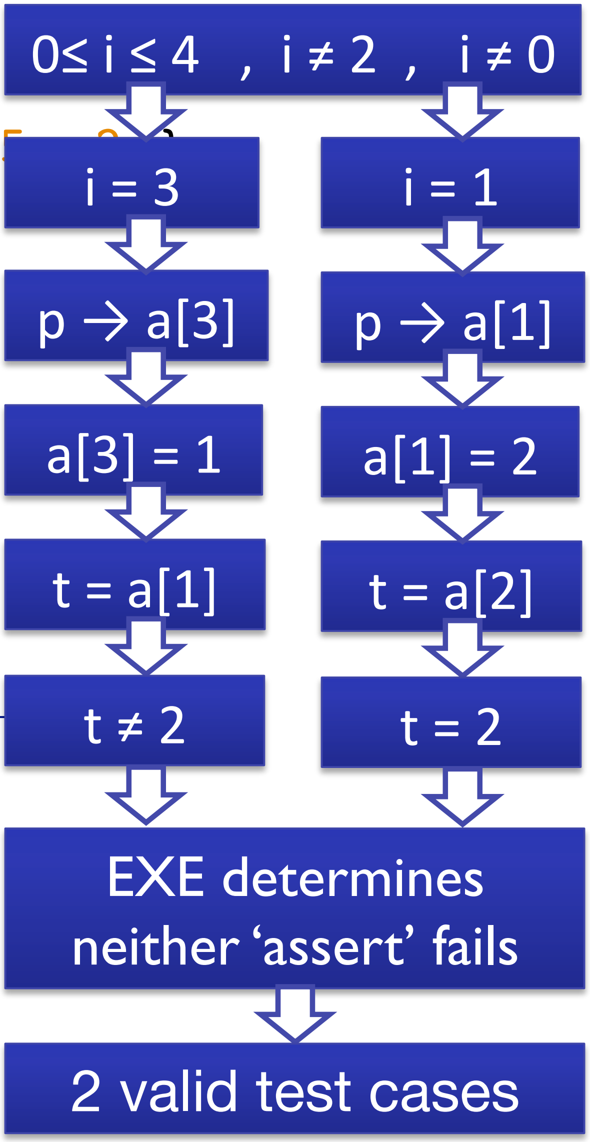
Running...

```
int main(void) {
   unsigned int i, t, a[4] = { 1, 3, 5, 2 };
   make_symbolic(&i);

   if (i >= 4)
      exit(0);
   char *p = (char *)a + i * 4;
   *p = *p - 1
   t = a[*p];
   t = t / a[i];
   if (t == 2)
      assert(i == 1);
   else
      assert(i == 3);
   return 0;
}
```



Running...

```
int main(void) {
   unsigned int i, t, a[4] = { 1, 3, 5, 2 };
   make_symbolic(&i);

   if (i >= 4)
      exit(0);
   char *p = (char *)a + i * 4;
   *p = *p - 1
   t = a[*p];
   t = t / a[i];
   if (t == 2)
      assert(i == 1);
   else
      assert(i == 3);
   return 0;
}
```



Output

test3.err

ERROR: simple.c:16 Division/modulo by zero!

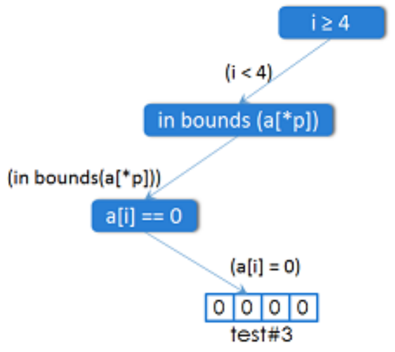
test3.out

```
# concrete byte values:
0 # i[0], 0 # i[1], 0 # i[2], 0 # i[3]
```

i = 0

test3.forks

```
# take these choices to follow path
0 # false branch (line 5)
0 # false (implicit: pointer overflow check on line 9)
1 # true (implicit: div−by−0 check on line 16)
```



Symbolic Execution

- Tracks constraints on symbolic inputs that lead to an execution point
  - Collected from conditionals executed so far
  - And other statements that restrict values of variable
- Executes all paths (it can in a reasonable time)
  - Assesses whether a path is legal given concrete inputs and constraints collected on symbolic inputs
  - If so, forks a new analysis at each conditional
- Generate test cases at security-sensitive operations to detect flaws

Challenges

- Exponential number of paths in a program, so still intractable to achieve full coverage
  - Even to ensure that the symbolic executor reaches a particular statement in the program may require some assistance (e.g., from static analysis)
  - Problem: Loops and floating point numbers
- Can be expensive
  - Need to call a constraint solver to produce test cases
    - Constraint satisfaction problems are intractable, but significant advancements in this area have improved effectiveness in practice

Challenges

- What types of flaws do you want to find?
  - Checks must be generate to look for those flaws
- Focus was initially on basic types of errors
  - Division by zero
  - Overflow
  - Out-of-bounds memory reference
- There are lots of different types of flaws that are possible, including more types of memory errors

Challenges

- Environment
  - If the program interacts with environment, need some way to gather information resulting from such interactions
  - System calls – what are the return values from the operating system from a system?
    - Could vary depending on the state of the OS, which is not modeled by the symbolic executor
  - Multi-threaded programs
    - Another thread may impact variables concurrently, which is not modeled by the executor

Utility

- Nonetheless, symbolic execution finds many flaws
- Used to find bugs in many programs including
  - 2 packet filters (FreeBSD & Linux)
  - Filesystems
  - DHCP server (udhcpd)
  - Perl compatible regular expressions library (pcre)
  - XML parser library (expat)
- Like dynamic analysis, detects real flaws
  - No false positives!

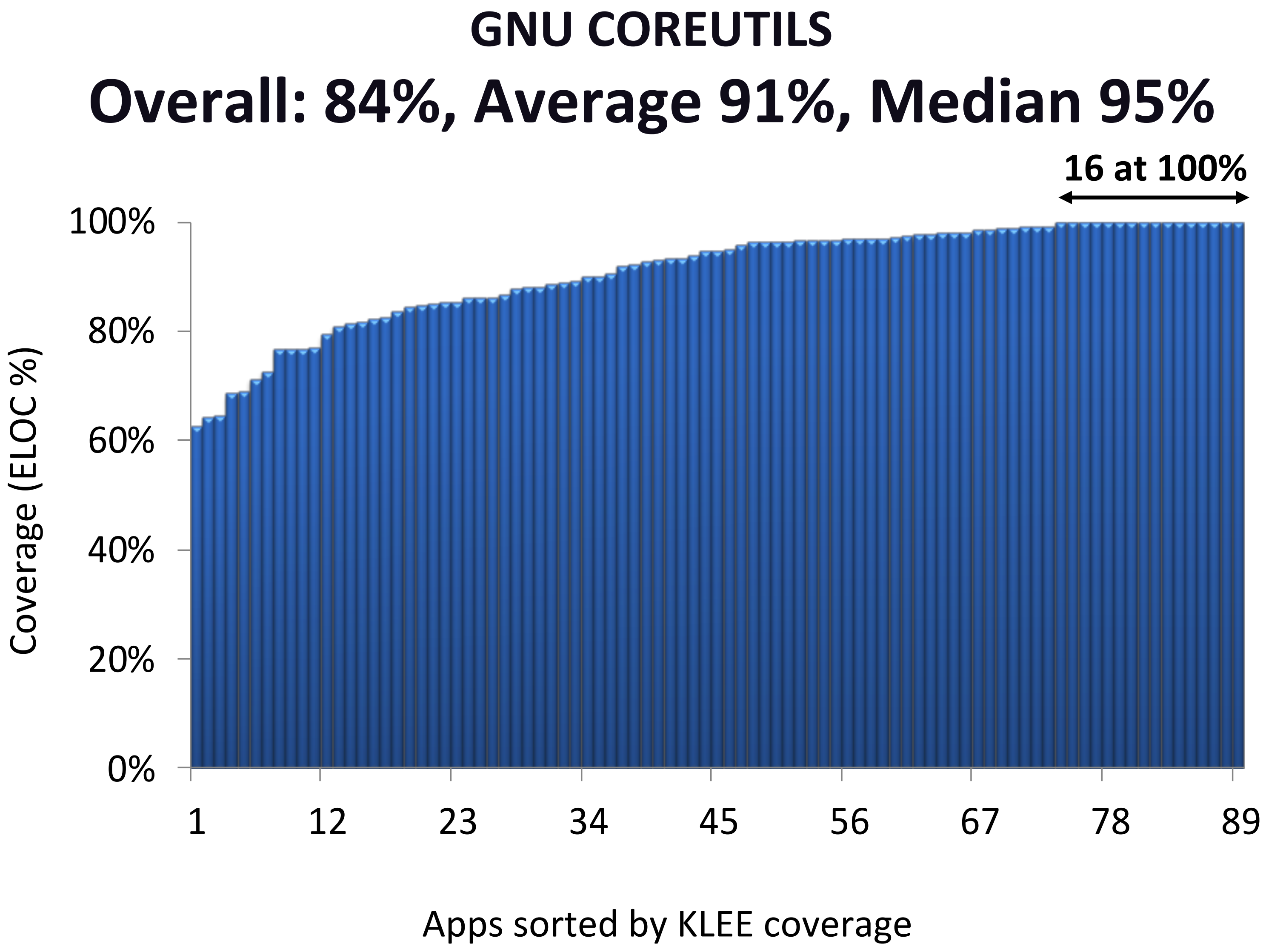
Results – Bugs found

- 10 memory error crashes in GNU COREUTILS
  - More than found in previous 3 years combined
  - Generates actual command lines exposing crashes

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Results – Line Coverage



Mixing Concrete and Symbolic

- This is called "concolic execution"
  - Used to deal with the environmental limitations
- From concrete to symbolic and back
  - Run program concretely until call Function A
  - Run Function A symbolically in full (all paths)
  - Then, produce one or more return values for Function A to continue to run program concretely
- From symbolic to concrete and back
  - Run symbolically until it reaches an external component (e.g., system call) and then run concretely on that

Static Analysis Can Help

- Address/mitigate limitations of symbolic execution
  - Limitation: exponential number of paths
    - How do we enable the analysis to check for flaws at a particular statement if the control flow is complex?
    - I.e., Symbolic execution may take a long time to reach that statement

Static Analysis Can Help

- Address/mitigate limitations of symbolic execution
  - Taint analysis: can determine what statements use data tainted by interesting inputs
    - Some statements may be security-sensitive, so we want to test what values interesting inputs may be assigned at such statements
  - Symbolic execution would make such inputs symbolic, but it may be difficult or slow for the symbolic execution to reach these security-sensitive statements
    - A static taint analysis would identify the control flows that lead from the statements receiving the interesting inputs to the security-sensitive statement
    - Direct the control flow of the symbolic analysis along that path

Helping Fuzzing

- One problem in fuzzing is to generate inputs to cover all paths
  - Can symbolic execution help with this?

Helping Fuzzing

- One problem in fuzzing is to generate inputs to cover all paths
  - Can symbolic execution help with this?
  - Driller: Augmenting Fuzzing through Symbolic Execution
    - Slides from Nick Stephens at NDSS 2016

Helping 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!"

Helping Fuzzing

```
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!"

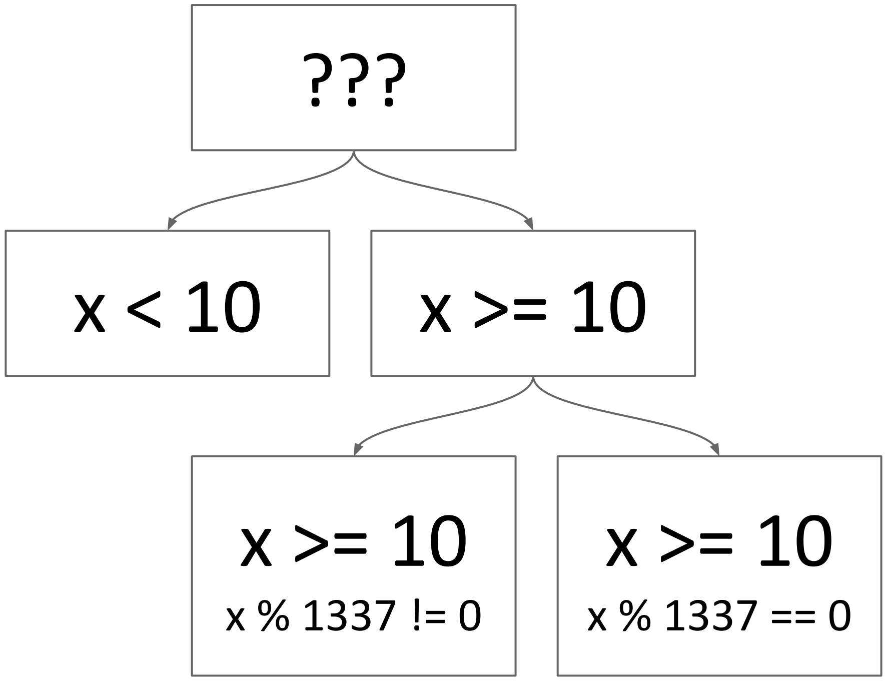
3 ⇒ "You lose!"

....

57 ⇒ "You lose!"

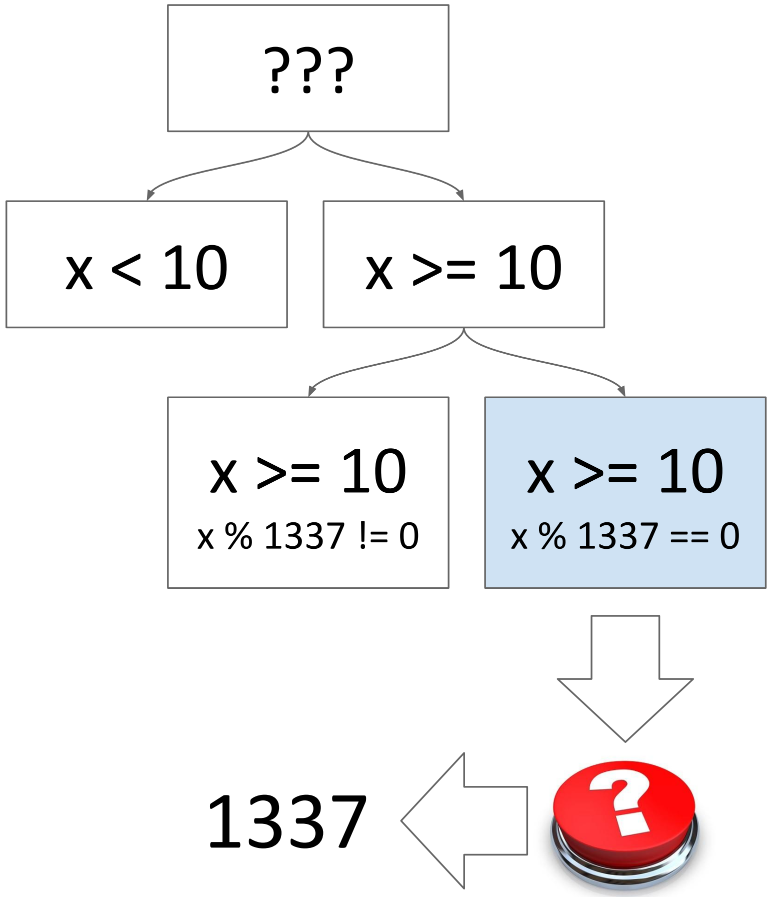
With Symbolic Execution

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



With Symbolic Execution

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



Different Approaches

PENNSTATE 1855

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 Exec

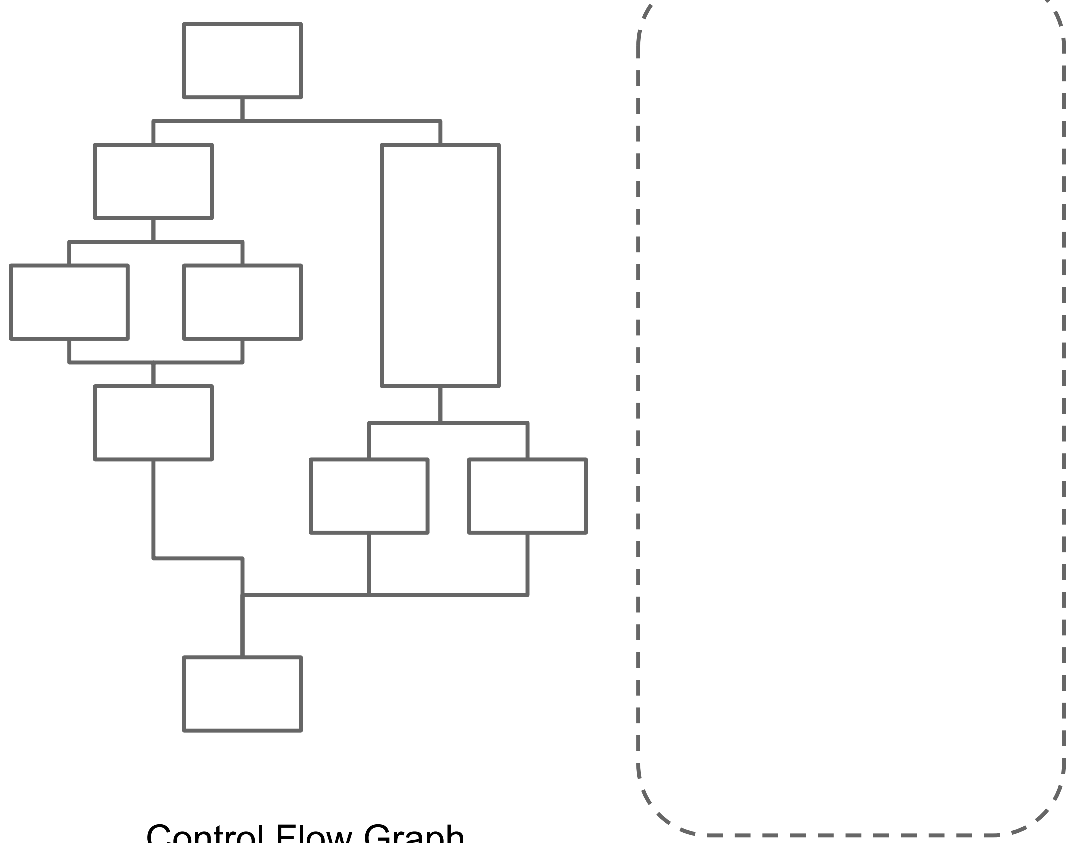
```
x = input()

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!”
```

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

Fuzzing Wins

Symbolic Execution Wins

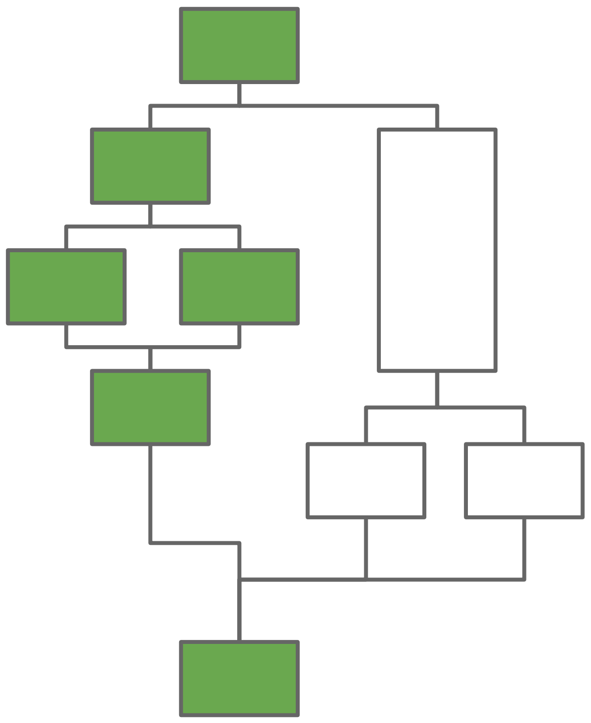
Combining the Two



Control Flow Graph

Combining the Two

“Cheap” fuzzing coverage



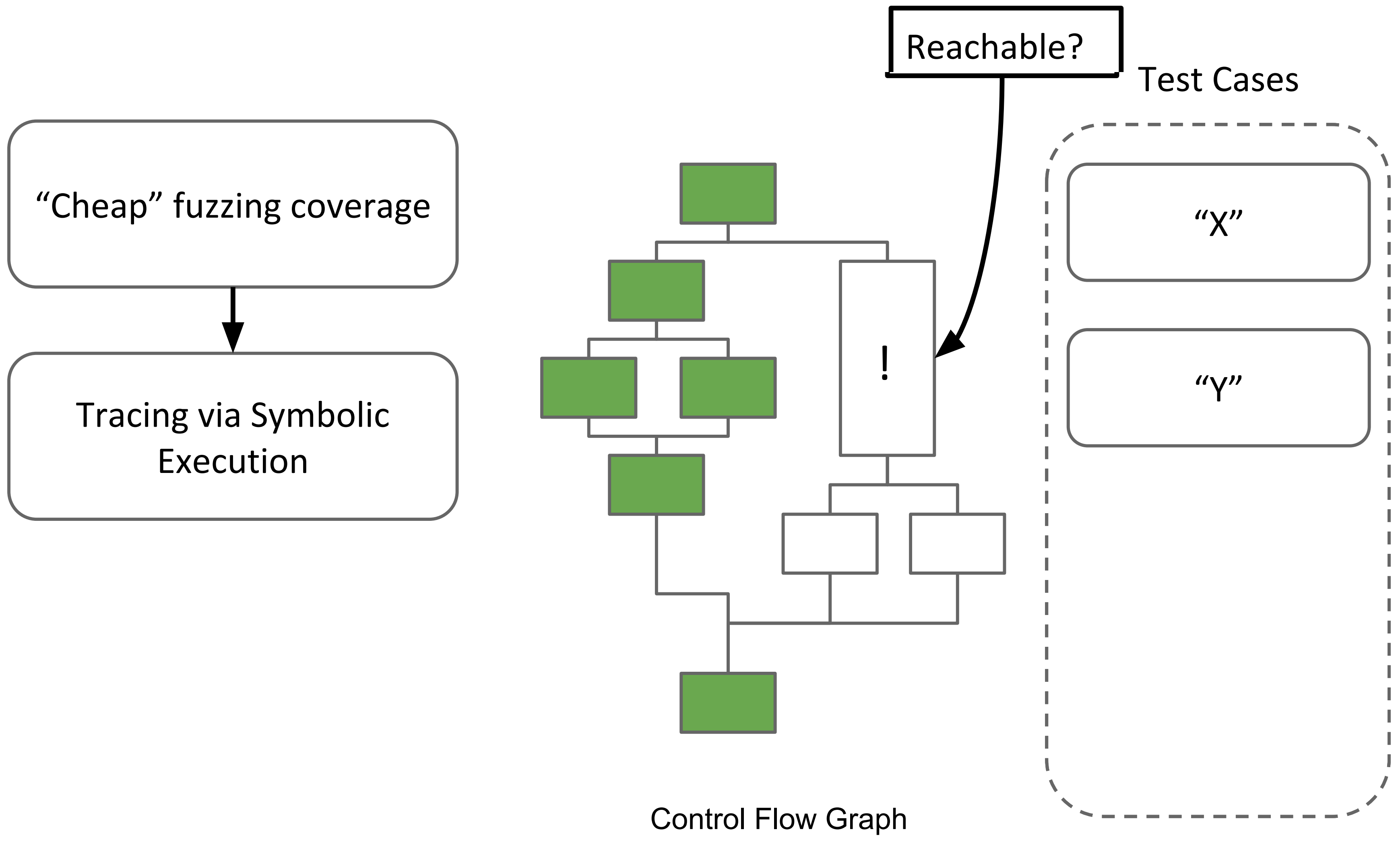
Control Flow Graph

Test Cases

“X”

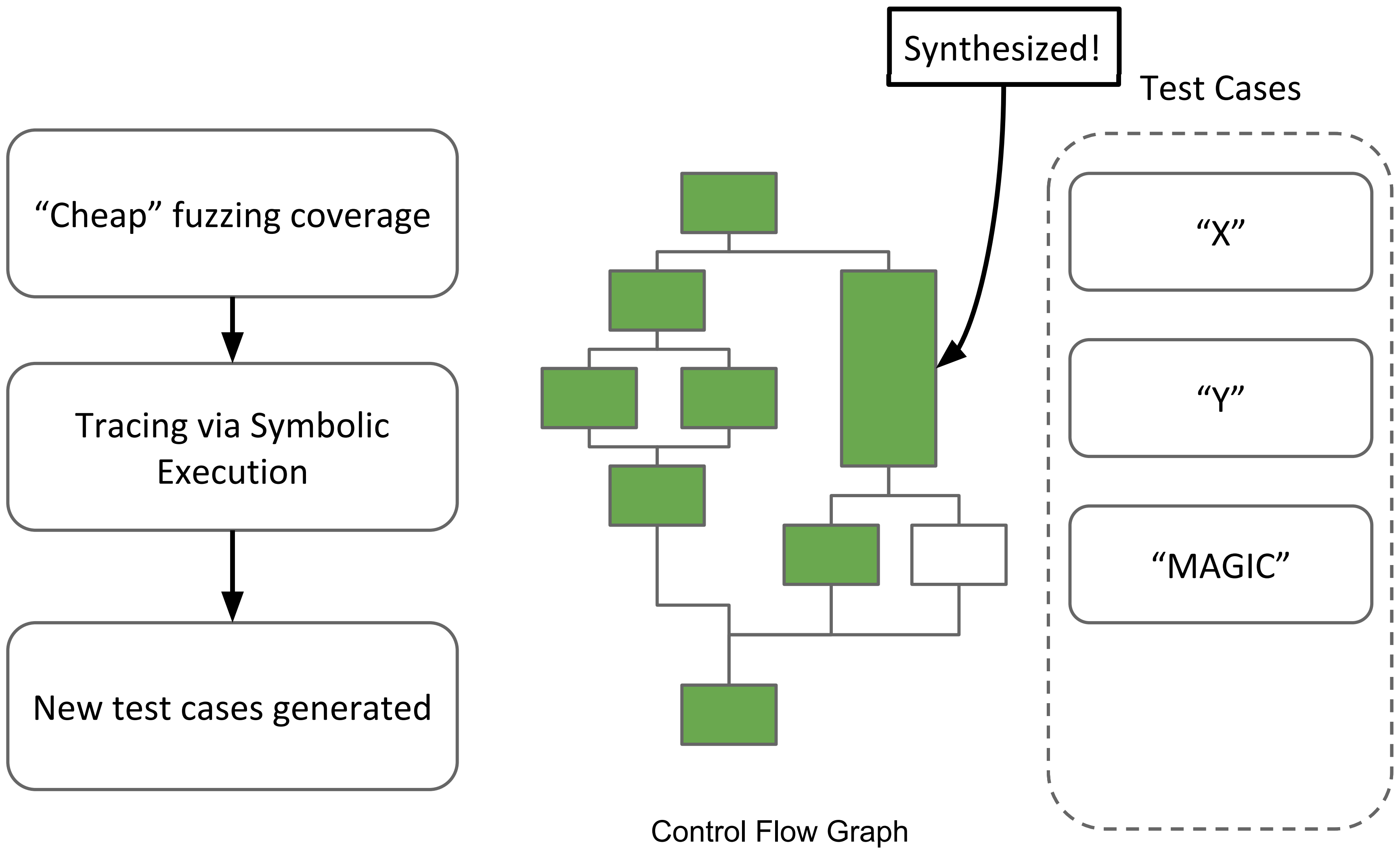
“Y”

Combining the Two



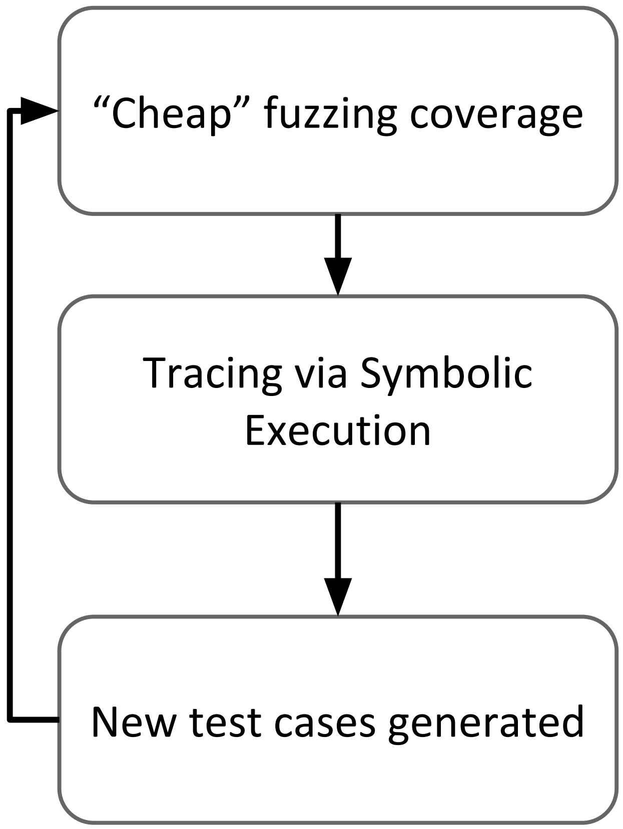
Control Flow Graph

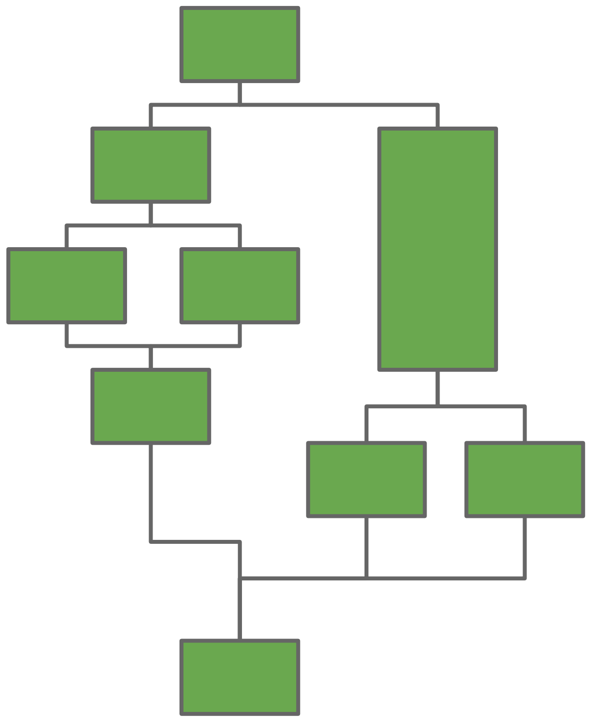
Combining the Two



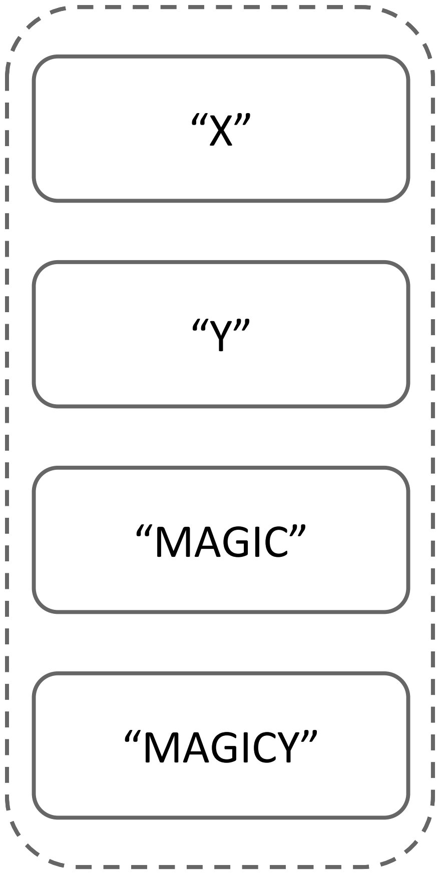
Combining the Two

Towards completer code coverage!



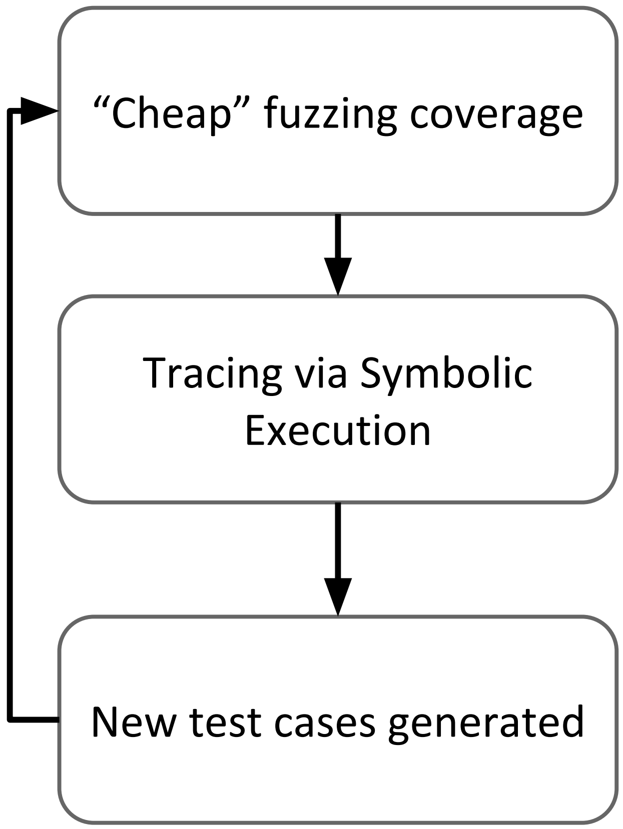

Control Flow Graph

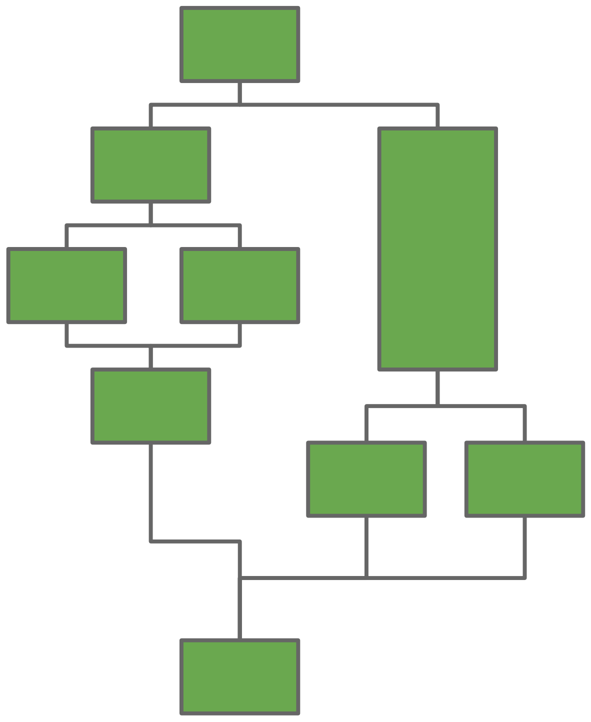
Test Cases



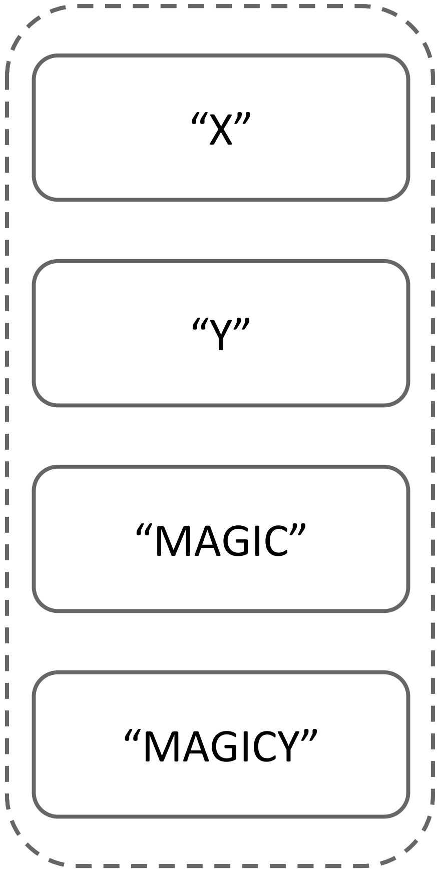
Combining the Two

Towards completer code coverage!




Control Flow Graph

Test Cases



Take Away

- Symbolic Execution is a method for detecting software flaws that emulates execution of the program under (some) symbolic inputs
  - Like dynamic analysis (fuzzing)
    - On each conditional, collect constraints implied by conditional over the symbolic variables
  - Like static analysis
    - Collected constraints can be solved to determine a specific input values to reach a specific program statement
- Can be combined with fuzzing to enhance program coverage and can be supplemented by static analysis