Teaching with angr: A Symbolic Execution Curriculum and CTF

Jacob M. Springer[1,2], Wu-chang Feng[1]

[1]Portland State University
[2]Swarthmore College

Outline

- What is symbolic execution?
- How do we teach it?

Symbolic execution: why should you care?

Symbolic execution: why should you care?

- Program analysis and testing

Symbolic execution: why should you care?

- Program analysis and testing
- Microsoft applications (PowerPoint, Word, etc.)

Symbolic execution: why should you care?

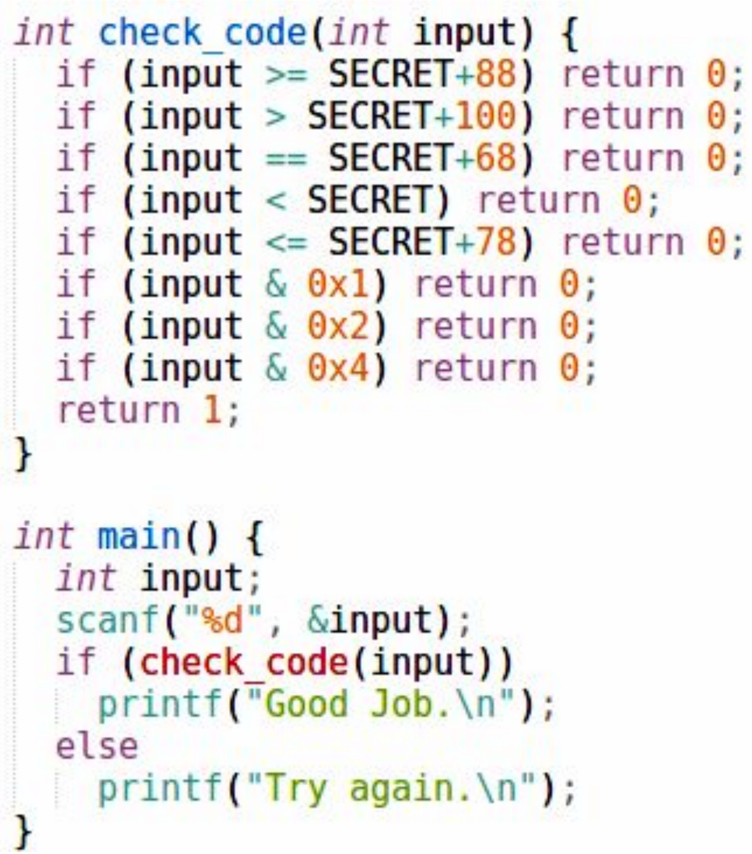
- Program analysis and testing
- Microsoft applications (PowerPoint, Word, etc.)
- DARPA's Cyber-Grand Challenge

Symbolic execution: why should you care?

- Program analysis and testing
- Microsoft applications (PowerPoint, Word, etc.)
- DARPA's Cyber-Grand Challenge
- Important for students to understand and apply

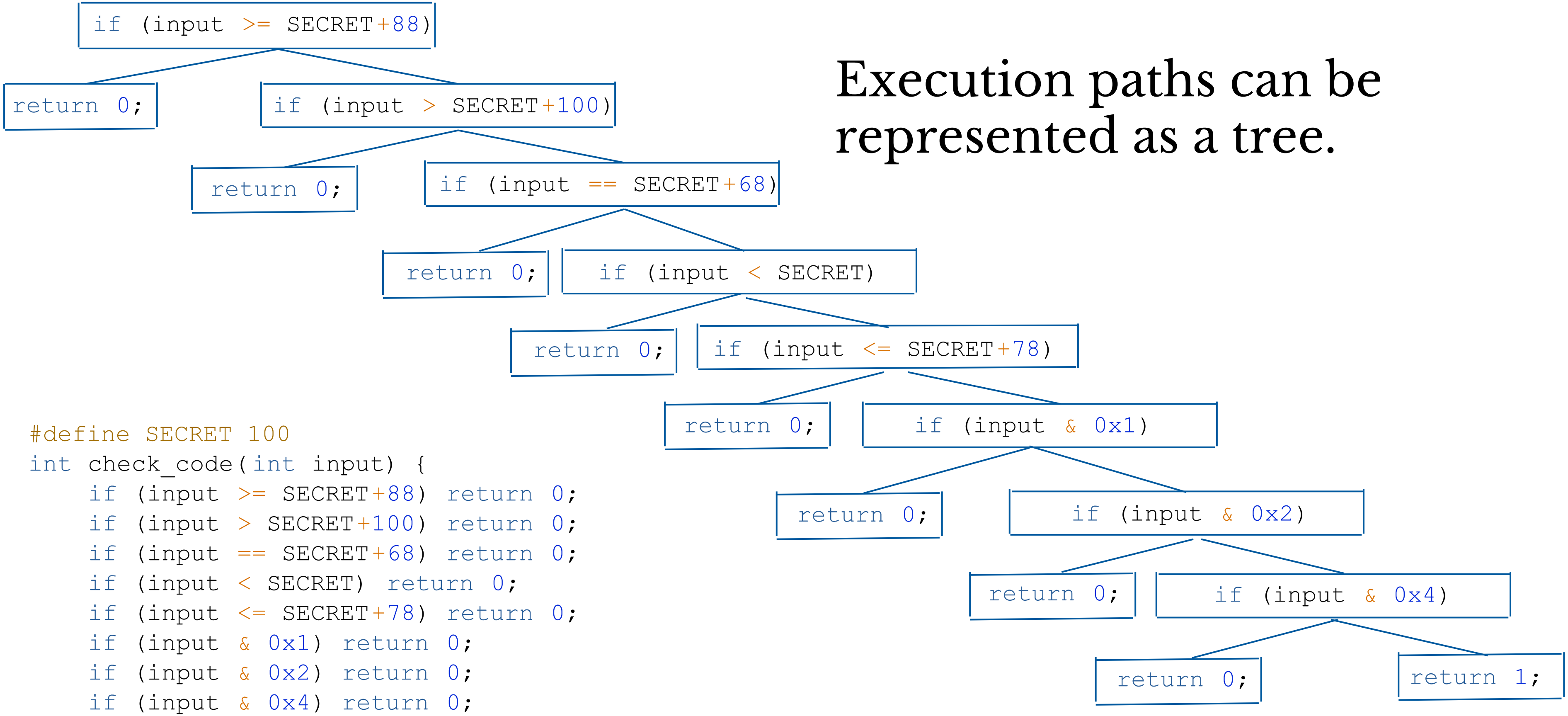
What is symbolic execution?

Find input to print “Good Job.”



```
int check_code(int input) {
  if (input >= SECRET+88) return 0;
  if (input > SECRET+100) return 0;
  if (input == SECRET+68) return 0;
  if (input < SECRET) return 0;
  if (input <= SECRET+78) return 0;
  if (input & 0x1) return 0;
  if (input & 0x2) return 0;
  if (input & 0x4) return 0;
  return 1;
}

int main() {
  int input;
  scanf("%d", &input);
  if (check_code(input))
    printf("Good Job.\n");
  else
    printf("Try again.\n");
}
```

Execution paths can be represented as a tree.


if (input >= SECRET+88)

return 0;

if (input > SECRET+100)

return 0;

if (input == SECRET+68)

return 0;

if (input < SECRET)

return 0;

if (input <= SECRET+78)

return 0;

if (input & 0x1)

return 0;

if (input & 0x2)

return 0;

if (input & 0x4)

return 0;

return 1;

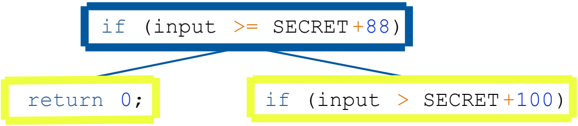
```
#define SECRET 100
int check_code(int input) {
    if (input >= SECRET+88) return 0;
    if (input > SECRET+100) return 0;
    if (input == SECRET+68) return 0;
    if (input < SECRET) return 0;
    if (input <= SECRET+78) return 0;
    if (input & 0x1) return 0;
    if (input & 0x2) return 0;
    if (input & 0x4) return 0;
    return 1;
}
```

Animation: Building a Set of Paths

```
if (input >= SECRET+88)
```

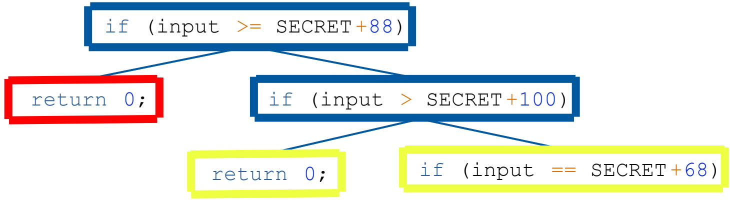
Legend:
Blue = already executed
Yellow = active
Red = terminated

Animation: Building a Set of Paths



Legend:
Blue = already executed
Yellow = active
Red = terminated

Animation: Building a Set of Paths



`if (input >= SECRET+88)`

`return 0;`

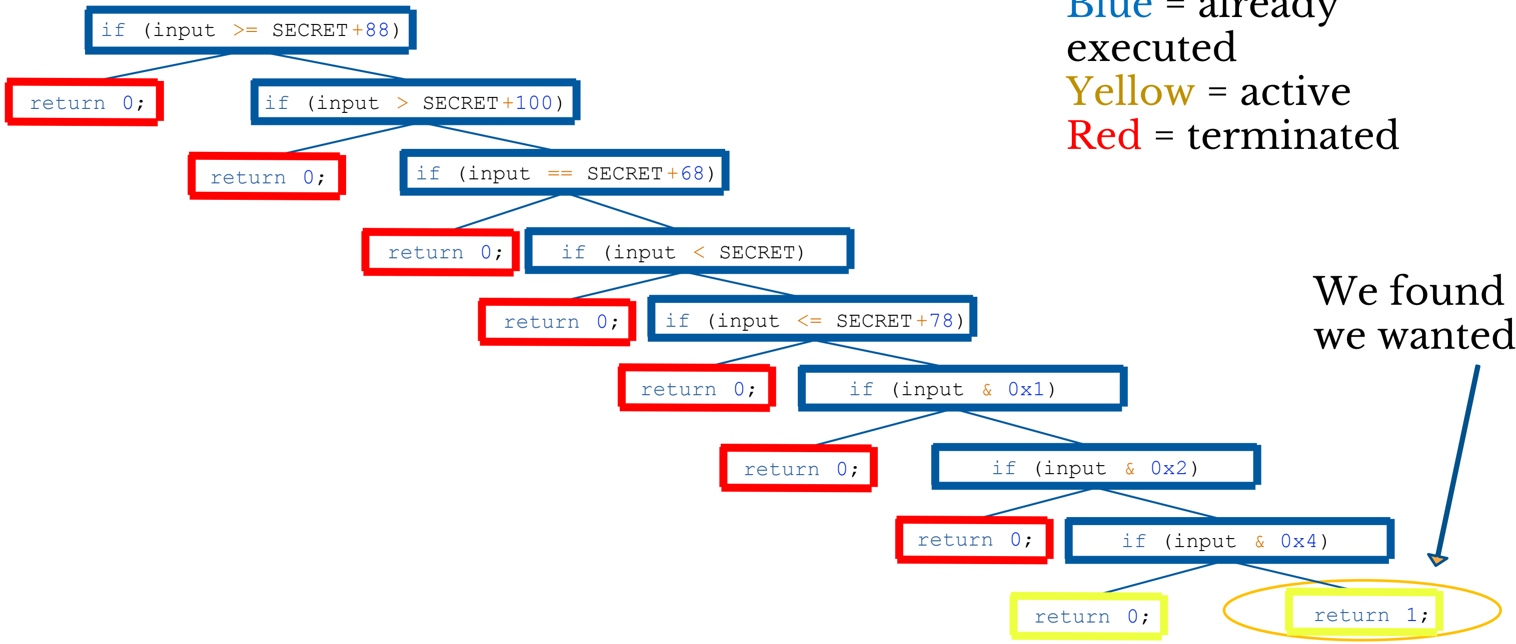
`if (input > SECRET+100)`

`return 0;`

`if (input == SECRET+68)`

Legend:
Blue = already executed
Yellow = active
Red = terminated

Animation: Building a Set of Paths



```
if (input >= SECRET+88)
return 0;
if (input > SECRET+100)
return 0;
if (input == SECRET+68)
return 0;
if (input < SECRET)
return 0;
if (input <= SECRET+78)
return 0;
if (input & 0x1)
return 0;
if (input & 0x2)
return 0;
if (input & 0x4)
return 0;
return 1;
```

Legend:
Blue = already executed
Yellow = active
Red = terminated

We found what we wanted!

Applying symbolic execution

```
if (input >= SECRET+88)
if (input > SECRET+100)
if (input == SECRET+68)
if (input < SECRET)
if (input <= SECRET+78)
if (input & 0x1)
if (input & 0x2)
if (input & 0x4)
return 1;
```

Once we have a path, we can build an equation that can be solved by the computer:

```
  input >= SECRET+88
∧ input > SECRET+100
∧ input == SECRET+68
∧ input < SECRET
∧ input <= SECRET+78
∧ input & 0x1
∧ input & 0x2
∧ input & 0x4
```

Angr-y CTF

Goal: Build a curriculum and a set of capture-the-flag (CTF) levels to introduce students to symbolic execution

Our Approach

Our Approach

Modeled after MetaCTF (USENIX 3GSE 2015)

- Find a password that causes a program to print "Good Job."

Our Approach

Modeled after MetaCTF (USENIX 3GSE 2015)

- Find a password that causes a program to print "Good Job."

18 scaffolded levels

- Requires symbolic execution to solve

Our Approach

Modeled after MetaCTF (USENIX 3GSE 2015)

- Find a password that causes a program to print "Good Job."

18 scaffolded levels

- Requires symbolic execution to solve

Uses angr (angr.io)

A typical level

A typical level

- Student receives a binary and a template angr script

A typical level

- Student receives a binary and a template angr script
- Student edits the template to analyze the binary

A typical level

- Student receives a binary and a template angr script
- Student edits the template to analyze the binary
- Student runs the script which prints a password

A typical level

- Student receives a binary and a template angr script
- Student edits the template to analyze the binary
- Student runs the script which prints a password
- Student runs the binary and types in the password to confirm their work

The levels are scaffolded

What does scaffolding mean?

- Support structure, just like a scaffold
- Guided, incremental introduction of concepts

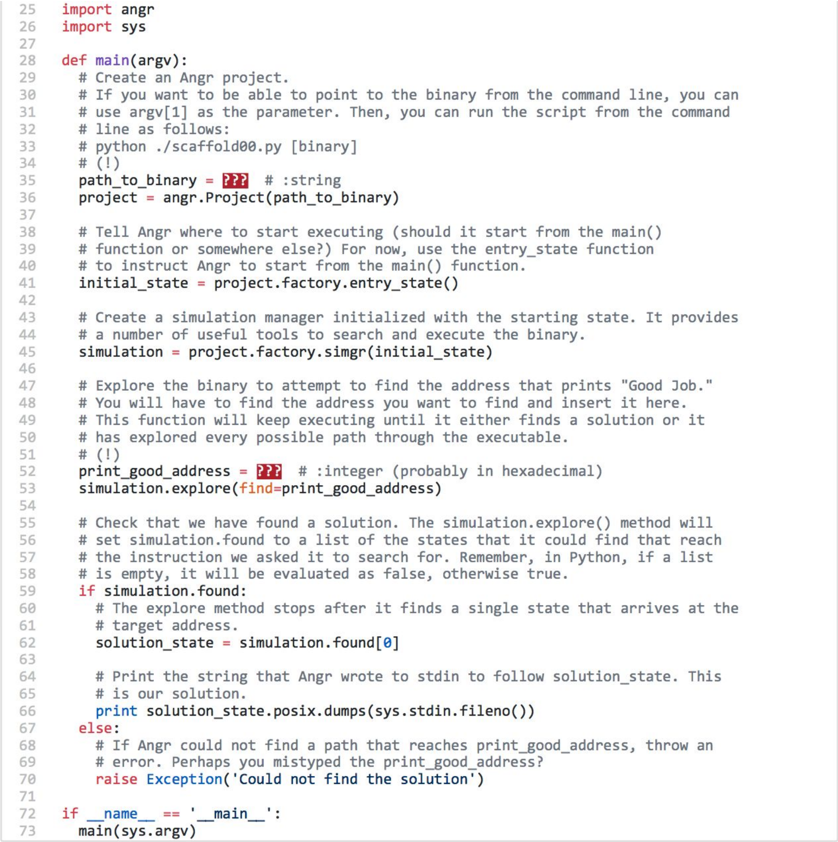
00_angr_find
01_angr_avoid
02_angr_find_condition
03_angr_symbolic_regi...
04_angr_symbolic_stack
05_angr_symbolic_me...
06_angr_symbolic_dyn...
07_angr_symbolic_file
08_angr_constraints
09_angr_hooks
10_angr_simprocedures
11_angr_sim_scanf
12_angr_veritesting
13_angr_static_binary
14_angr_shared_library
15_angr_arbitrary_read
16_angr_arbitrary_write
17_angr_arbitrary_jump

CTF Modules

- Basic symbolic execution
- Symbol injection
- Handling complexity
- Automated exploitation

Scaffolding for pedagogy: not frustrating

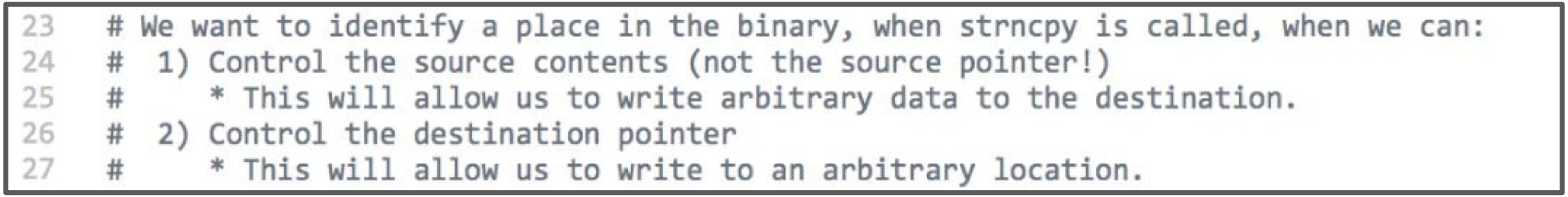
- Level 1
- Well documented
- Only need to change two lines



```python
import angr
import sys

def main(argv):
  # Create an Angr project.
  # If you want to be able to point to the binary from the command line, you can
  # use argv[1] as the parameter. Then, you can run the script from the command
  # line as follows:
  # python ./scaffold00.py [binary]
  # (!)
  path_to_binary = ???  # :string
  project = angr.Project(path_to_binary)

  # Tell Angr where to start executing (should it start from the main()
  # function or somewhere else?) For now, use the entry_state function
  # to instruct Angr to start from the main() function.
  initial_state = project.factory.entry_state()

  # Create a simulation manager initialized with the starting state. It provides
  # a number of useful tools to search and execute the binary.
  simulation = project.factory.simgr(initial_state)

  # Explore the binary to attempt to find the address that prints "Good Job."
  # You will have to find the address you want to find and insert it here.
  # This function will keep executing until it either finds a solution or it
  # has explored every possible path through the executable.
  # (!)
  print_good_address = ???  # :integer (probably in hexadecimal)
  simulation.explore(find=print_good_address)

  # Check that we have found a solution. The simulation.explore() method will
  # set simulation.found to a list of the states that it could find that reach
  # the instruction we asked it to search for. Remember, in Python, if a list
  # is empty, it will be evaluated as false, otherwise true.
  if simulation.found:
    # The explore method stops after it finds a single state that arrives at the
    # target address.
    solution_state = simulation.found[0]

    # Print the string that Angr wrote to stdin to follow solution_state. This
    # is our solution.
    print solution_state.posix.dumps(sys.stdin.fileno())
  else:
    # If Angr could not find a path that reaches print_good_address, throw an
    # error. Perhaps you mistyped the print_good_address?
    raise Exception('Could not find the solution')

if __name__ == '__main__':
  main(sys.argv)
```

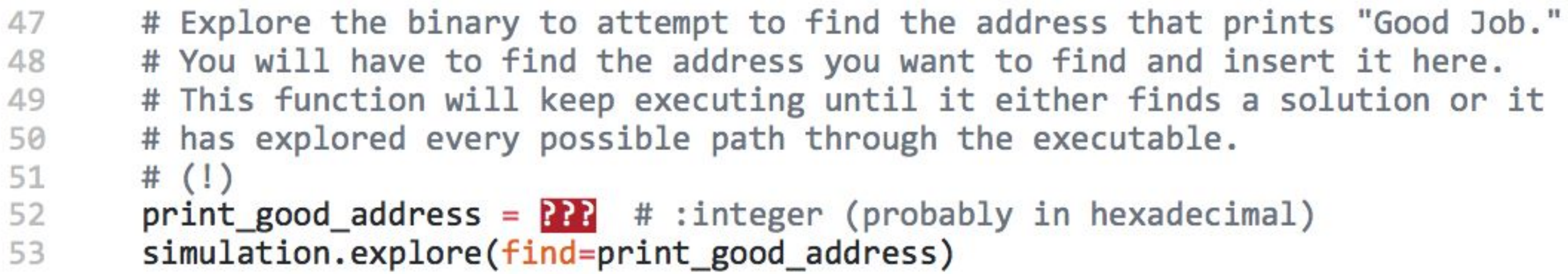
Scaffolding for pedagogy: guided

- Tells student how to get started



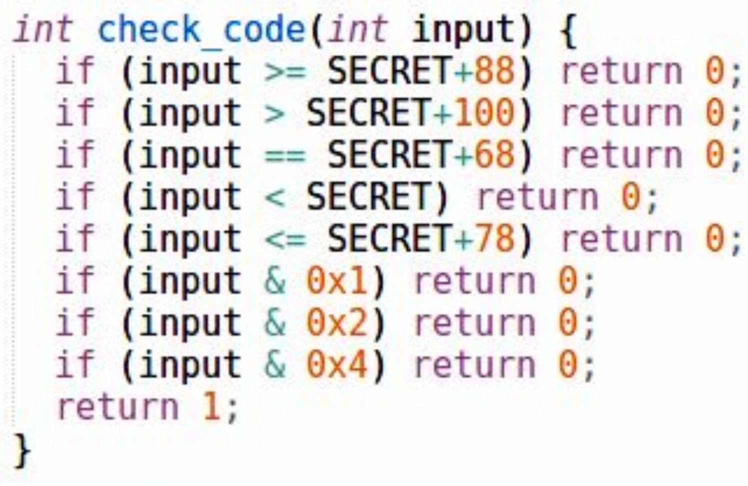
```
# We want to identify a place in the binary, when strncpy is called, when we can:
#  1) Control the source contents (not the source pointer!)
#     * This will allow us to write arbitrary data to the destination.
#  2) Control the destination pointer
#     * This will allow us to write to an arbitrary location.
```

Scaffolding: simple



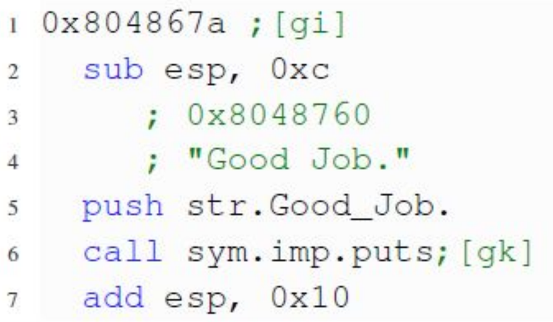
```
# Explore the binary to attempt to find the address that prints "Good Job."
# You will have to find the address you want to find and insert it here.
# This function will keep executing until it either finds a solution or it
# has explored every possible path through the executable.
# (!)
print_good_address = ???  # :integer (probably in hexadecimal)
simulation.explore(find=print_good_address)
```

MetaCTF Example



```
int check_code(int input) {
  if (input >= SECRET+88) return 0;
  if (input > SECRET+100) return 0;
  if (input == SECRET+68) return 0;
  if (input < SECRET) return 0;
  if (input <= SECRET+78) return 0;
  if (input & 0x1) return 0;
  if (input & 0x2) return 0;
  if (input & 0x4) return 0;
  return 1;
}
```

```
int main() {
  int input;
  scanf("%d", &input);
  if (check_code(input))
    printf("Good Job.\n");
  else
    printf("Try again.\n");
}
```



```
0x804867a ;[gi]
  sub esp, 0xc
      ; 0x8048760
      ; "Good Job."
  push str.Good_Job.
  call sym.imp.puts;[gk]
  add esp, 0x10
```

Scaffolding: builds on previous concepts

```
0x804867a ;[gi]
  sub esp, 0xc
      ; 0x8048760
      ; "Good Job."
  push str.Good_Job.
  call sym.imp.puts;[gk]
  add esp, 0x10
```

```
  # Explore the binary to attempt to find the address that prints "Good Job."
  # You will have to find the address you want to find and insert it here.
  # This function will keep executing until it either finds a solution or it
  # has explored every possible path through the executable.
  # (!)
  print_good_address = 0x804867a  # :integer (probably in hexadecimal)
  simulation.explore(find=print_good_address)
```

Scaffolding: incremental and reinforcing

- Level 02 (find_condition)

- Level 03 (symbolic_registers)

Scaffolding: incremental and reinforcing

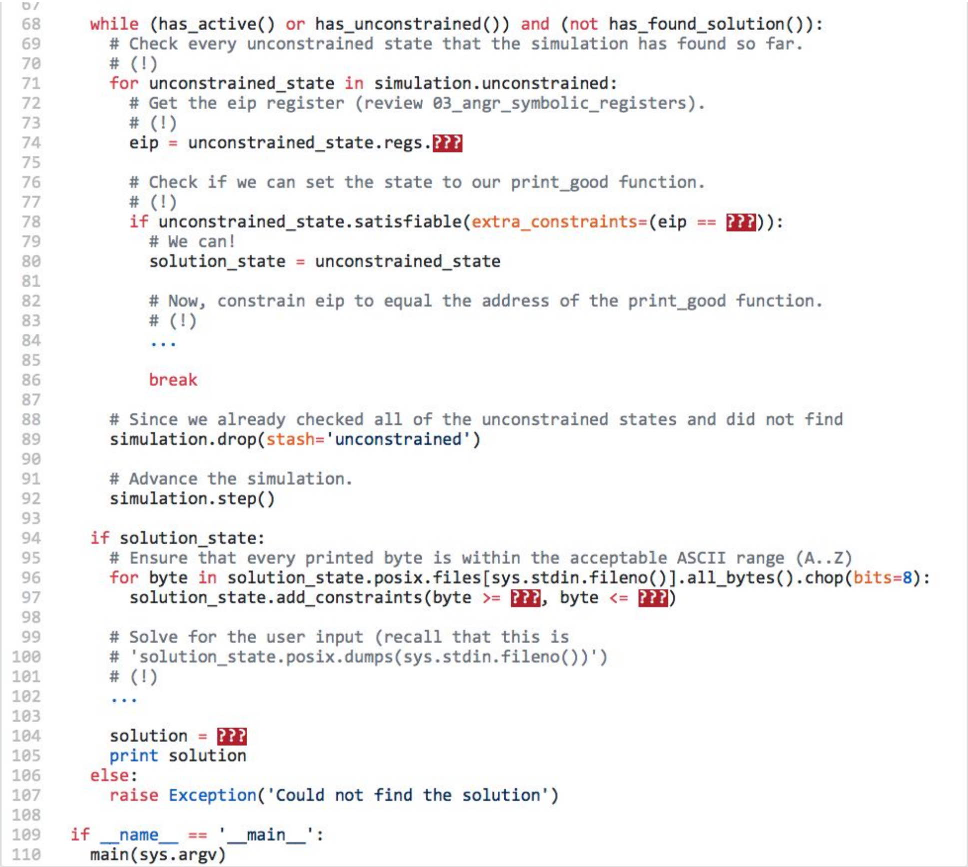
- Level 02 (find_condition)
    - 1. Load binary
    - 2. Define the termination condition (Has the program printed “Good Job.”?)
    - 3. Search binary for condition
- Level 03 (symbolic_registers)

Scaffolding: incremental and reinforcing

- Level 02 (find_condition)
    - 1. Load binary
    - 2. Define the termination condition (Has the program printed “Good Job.”?)
    - 3. Search binary for condition
- Level 03 (symbolic_registers)
    - 1. Load binary
    - 2. Inject symbols
    - 3. Define the termination condition (Has the program printed “Good Job.”?)
    - 4. Search binary for condition

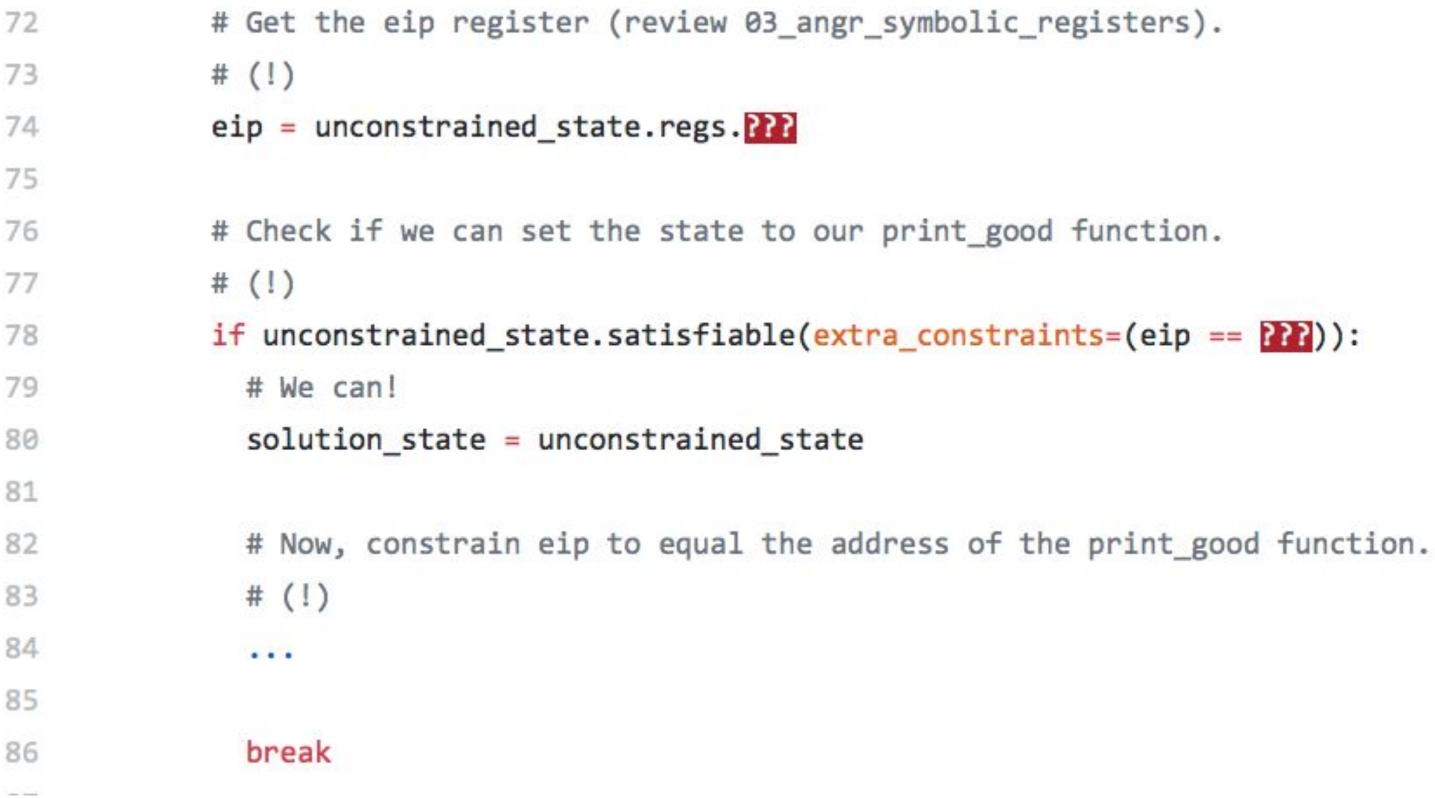
Scaffolding: conceptual

- First glance:
  seems complicated



```
  while (has_active() or has_unconstrained()) and (not has_found_solution()):
    # Check every unconstrained state that the simulation has found so far.
    # (!)
    for unconstrained_state in simulation.unconstrained:
      # Get the eip register (review 03_angr_symbolic_registers).
      # (!)
      eip = unconstrained_state.regs.???

      # Check if we can set the state to our print_good function.
      # (!)
      if unconstrained_state.satisfiable(extra_constraints=(eip == ???)):
        # We can!
        solution_state = unconstrained_state

        # Now, constrain eip to equal the address of the print_good function.
        # (!)
        ...

        break

    # Since we already checked all of the unconstrained states and did not find
    simulation.drop(stash='unconstrained')

    # Advance the simulation.
    simulation.step()

  if solution_state:
    # Ensure that every printed byte is within the acceptable ASCII range (A..Z)
    for byte in solution_state.posix.files[sys.stdin.fileno()].all_bytes().chop(bits=8):
      solution_state.add_constraints(byte >= ???, byte <= ???)

    # Solve for the user input (recall that this is
    # 'solution_state.posix.dumps(sys.stdin.fileno())')
    # (!)
    ...

    solution = ???
    print solution
  else:
    raise Exception('Could not find the solution')

if __name__ == '__main__':
  main(sys.argv)
```

Scaffolding: conceptual, part 2



```
# Get the eip register (review 03_angr_symbolic_registers).
# (!)
eip = unconstrained_state.regs.???

# Check if we can set the state to our print_good function.
# (!)
if unconstrained_state.satisfiable(extra_constraints=(eip == ???)):
  # We can!
  solution_state = unconstrained_state

  # Now, constrain eip to equal the address of the print_good function.
  # (!)
  ...

  break
```

What does metamorphic mean?

- Different `SECRET` for every student
- Can generate arbitrary C code

```c
int check_code(int input) {
  if (input >= SECRET+88) return 0;
  if (input > SECRET+100) return 0;
  if (input == SECRET+68) return 0;
  if (input < SECRET) return 0;
  if (input <= SECRET+78) return 0;
  if (input & 0x1) return 0;
  if (input & 0x2) return 0;
  if (input & 0x4) return 0;
  return 1;
}

int main() {
  int input;
  scanf("%d", &input);
  if (check_code(input))
    printf("Good Job.\n");
  else
    printf("Try again.\n");
}
```

Metamorphic levels

- Reduce cheating
- Allow reuse
- Maintain consistency of difficulty across students

Evaluation

- Offered Winter 2018 in Portland State University's CS 492/592: Malware course
  - Last 2 weeks focused on symbolic execution
- Survey given at the end of two weeks
  - 33 of 42 responded

Results

Curriculum and scaffolding allow students to complete most levels

Completion percentage	Number of students
95-100%	25
85-95%	4
75-85%	6
Below 75%	7

Survey

- Ratings evaluate helpfulness of curriculum and CTF
    - Very Unhelpful = 1
    - Very Helpful = 5
- Q1: Rate the lecture material for understanding the concepts

Rating	1	2	3	4	5	Mean
Q1	1	1	2	17	12	4.15

- Q2: Rate the CTF exercises for understanding the concepts

Rating	1	2	3	4	5	Mean
Q2	2	1	3	18	9	3.94

Survey

- Q3: Rate the CTF exercises for developing skills in using symbolic execution techniques

Rating	1	2	3	4	5	Mean
Q3	1	3	3	16	10	3.94

Try the CTF!

https://malware.oregonctf.org

Also on GitHub http://github.com/jakespringer/angr_ctf