Symbolic execution

Portland State
Computer Science

Symbolic execution

- Workhorse for modern program analysis and testing
  - Changing the way we test and analyze code
- Focus is on executing all code paths through a program
  - Potentially test an infinite number of input cases
  - More effective than brute-force fuzzing or test case generation
  - Enumerates constraints on inputs that lead to specific program states
- Informally
  - Algebra for your programs
    - Find all "x" as program input that can lead you to state "y" in your program
  - CTFs for the lazy
    - What input will
      - Cause my program to print "Good Job." (CS 201, 492/592)
      - The level flag (RE CTFs)
      - Or get me a contract's $ (this class)?

Why?

- One poster child

Chinese bank's software chief jailed after finding way to withdraw US$1m in 'free' cash from ATMs

- Flaw in system meant that withdrawals made around midnight were not recorded in the system

  According to the reports, the bank's system didn't properly record withdrawals made around midnight — effectively spitting out cash without removing the total from a user's account.

- Symbolic execution can ensure this can not happen
- Manticore scripts in labs attempt to do this for Ethereum contracts

Going mainstream

- AppStore (iOS) requires apps to compile to LLVM bytecode
  - For automated analysis via fuzzers and symbolic execution
- Incorprated into the testing of all Microsoft OSes and Office applications since 2007
  - SAGE tool
  - Spun out into Project Springfield
  - Spun out into Microsoft Security Risk Detection service
  - Morphed into Azure cloud service for automated analysis
- Now an industry based around formally verifying smart contracts
  - Tools: Manticore, Mythril, Oyente, etc.
  - Services: DecentralStation.com, Trail of Bits

Toy example

- Consider this program

You are here

You want to find an input that arrives here

```
1 user_input = raw_input('Enter the password: ')
2 if user_input == '...':
3   print 'Good Job.'
4 else:
5   print 'Try again.'
```

Step 1: Inject a Symbol

- Similar to variables in Algebra

$$x^2 + 2x + 3 = 4$$

  - Variable x is a number in equation whose value is unknown
  - Don't know x but can solve for it based on equation that constrains it
- Symbolic execution
  - Start with

```
1 user_input = raw_input('Enter the password: ')
2 if user_input == '...':
3   print 'Good Job.'
4 else:
5   print 'Try again.'
```

Inject symbol.

```
1 user_input = λ
2 if user_input == '...':
3   print 'Good Job.'
4 else:
5   print 'Try again.'
```

  - Symbol λ is like x, but it's a variable in the program whose desired value is unknown
  - Don't know what λ is but can solve for it based on the execution paths that constrain it

What is an execution path?

- A possible way to travel through the program

```
1 user_input = λ
2 if user_input == '...':
3   print 'Good Job.'
4 else:
5   print 'Try again.'
```

  - …has two possible execution paths.
- Symbolic execution engine performs execution using injected symbol λ for `user_input`
  - Attempts to find an execution path that reaches line 3, then solves for symbol λ.
- How?

Step 2: Branch

- When execution reaches an if statement that depends on a symbol, execution engine branches
  - Split into two different possible execution paths based on conditional
  - Symbols updated with constraints the conditional branch imposes.

You are here 

```
1 user_input = λ
2 if user_input == 'hunter2':
3   print 'Good Job.'
4 else:
5   print 'Try again.'
```

Path 1: λ equals 'hunter2'

```
user_input = λ
        ⬇
print 'Good Job.'
```

Path 2: λ does not equal 'hunter2'

```
user_input = λ
        ⬇
print 'Try again.'
```

Step 3: Evaluate each branch

- Imagine the engine picked the `else` path first with the constraint that `(λ != “hunter2”)`.

```
1 user_input = λ
2 if user_input == 'hunter2':
3   print 'Good Job.'
4 else:
5   print 'Try again.'
```

You are here ➡ line 5

  - Reaches the end of the execution without finding what we wanted (e.g. Line 3)
  - Continue with the other branch (i.e. the running state on the other execution path)

- Execute the `(λ == "hunter2")` path.

You are here 

```
1 user_input = λ
2 if user_input == 'hunter2':
3   print 'Good Job.'
4 else:
5   print 'Try again.'
```

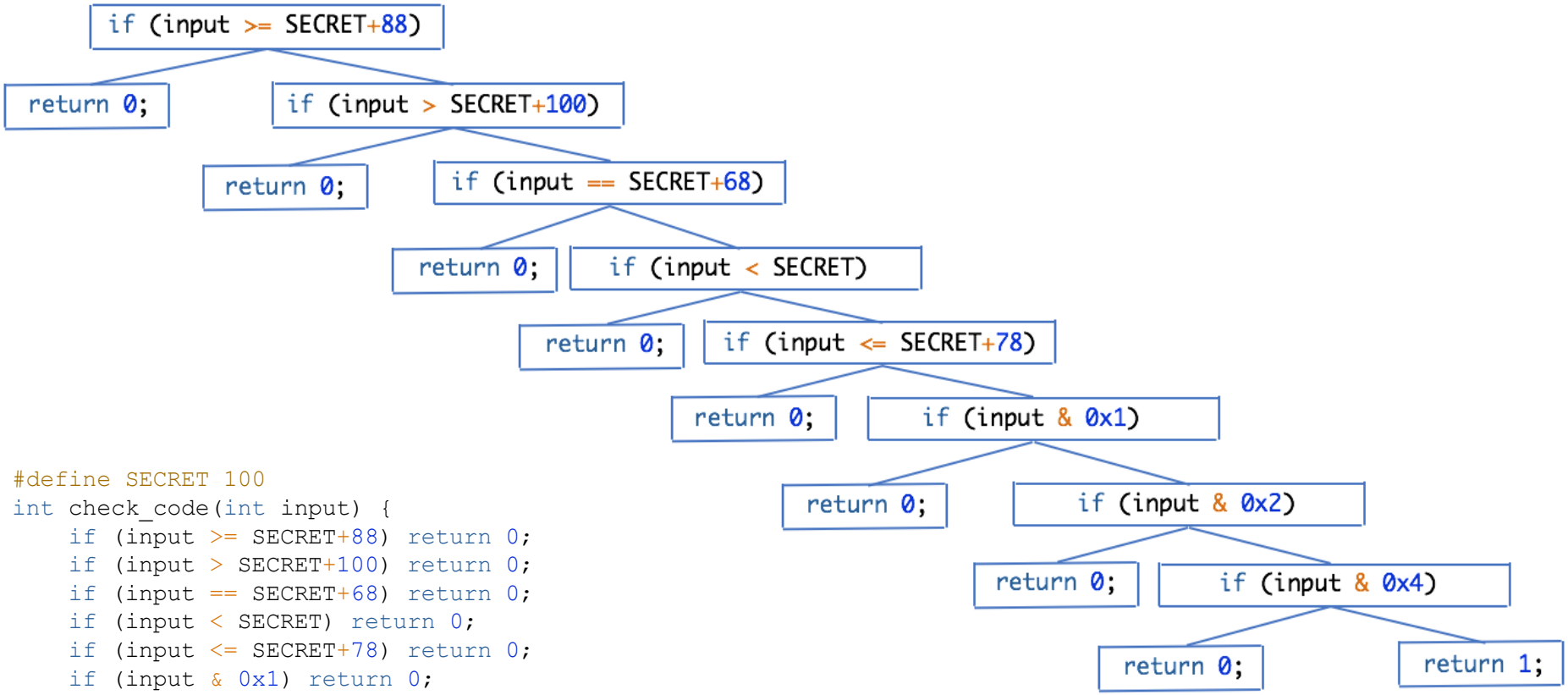
  - Successful execution path found!
  - Now, constrain the symbol to solve for a λ to find an actual input that reaches Line 3
    - Called "concretization"
      - In this example only 1 concrete solution exists
    - In general, many solutions can exist, consider
      `if 'foo' in user_input:`
      - Symbolic execution engine uses heuristics to concretize as many as you'd like

Solving a More Complex Example: Part 1

- From Ch06CAsm_Conditionals

```
#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;
}
```

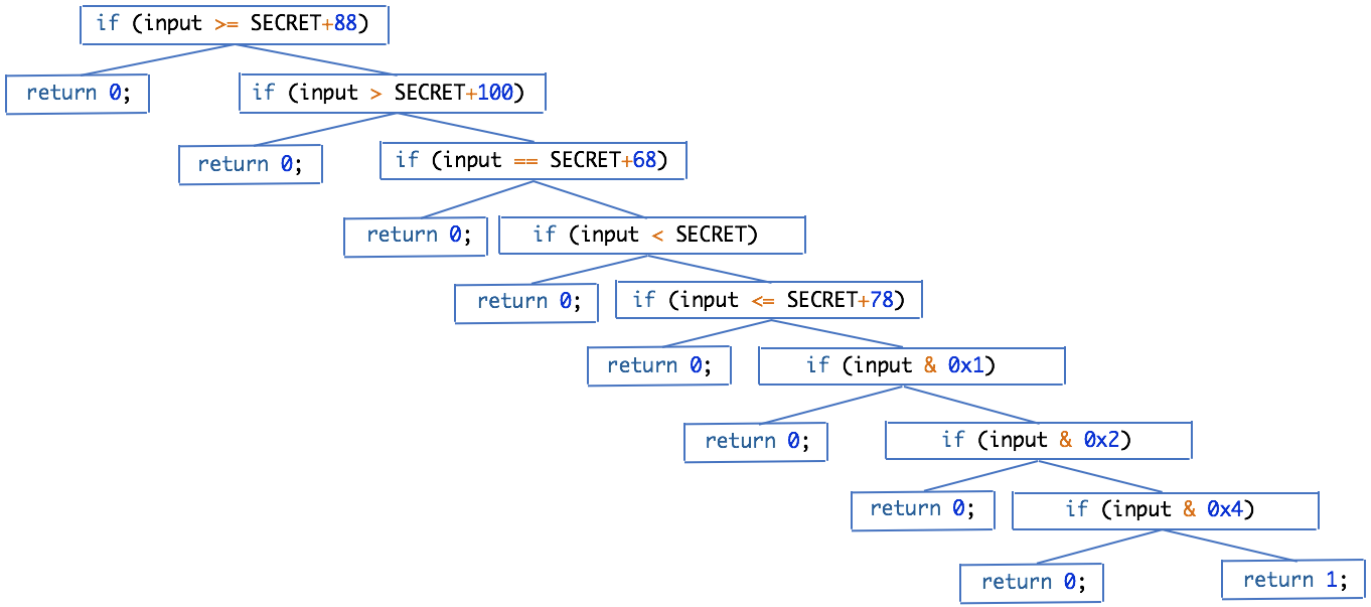
- Possible paths through the program
  - Can be represented as a tree




```
#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;
}
```

Symbolic execution

- Want to perform a tree search to find a path that returns 1.
- Engine steps through program to generate all possible execution paths
  - Path stores state of the program, as well as a history of the previous states that led to current state



Details

1. Engine starts the program at the program entry point (the function dispatcher).

2. Executes instructions in each running (nonterminated) state until a branching point is reached or the state terminates

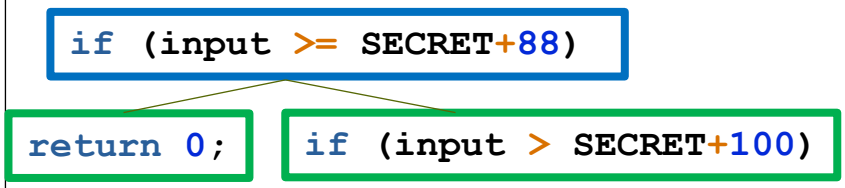
3. At every branching point, state is split into multiple states, and added to the set of running states

4. Steps 2..4 repeated until desired state is found or all states terminate

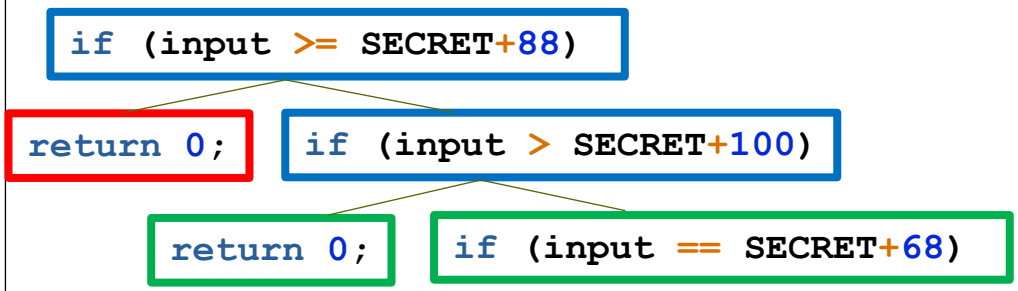
Animation

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

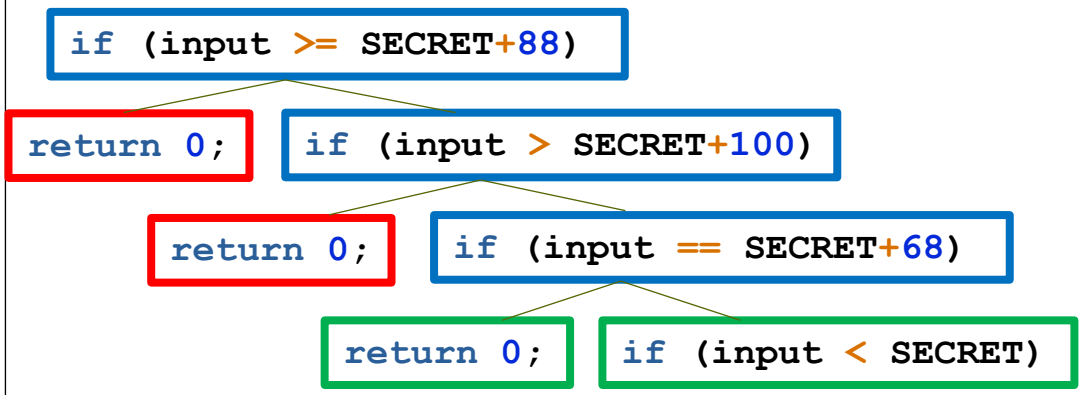
Legend:
Blue = already executed
Green = active
Red = terminated



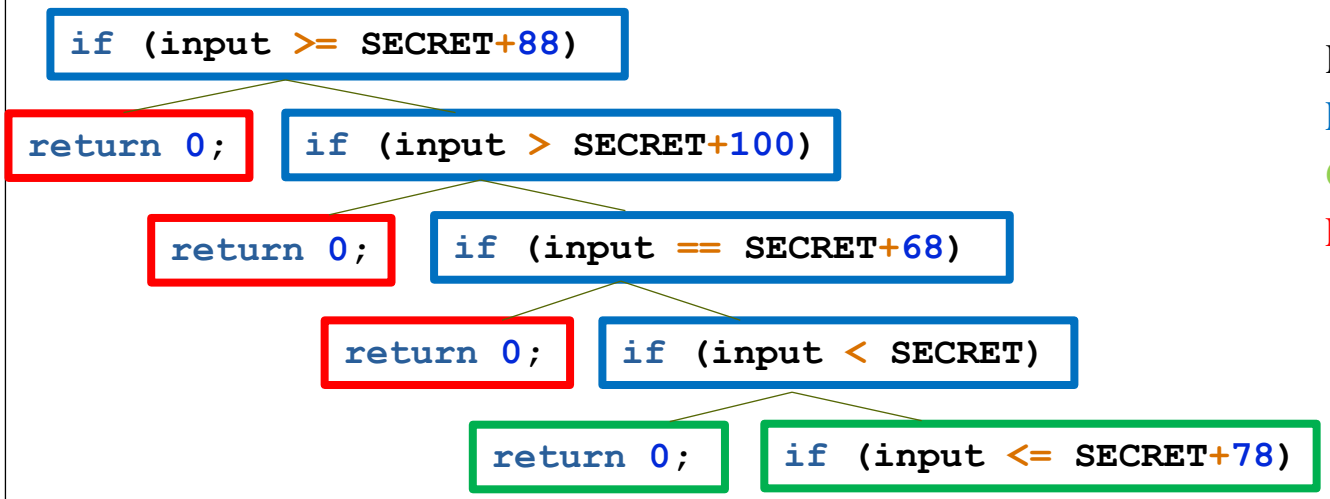
Legend:
Blue = already executed
Green = active
Red = terminated



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

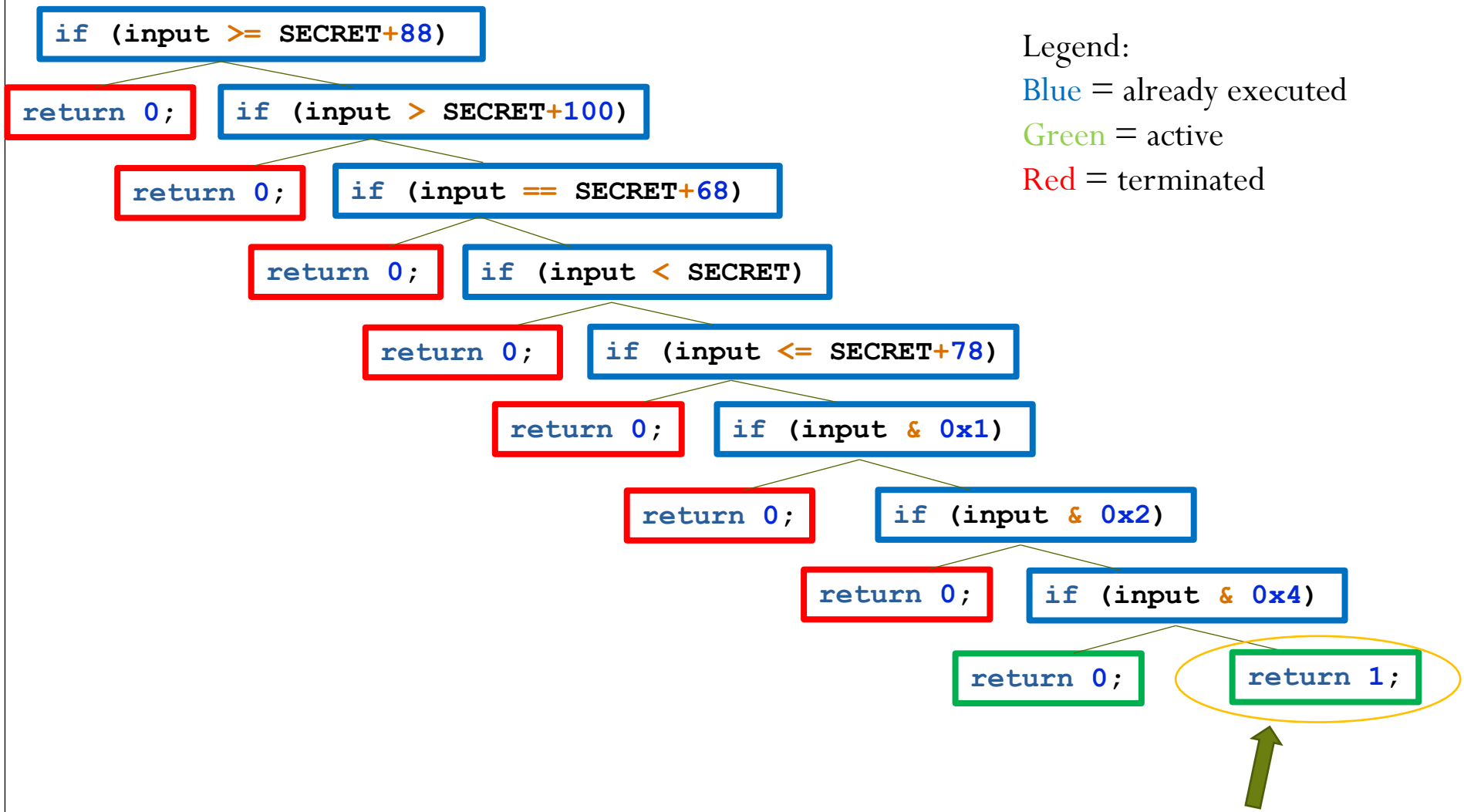


Legend:
Blue = already executed
Green = active
Red = terminated

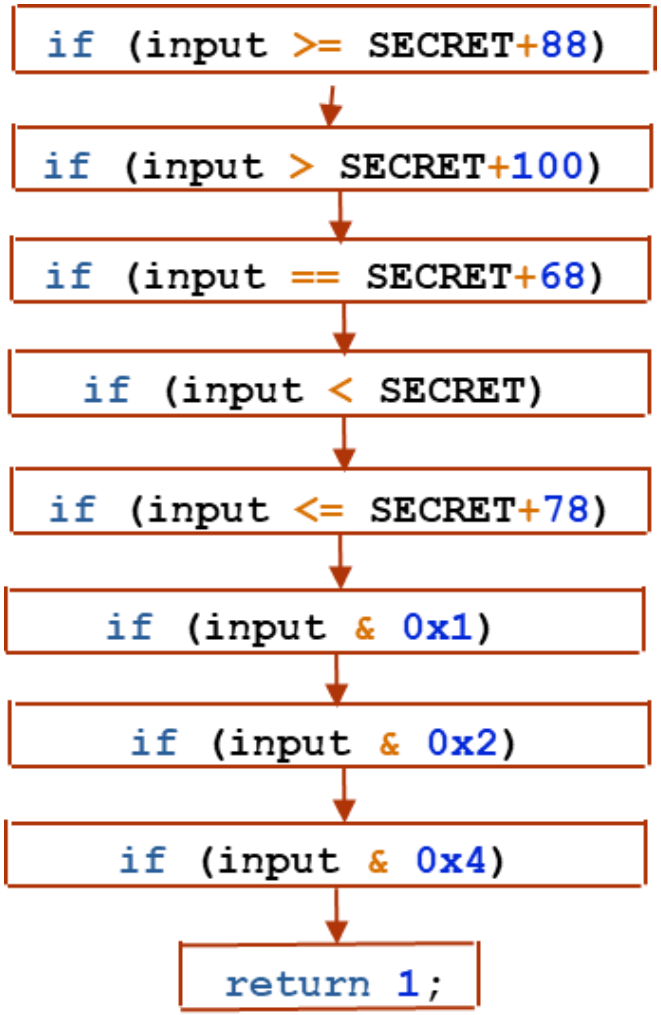


Legend:
Blue = already executed
Green = active
Red = terminated

. . . etc

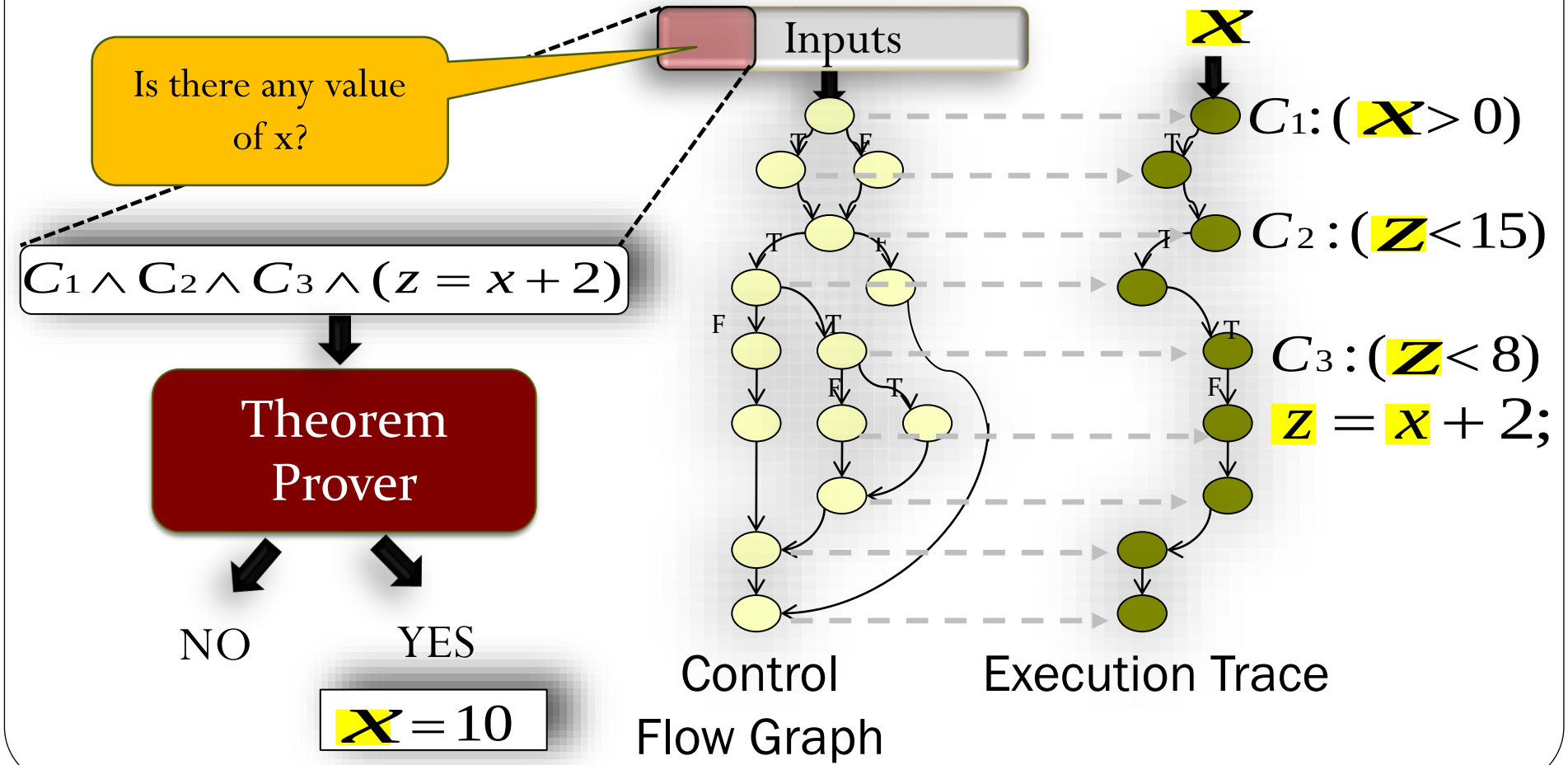

if (input >= SECRET+88)
return 0;
if (input > SECRET+100)
return 0;
if (input == SECRET+68)
return 0;
if (input < SECRET)
return 0;
if (input <= SECRET+78)
return 0;
if (input & 0x1)
return 0;
if (input & 0x2)
return 0;
if (input & 0x4)
return 0;
return 1;
Legend:
Blue = already executed
Green = active
Red = terminated
We found what we wanted!

Path found to get us what we want...

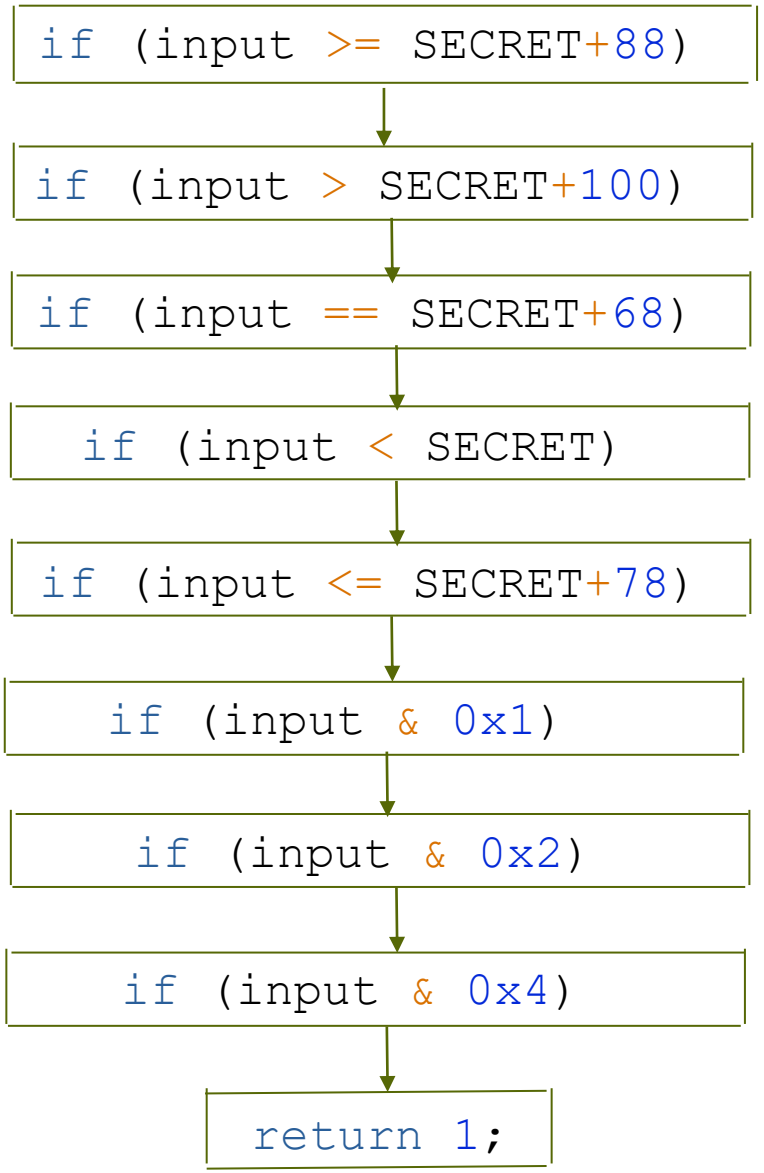


Solving a More Complex Example: Part 2

- Found a path that gets us the solution
  - Build an equation on inputs based on path's constraints
  - Send to a satisfiability modulo theories (SMT) solver to find solution
  - Solver returns no solution or a set of concrete inputs that solve constraints



Solving a More Complex Example: Part 2




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

Assuming SECRET is known,
SMT solver generates a concrete solution
if it exists

Manticore

Portland State
Computer Science

Manticore

- Open-source symbolic execution engine from Trail of Bits that can
  - Step through programs and follow any branch
  - Search for a program state that meets a given criteria
  - Solve for symbolic variables given path (and other) constraints
  - Written in Python and can operate on a variety of programs and binaries
- Others available including Oyente, Mythril

General usage

- Specify contracts and their parameters within Manticore Python script
- Specify end conditions desired
- Have Manticore solve for input
- This class
  - Walkthrough the mechanics of the technique
  - Use Manticore to automatically find solutions to CTF levels

Donation level

```
pragma solidity 0.4.24;
contract Donation {
    using SafeMath for uint256;
    uint256 public funds;

    constructor(address _ctfLauncher, address _player) public payable {
        funds = funds.add(msg.value);
    }

    function() external payable {
        funds = funds.add(msg.value);
    }

    function withdrawDonationsFromTheSuckersWhoFellForIt() external {
        msg.sender.transfer(funds);
        funds = 0;
    }
}
```

Have Manticore automatically generate the transaction that wipes out the contract's balance (via calculation of `msg.data`)

Solution script

```
# Import Manticore's EVM supporting symbolic execution
from manticore.ethereum import ManticoreEVM

# Parse arguments
#   arg1 = from_address = Your wallet address
from_address = int(sys.argv[1], 16) if len(sys.argv)>1 else "<your address here>"

#   arg2 = si_level_address = Your Donation CTF level address
si_level_address = int(sys.argv[2], 16) if len(sys.argv)>2 else "<SI ctf level address>"

#   arg3 = sol_file = Donation CTF level source code to symbolically execute
sol_file = sys.argv[3] if len(sys.argv)>3 else "../SI_ctf_levels/Donation.sol"


# Fix the amount of gas to use (can omit if you wish to rely on ManticoreEVM estimate)
gas = 100000

# Set the amount of ETH you want to obtain from the contract (0.05 ETH)
contract_balance = int(0.05 * 10**18)

# Read in the contract source
with open(sol_file, "r") as f:
    contract_src = f.read()

# Instantiate Manticore's Symbolic Ethereum Virtual Machine
m = ManticoreEVM()
```

```
# Create an account for your wallet address on the EVM.
# Give it enough to deploy vulnerable contract
#   (technically not what is done in real-life)
user_account = m.create_account(address=from_address, balance=contract_balance)

# Create the Donation CTF level contract on the EVM using wallet
#   contract_src = Prior source code
#   contract_name = Name of contract in source code
#   owner = Uses your wallet to deploy (OK for this level)
#   balance = Deploy with msg.value that the CTF level is deployed with
#   args = Arguments to deploy contract (null in this case)
contract_account = m.solidity_create_contract(
    contract_src,
    contract_name="Donation",
    owner=user_account,
    balance=contract_balance,
    args=(0,0)
)
# Ethereum contracts called via msg.data with 4 bytes of the keccak256 hash of the
#   function signature with whitespace removed (e.g. someFunction(uint256,uint256))
# Make symbolic buffer to hold msg.data and have Manticore calculate the "winning" value
sym_args = m.make_symbolic_buffer(4)

# Issue a symbolic transaction to the EVM by setting msg.data to symbolic buffer
m.transaction(
    caller=user_account,
    address=contract_account.address,
    data=sym_args,
    value=0,
    gas=gas
)
```

```
# Symbolically execute program to find an exploit that obtains our funds back.
for state in m.running_states:
  world = state.platform

  # Check if funds can be retrieved
  if state.can_be_true(world.get_balance(user_account.address) == contract_balance):

    # If so, add constraint
    #   Then concretize symbolic buffer to provide one solution
    state.constraints.add(world.get_balance(user_account.address) == contract_balance)
    conc_args = state.solve_one(sym_args)

    # Print out our transaction to send to win
     print(f'''eth.sendTransaction({{data:"0x{conc_args.hex()}", from:"0x{from_address:04
0x}", to:"0x{si_level_address:040x}", gas:{gas}}})''')
    sys.exit(0)
```