CMPT 473
Software Testing, Reliability and Security

Symbolic Execution

Nick Sumner
wsumner@sfu.ca

Symbolic Execution

- As we have seen, building constraints that model code can be useful

CBMC was able to prove
certain errors couldn’t exist!

Symbolic Execution

- As we have seen, building constraints that model code can be useful
- With care, we can even use constraints to generate all inputs that are "interesting"

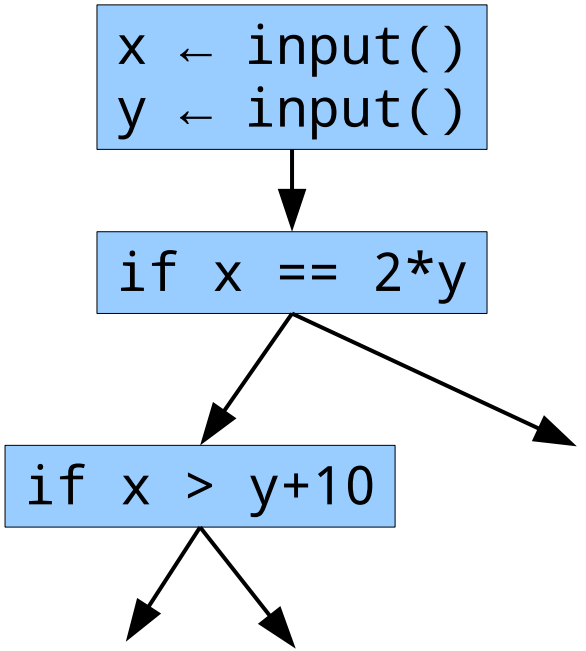
Symbolic Execution

- As we have seen, building constraints that model code can be useful
- With care, we can even use constraints to generate all inputs that are "interesting"
- Techniques for supporting this are known as ***symbolic execution***
  - (SymEx)

Symbolic Execution

- An approach for generating test inputs.

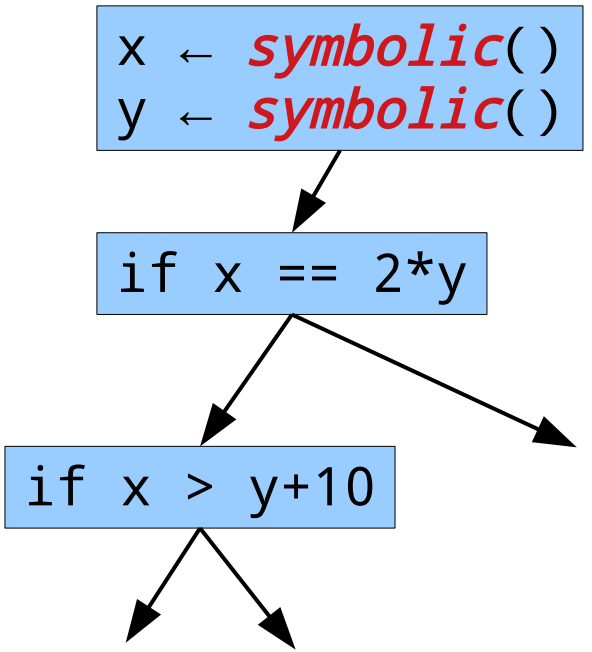
[Cadar & Sen, 2013]



Symbolic Execution

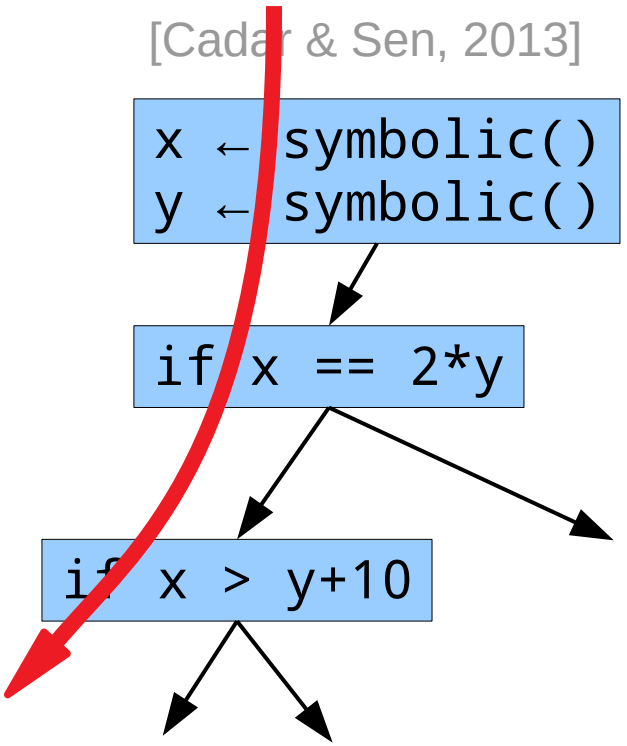
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values

[Cadar & Sen, 2013]



Symbolic Execution

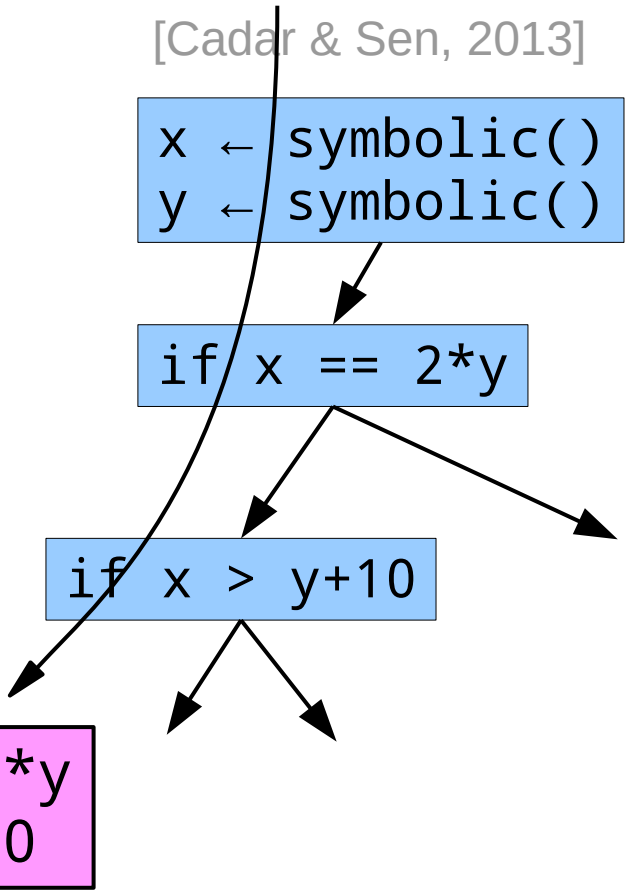
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.

[Cadar & Sen, 2013]



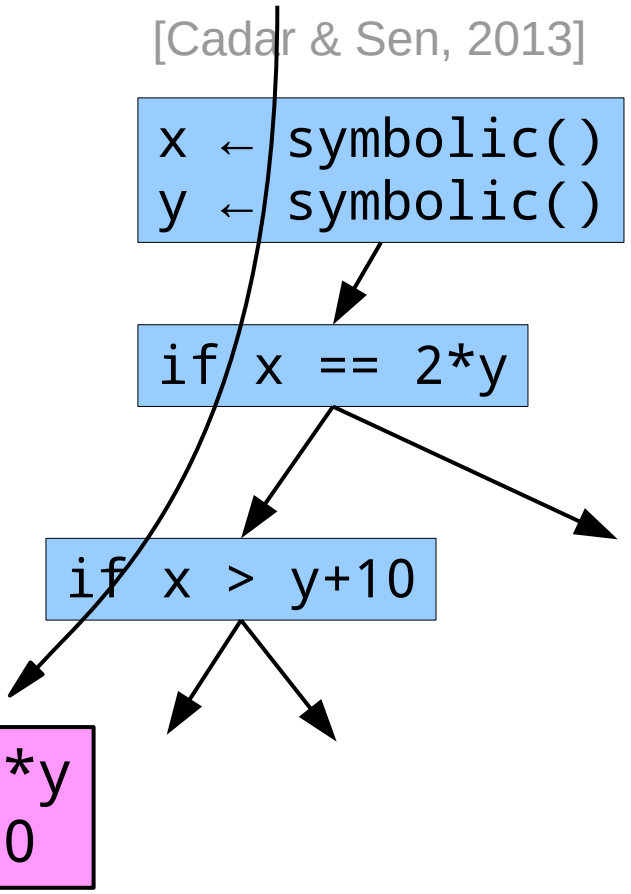
Path Constraint

Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.

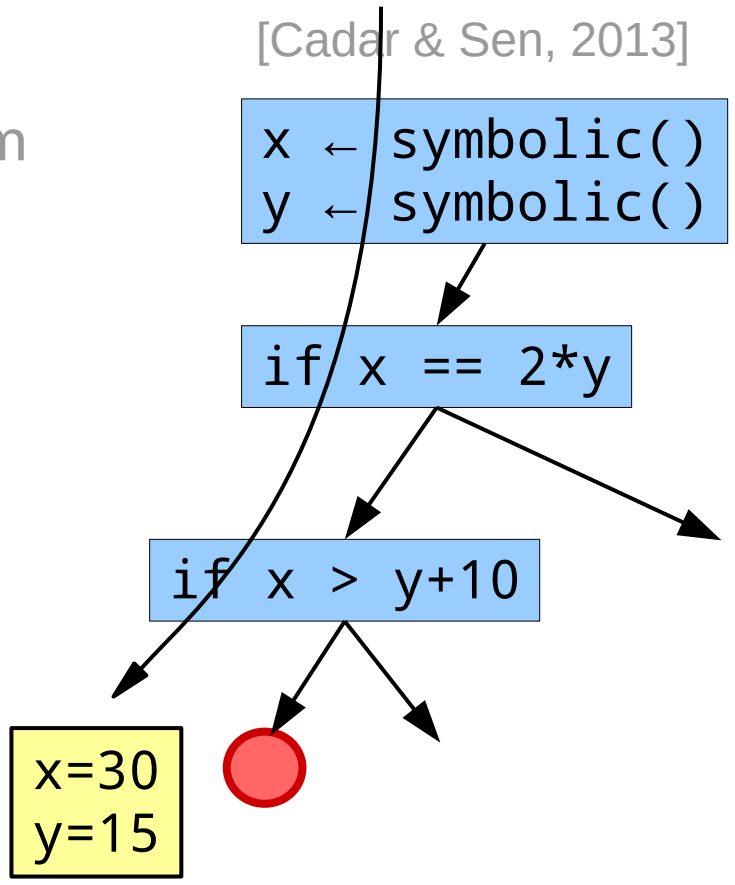
A path constraint represents all executions along that path

Path Constraint



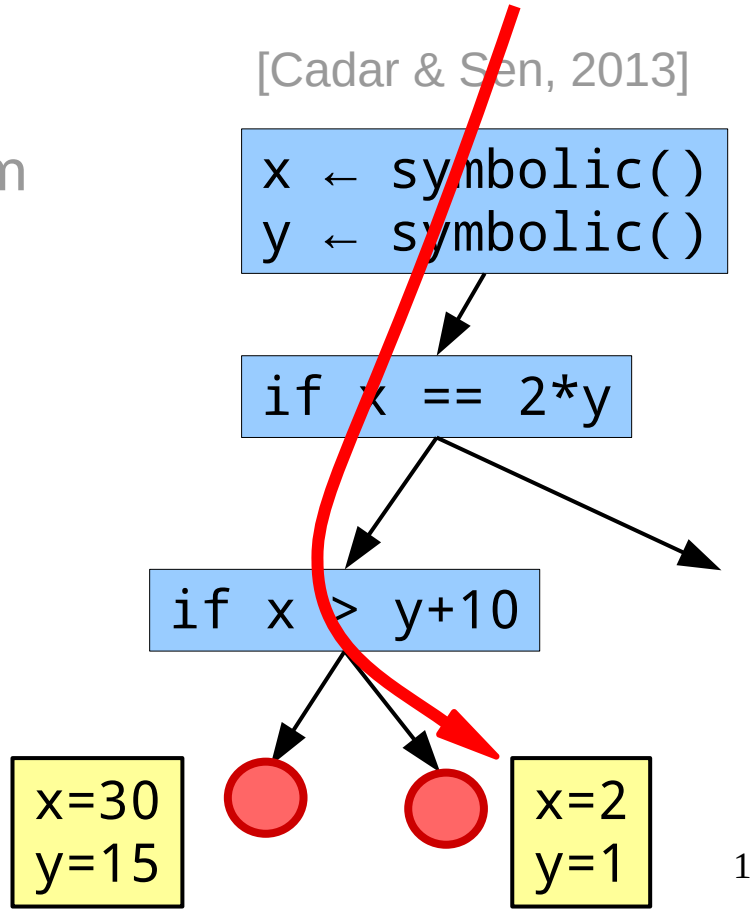
Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.



Symbolic Execution

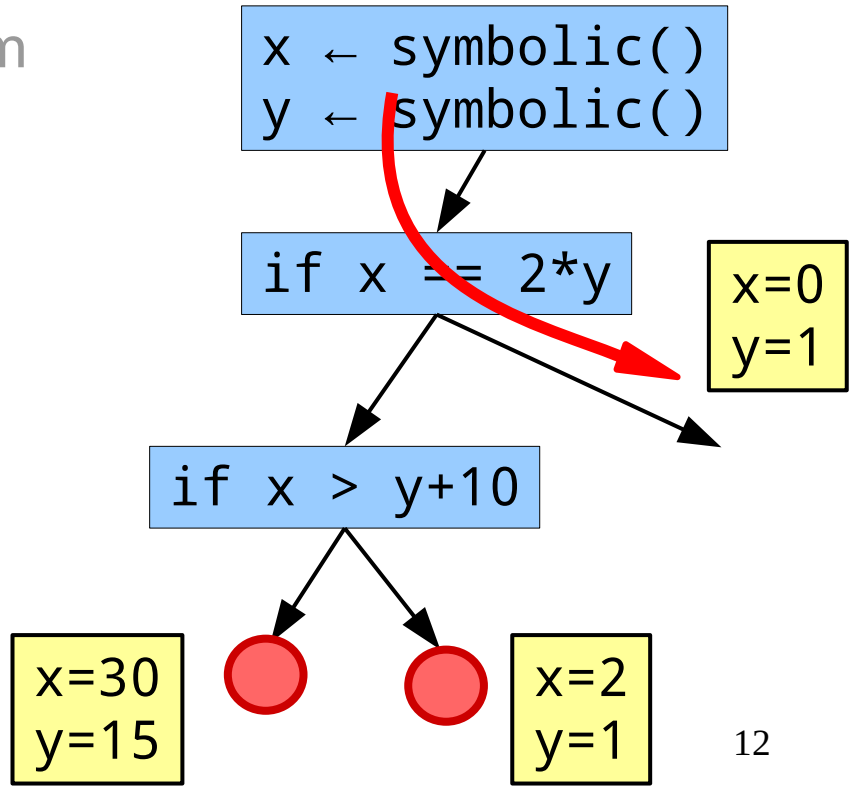
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.

[Cadar & Sen, 2013]



Using SymEx to solve problems

- Note that we described SymEx over *traces*.

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is *dynamic symbolic* execution.
  - There is also *static symbolic* execution (e.g. CBMC).

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is *dynamic symbolic* execution.
  - There is also *static symbolic* execution (e.g. CBMC).
- Applying constraint based reasoning on traces can also yield insights

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - There is also *static symbolic* execution (e.g. CBMC).
- Applying constraint based reasoning on traces can also yield insights
  - e.g. Suppose you are given two versions of a program v_1,v_2

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - There is also *static symbolic* execution (e.g. CBMC).
- Applying constraint based reasoning on traces can also yield insights
  - e.g. Suppose you are given two versions of a program v_1, v_2 and constraints on output φ_i in each from an input I

Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - There is also *static symbolic* execution (e.g. CBMC).
- Applying constraint based reasoning on traces can also yield insights
  - e.g. Suppose you are given two versions of a program v_1, v_2 and constraints on output φ_i in each from an input I

What is $wp(\varphi_1) \wedge \neg wp(\varphi_2)$?

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

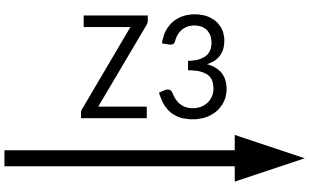
```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```



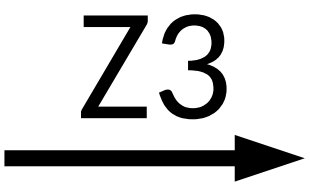

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```



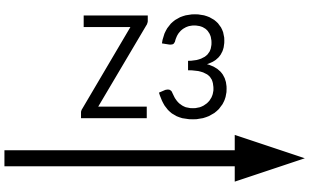

```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```




```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

```
x=22
y=11
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```


```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

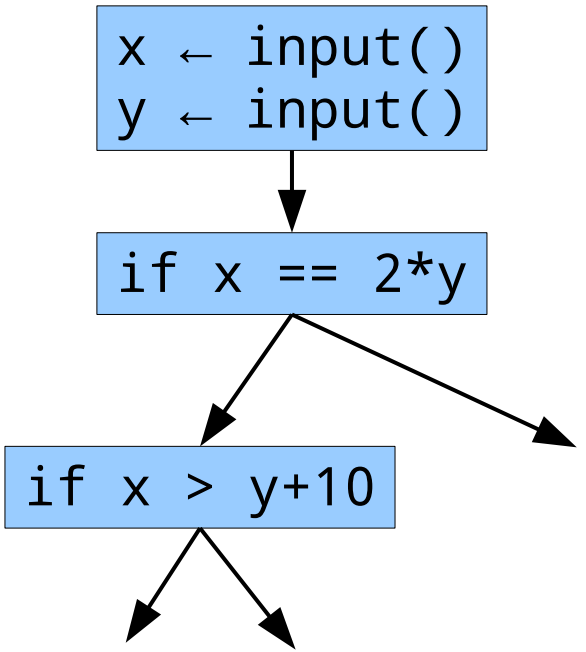
```
x=22
y=11
```

Try it online:
http://www.rise4fun.com/Z3/tutorial/

Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.

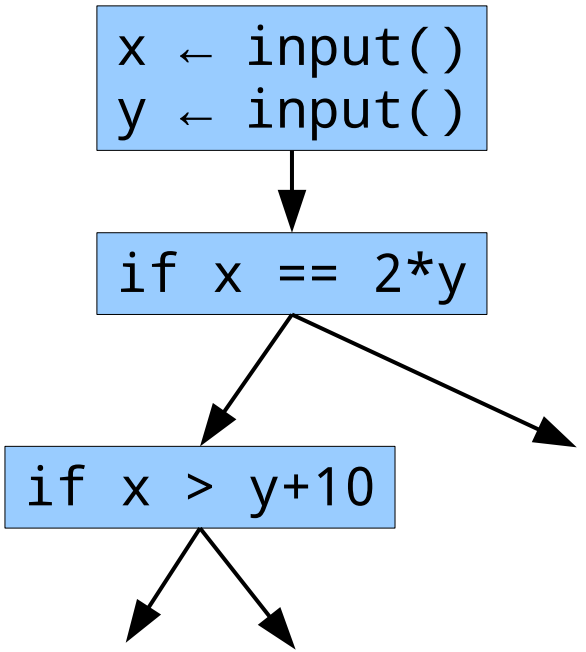
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.

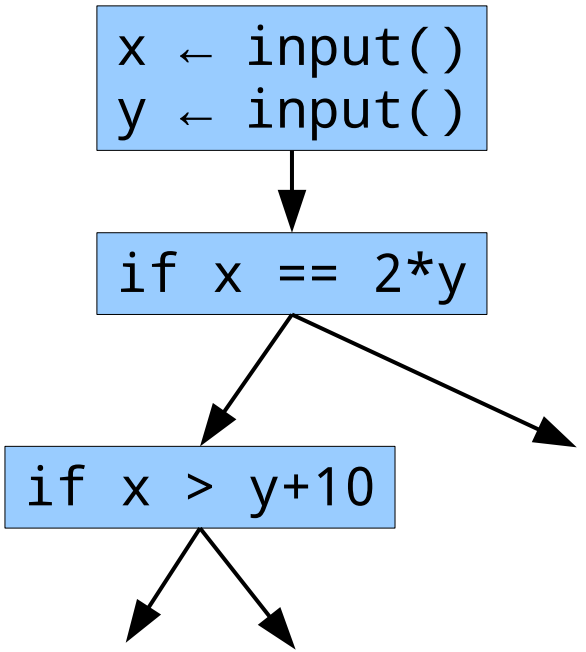
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches

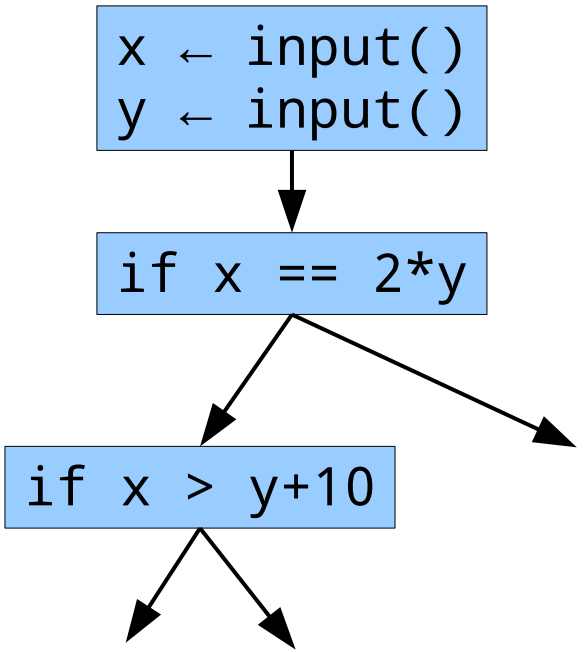
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

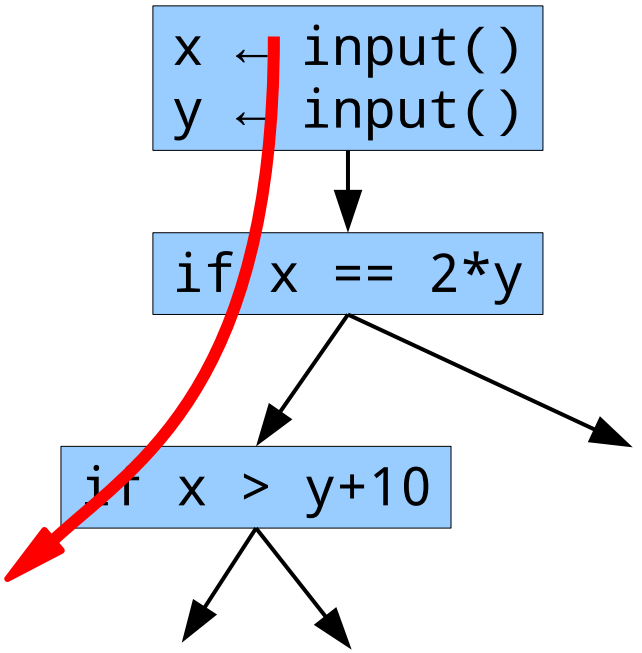
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

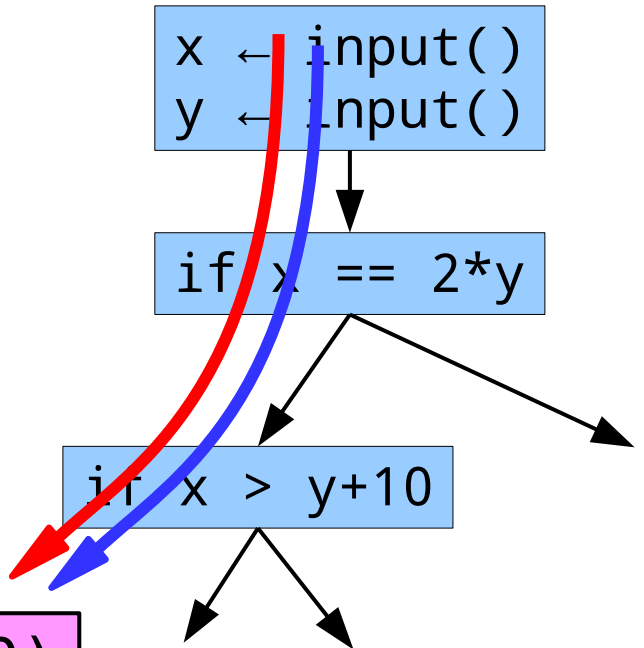
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

[Cadar & Sen, 2013]

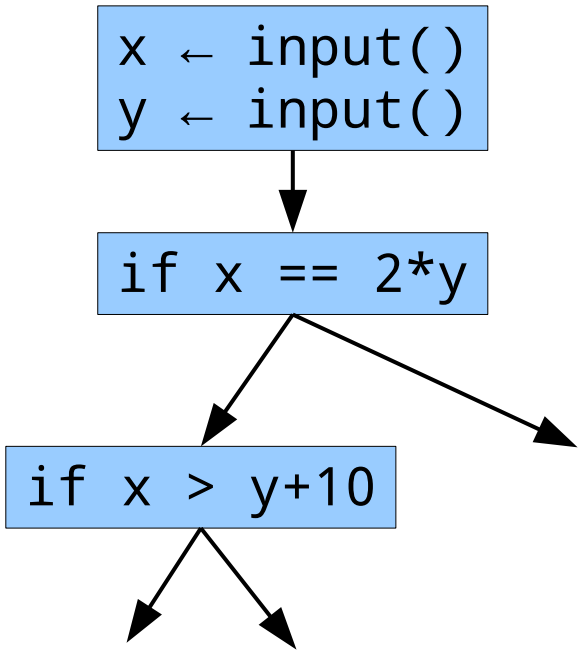


$(x=2*y) \wedge (x>y+10)$

Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

[Cadar & Sen, 2013]

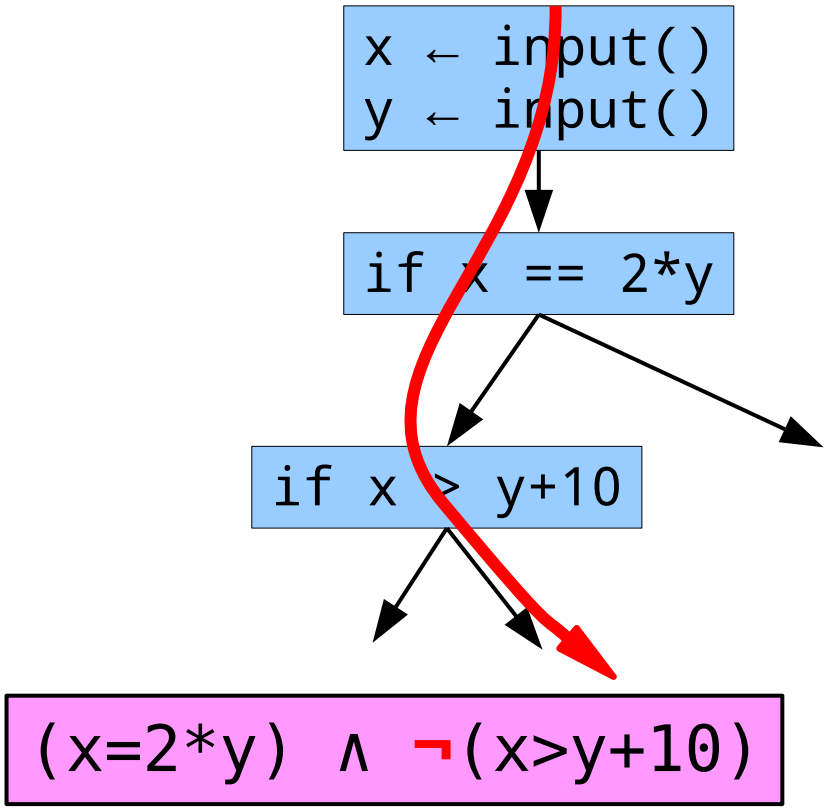


(x=2*y) ∧ ¬(x>y+10)

Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

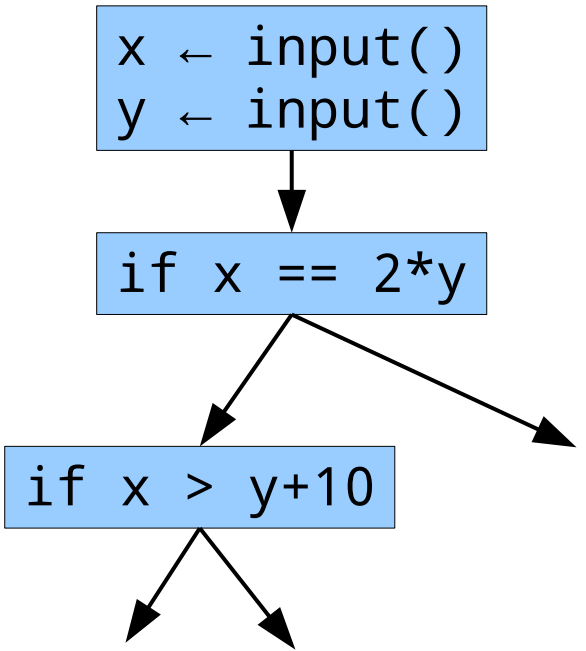
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

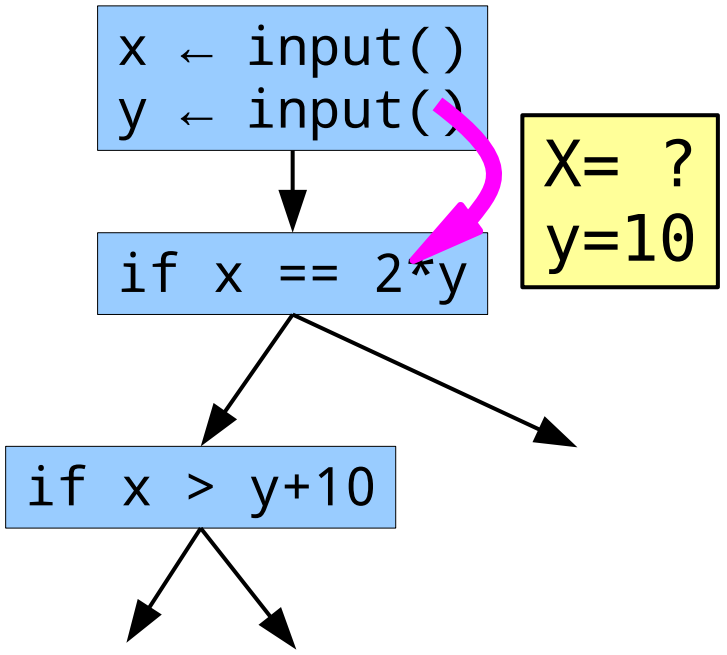
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

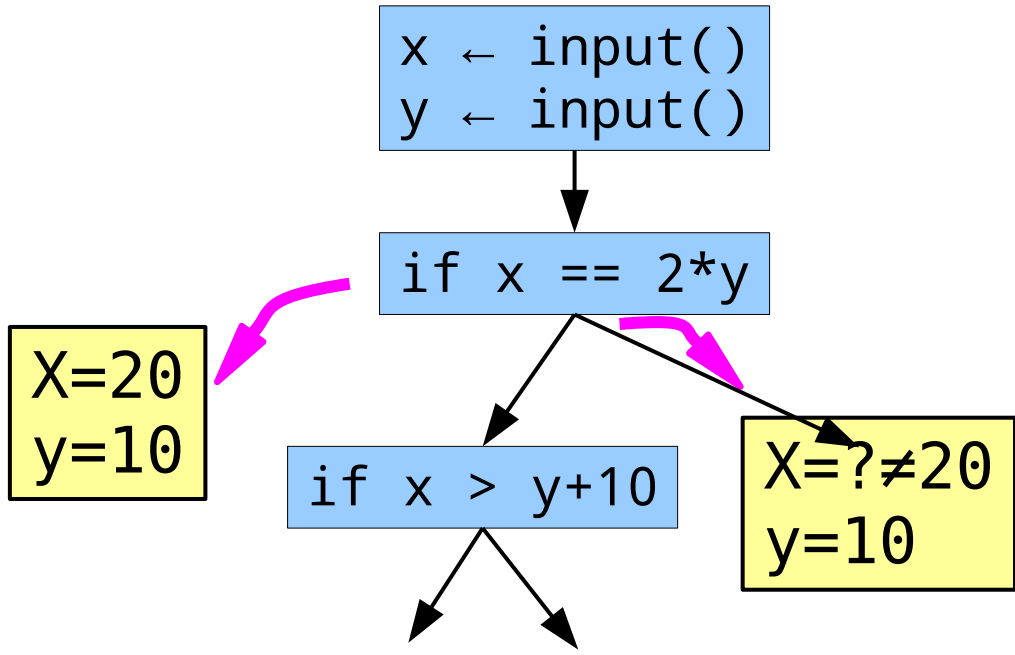
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

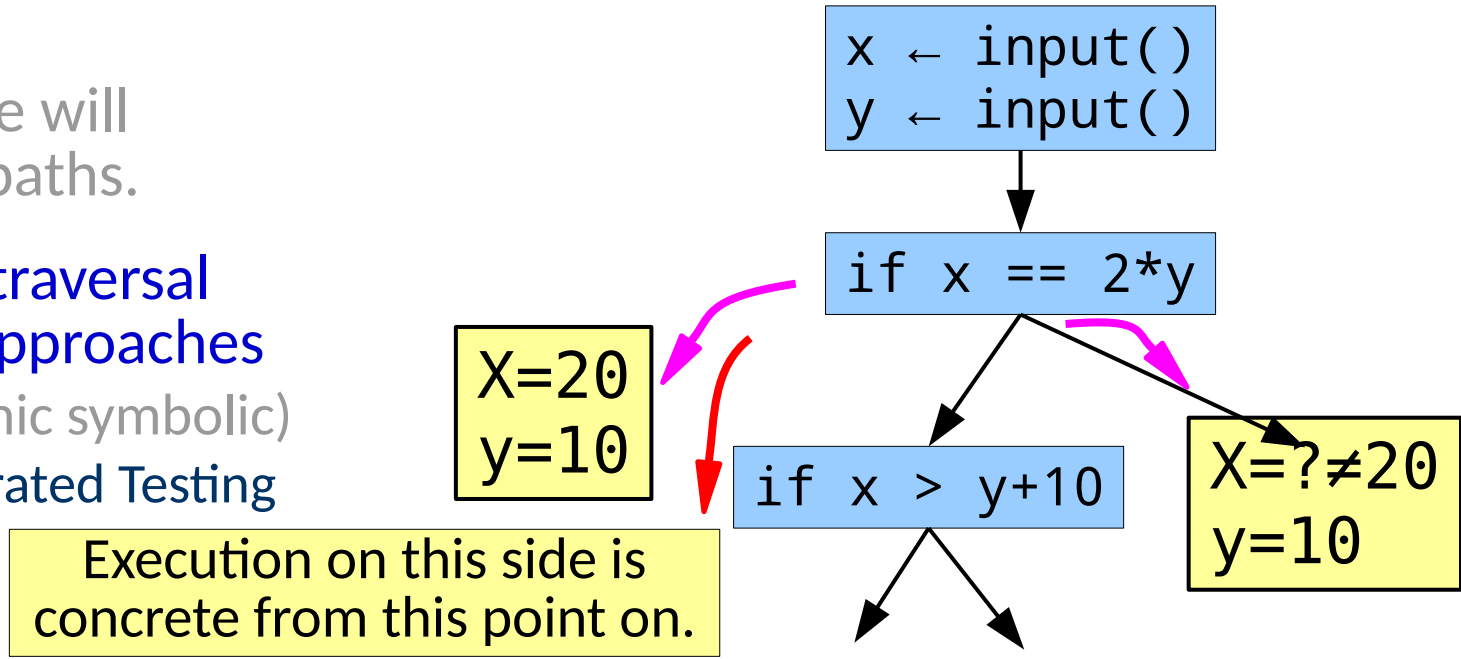
[Cadar & Sen, 2013]



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

[Cadar & Sen, 2013]



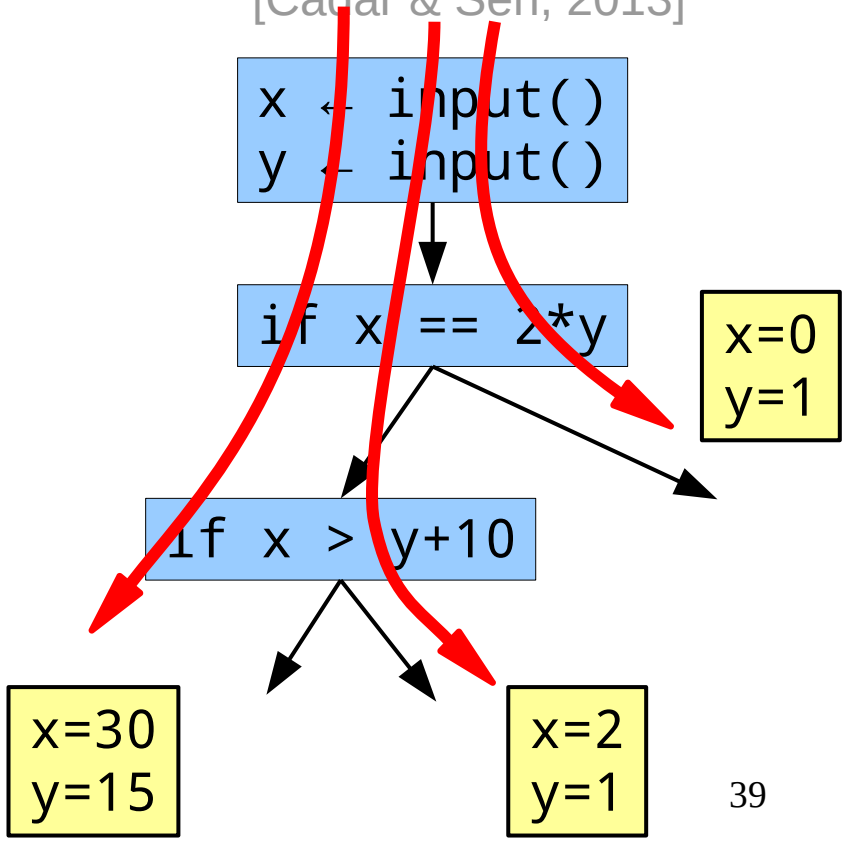
(Some) Applications

- Constructing test suites

(Some) Applications

- Constructing test suites

[Cadar & Sen, 2013]

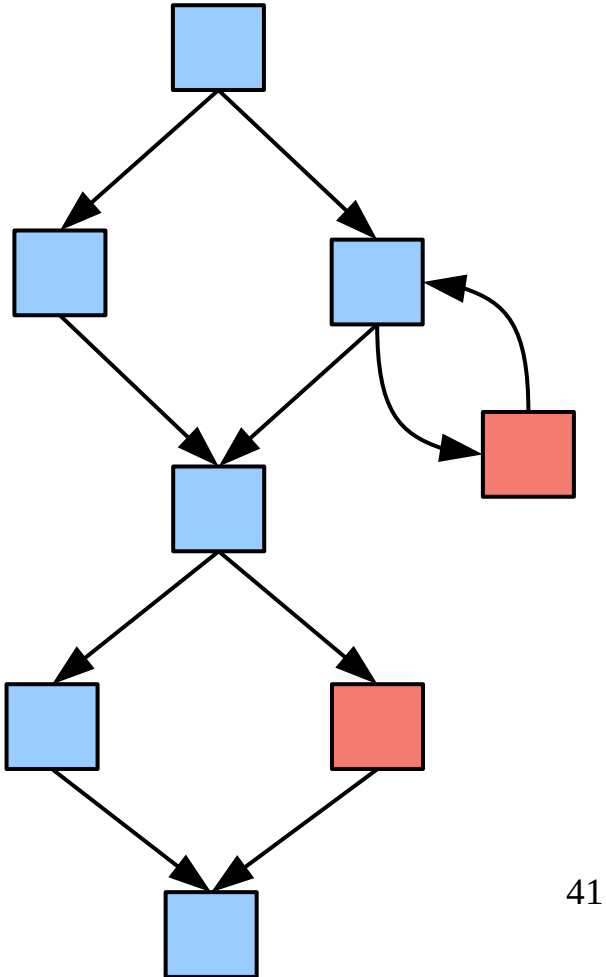


(Some) Applications

- Constructing test suites
- Targeted tests

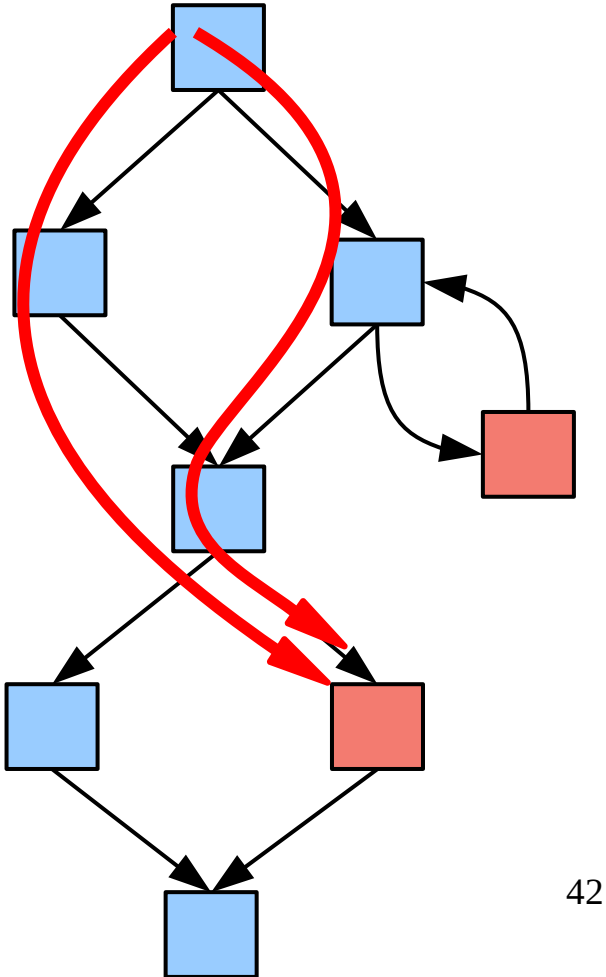
(Some) Applications

- Constructing test suites
- Targeted tests



(Some) Applications

- Constructing test suites
- Targeted tests

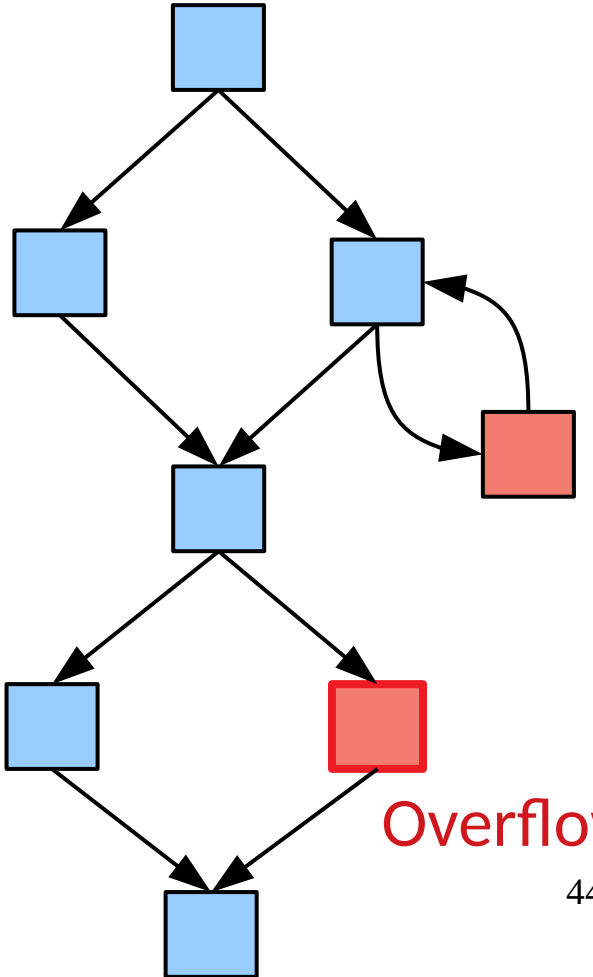


(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis

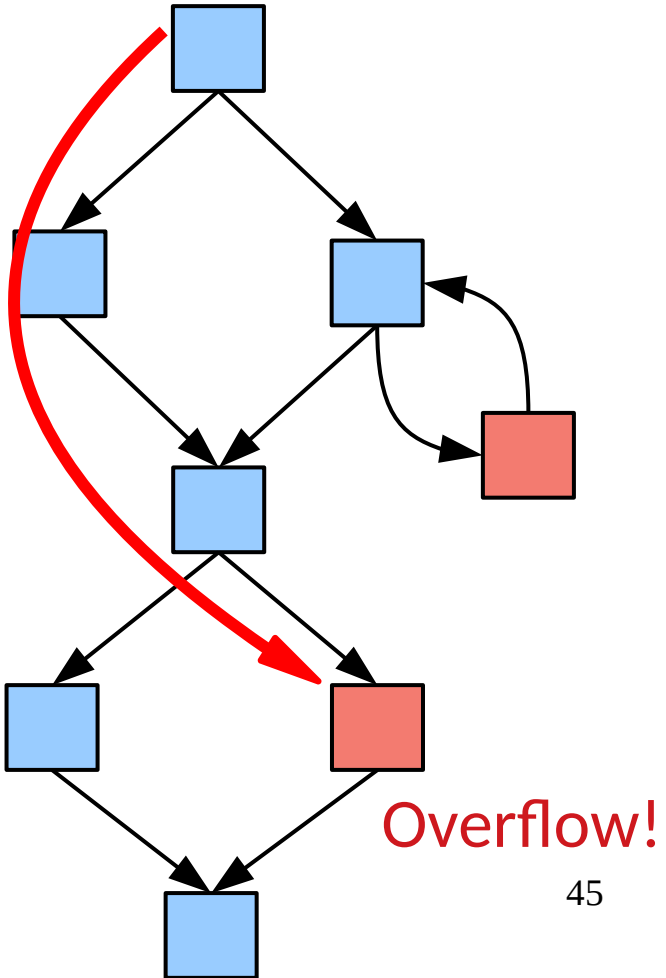
(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis



(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis

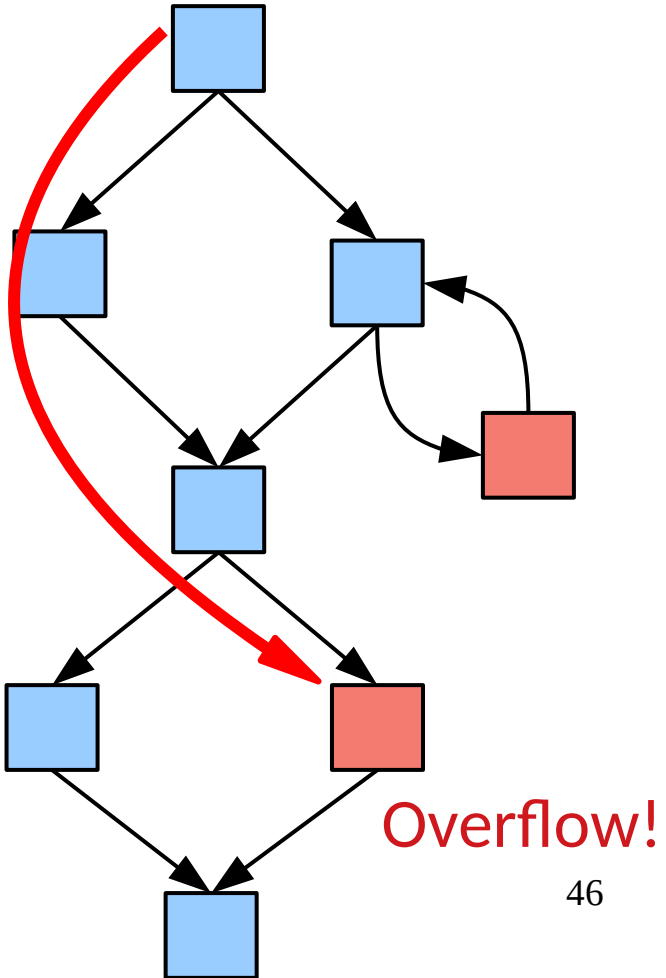


(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis

Input ⊢ Overflow ^ StartsShellcode

This is the core process for
Darpa Cybersecurity Grand Challenge entries!



(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis
- Test driven model checking (Yogi)
- …

The latest testing & verification services from MS are built around these techniques.

(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis
- Test driven model checking (Yogi)
- …

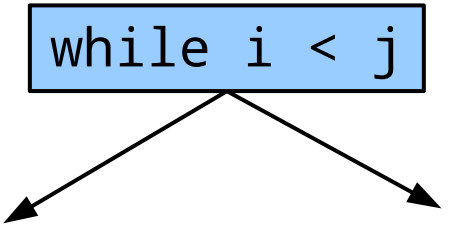
Let’s revisit a familiar example…

Challenges

- Path Explosion
- Challenging constraints
- Constraint representations & domain knowledge

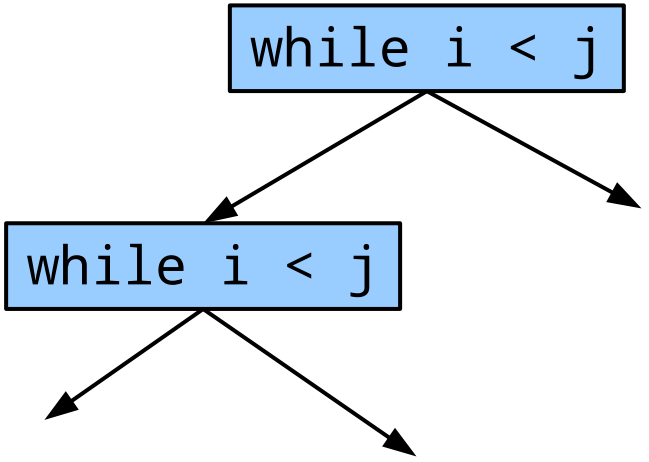
Path Explosion

- Loops



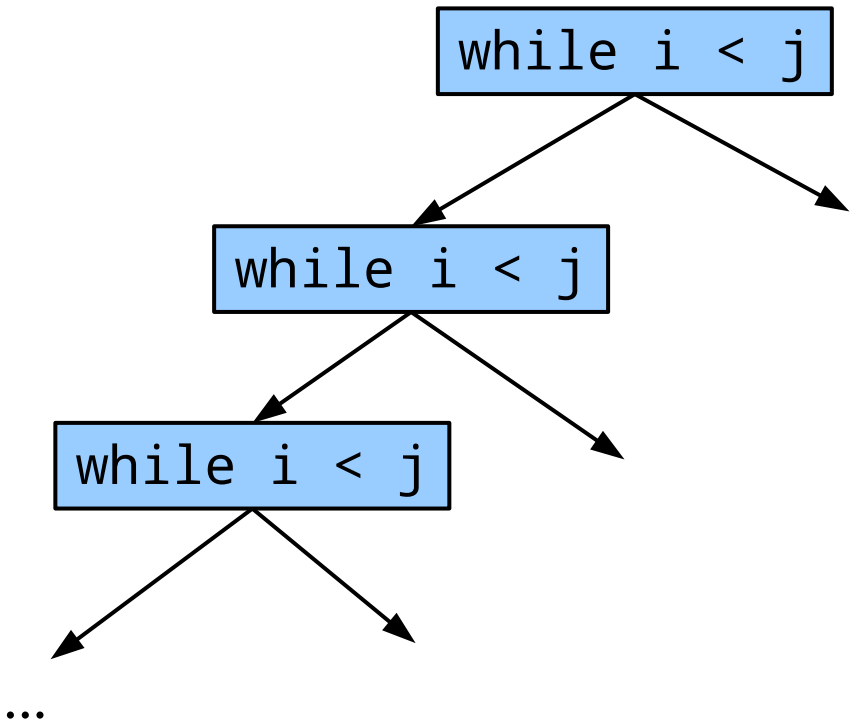
Path Explosion

- Loops



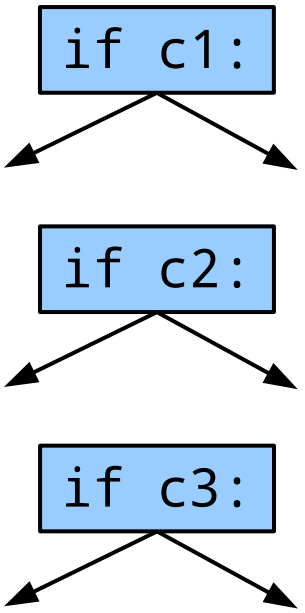
Path Explosion

- Loops



Path Explosion

- Loops
- Combinatorial Explosion



Path Explosion

- Loops
- Combinatorial Explosion

State of the art techniques carefully use summarization & representations to minimize these

Path Explosion

- Loops
- Combinatorial Explosion
- Strategies
  - Search heuristics
  - Memoization

Challenging Constraints

- Intuitively, we cannot solve all constraints

```
if hash(password) == y:
  print(“how odd”)
```

What would it imply if we could?

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?
  - IDEA: Observe the actual values of variables in runs we have

password = fritter
hash(password) = HJdjdskS&8sdh

```
if hash(password) == y:
  print(“how odd”)
```

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?
  - IDEA: Observe the actual values of variables in runs we have
    Substitute those observed values in challenging runs in the future

password = fritter
hash(password) = HjdjdskS&8sdh
y = HjdjdskS&8sdh

```
if hash(password) == y:
  print("how odd")
```

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?
  - IDEA: Observe the actual values of variables in runs we have
    Substitute those observed values in challenging runs in the future
  - Build a library of (input,output) pairs for challenging expressions
    (Use the theory of uninterpreted functions!)

Challenging Constraints

- Intuitively, we cannot solve all constraints
- How can we address this?
  - IDEA: Observe the actual values of variables in runs we have
    Substitute those observed values in challenging runs in the future
  - Build a library of (input,output) pairs for challenging expressions
    (Use the theory of uninterpreted functions!)

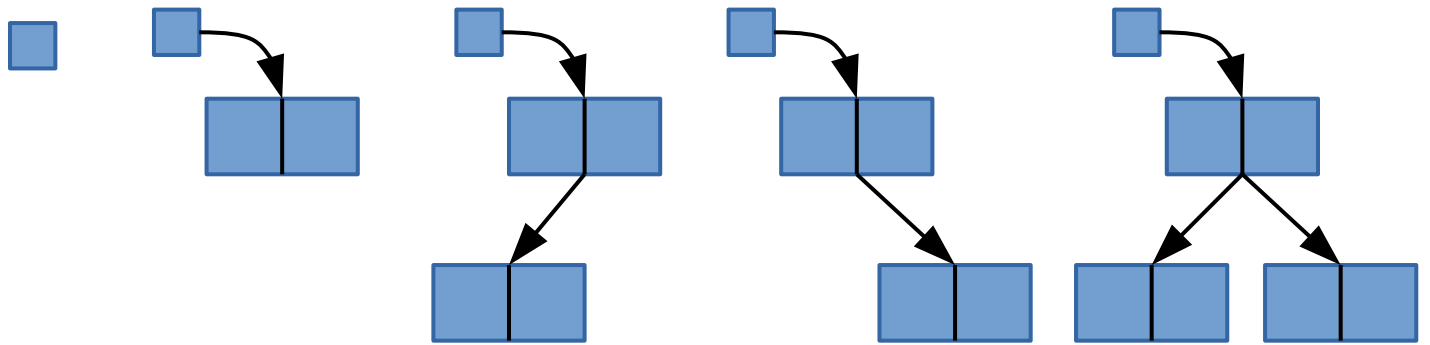
How do these affect our ability to explore the execution tree?

Domain Knowledge

- How should we represent memory?
  - A linear arrangement of memory?
  - Combinatorial aliasing relation pairs?

Domain Knowledge

- How should we represent memory?
  - A linear arrangement of memory?
  - Combinatorial aliasing relation pairs?
- Can we carefully explore interesting structures?
  - Korat style enumeration



Symbolic Execution

- Increasingly scalable every year

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints
- The resulting symbolic formulae have many uses beyond just testing.

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints
- The resulting symbolic formulae have many uses beyond just testing.

Try it out:
1) https://github.com/klee/klee
2) Symbolic PathFinder
3) http://research.microsoft.com/Pex/
4) http://angr.io/