CSE 403: Software Engineering, Winter 2016

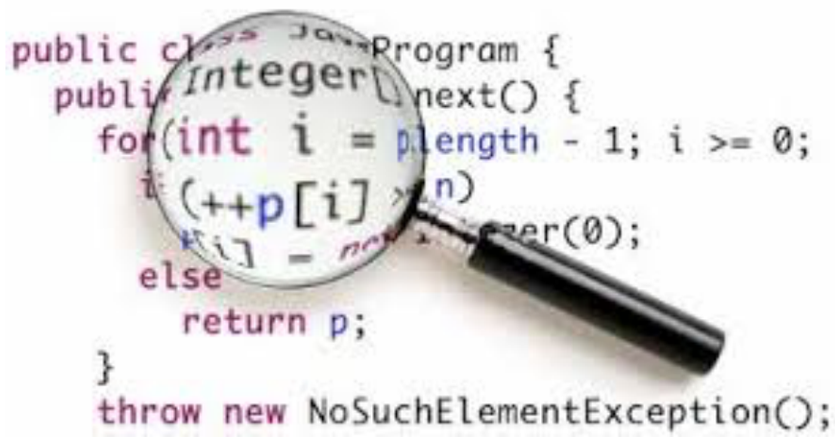
courses.cs.washington.edu/courses/cse403/16wi/

Symbolic Execution

Emina Torlak
emina@cs.washington.edu

Outline

- What is symbolic execution?
- How does it work?
- State-of-the-art tools



a brief introduction to symbolic execution

what

Recall from last time ...

Recall from last time ...

- Sound static analysis tools are great!
    - Can prove absence of many classes of important errors (such as runtime errors in safety critical systems)
    - High-quality commercial and open-source tools available

Recall from last time …

- Sound static analysis tools are great!
    - Can prove absence of many classes of important errors (such as runtime errors in safety critical systems)
    - High-quality commercial and open-source tools available
- But they are can be difficult to use unless you are an expert in static analysis …
    - They can produce many false positives on large and/or unusual code bases
    - For a sophisticated static analysis, telling a false positive from a real bug can be hard

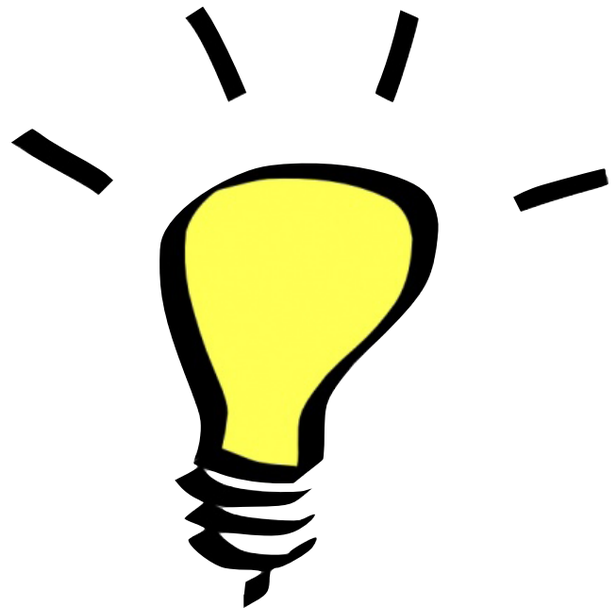
Symbolic execution

Symbolic execution

- A bug finding technique that is easy to use!
    - No false positives
    - Produces a *concrete* input (a test case) on which the program will fail to meet the specification
    - But it cannot, in general, prove the absence of errors

Symbolic execution

- A bug finding technique that is easy to use!
  - No false positives
  - Produces a *concrete* input (a test case) on which the program will fail to meet the specification
  - But it cannot, in general, prove the absence of errors
- Key idea
  - Evaluate the program on *symbolic* input values
  - Use an automated theorem prover to check whether there are corresponding *concrete* input values that make the program fail.



Symbolic execution

- A bug finding technique that is easy to use!
  - No false positives
  - Produces a *concrete* input (a test case) on which the program will fail to meet the specification
  - But it cannot, in general, prove the absence of errors
- Key idea
  - Evaluate the program on *symbolic* input values
  - Use an automated theorem prover to check whether there are corresponding *concrete* input values that make the program fail.

Demo!

Some history ...

1976: *A system to generate test data and symbolically execute programs* (Lori Clarke)

1976: *Symbolic execution and program testing* (James King)

2005-present: practical symbolic execution

Some history …

1976: *A system to generate test data and symbolically execute programs* (Lori Clarke)

1976: *Symbolic execution and program testing* (James King)

2005-present: practical symbolic execution

- Moore's Law
- Better theorem provers (SAT / SMT solvers)
- Heuristics to control exponential explosion
- Heap / environment modeling techniques, ….

symbolic execution by example

Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

$x \mapsto X$

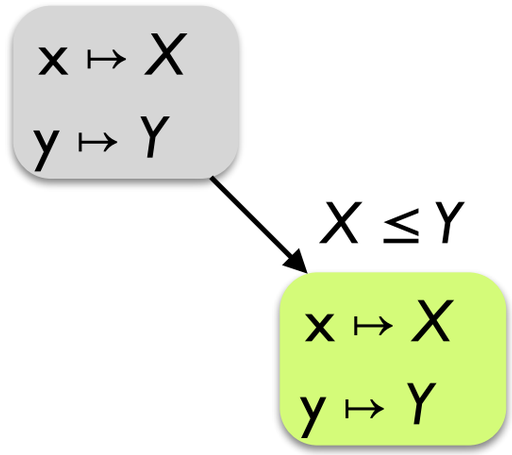
$y \mapsto Y$

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```



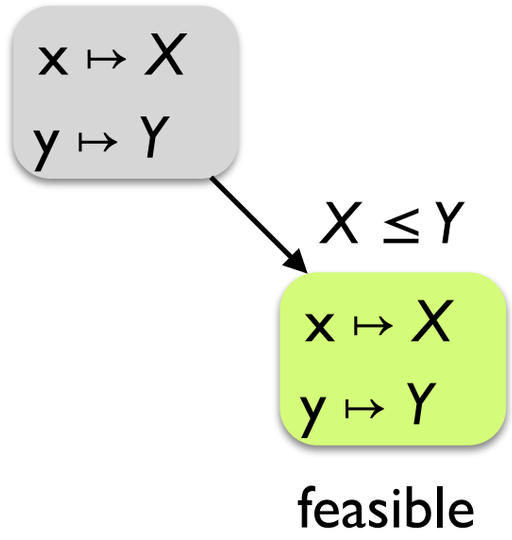
Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```



Execute the program on *symbolic values*.

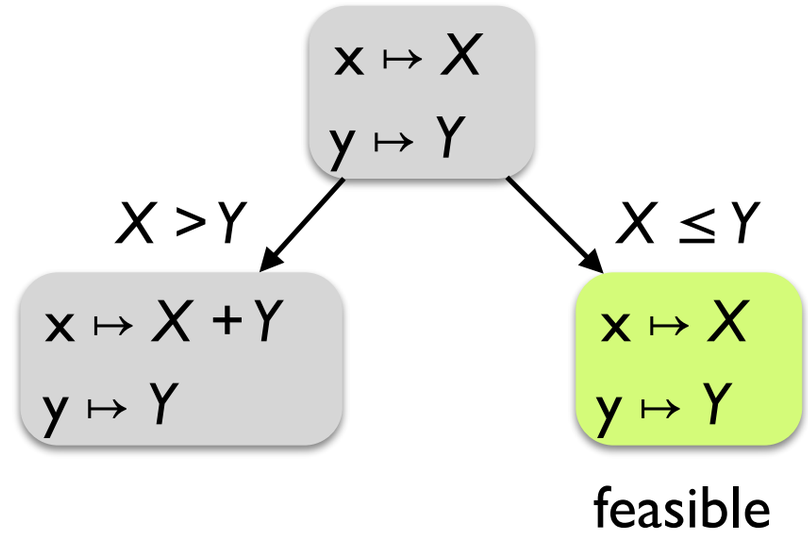
Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.

Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```



Execute the program on *symbolic values*.

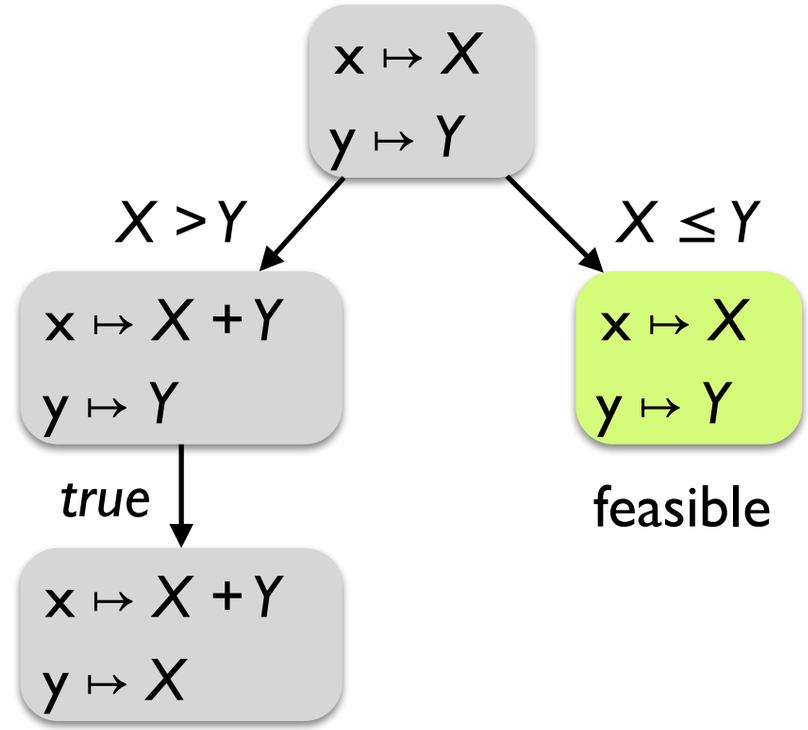
Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.

Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```



Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.

Classic symbolic execution

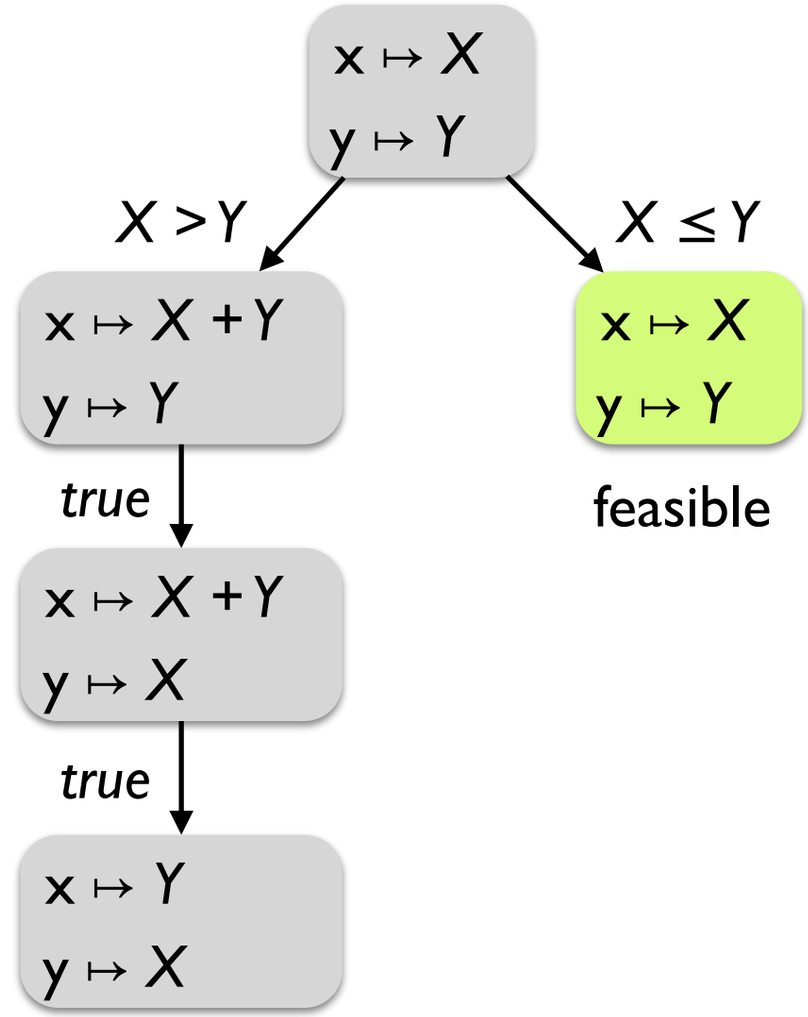
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Classic symbolic execution

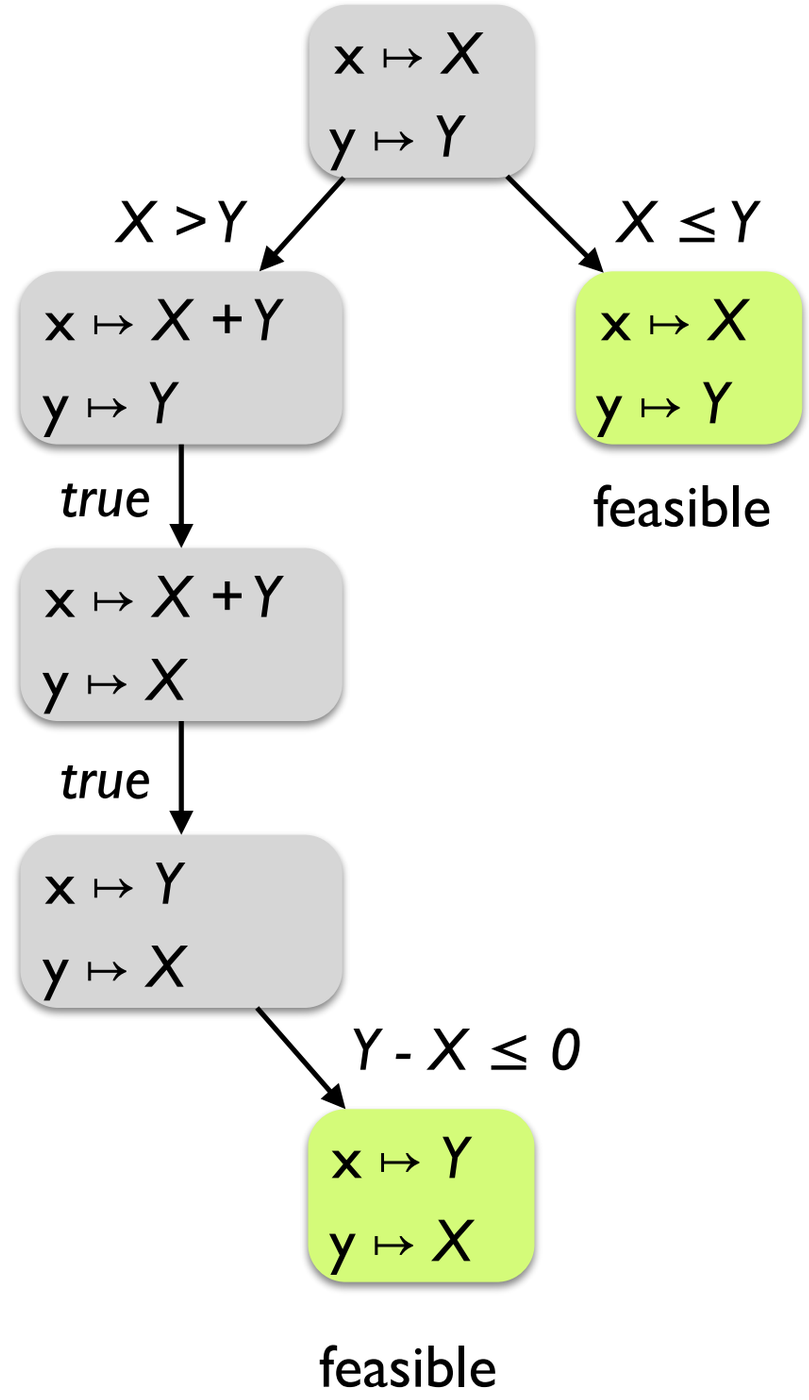
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Classic symbolic execution

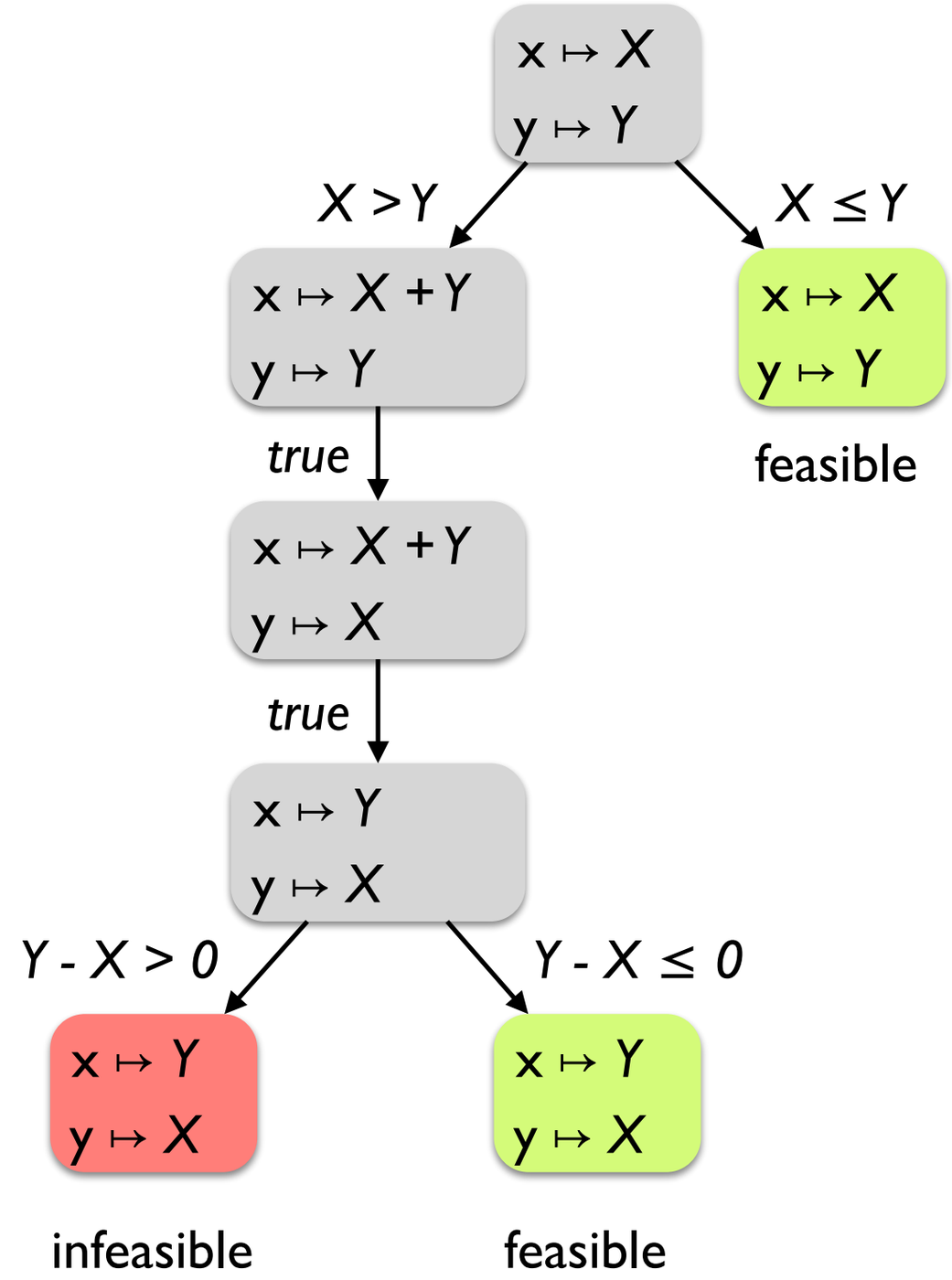
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.

All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible*.



Classic symbolic execution: practical issues

Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths

Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths

Heap modeling: symbolic data structures and pointers

Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths

Heap modeling: symbolic data structures and pointers

Solver limitations: dealing with complex PCs

Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths

Heap modeling: symbolic data structures and pointers

Solver limitations: dealing with complex PCs

Environment modeling: dealing with native / system / library calls

symbolic execution tools

Some state-of-the-art symbolic execution tools

- KLEE (symbolic execution for C, built on LLVM)
- SAGE (symbolic execution for x86)
- Jalangi (symbolic execution for JavaScript)
- Many, many others

Some state-of-the-art symbolic execution tools

- KLEE (symbolic execution for C, built on LLVM)
    - Found many bugs in open-source code, including the GNU Coreutils utility suite
    - Open-source: **https://klee.github.io/**
- SAGE (symbolic execution for x86)
- Jalangi (symbolic execution for JavaScript)
- Many, many others

Some state-of-the-art symbolic execution tools

- KLEE (symbolic execution for C, built on LLVM)
  - Found many bugs in open-source code, including the GNU Coreutils utility suite
  - Open-source: **https://klee.github.io/**
- SAGE (symbolic execution for x86)
  - Internal Microsoft tool
  - A huge cluster continuously running SAGE (**500+ machine years**)
    - 1/3 Windows 7 security bugs found by SAGE!
- Jalangi (symbolic execution for JavaScript)
- Many, many others

Summary

- Symbolic execution is a bug finding technique based on automated theorem proving:
  - Evaluates the program on symbolic inputs, and a solver finds concrete values for those inputs that lead to errors.
- Many success stories in the open-source community and industry.