Testing &
Symbolic Execution
Software Testing

• The most common way of measuring & ensuring correctness
Software Testing

• The most common way of measuring & ensuring correctness
Software Testing

- The most common way of measuring & ensuring correctness
Software Testing

- The most common way of measuring & ensuring correctness
Software Testing

- The most common way of measuring & ensuring correctness

Test Suite

- Test 1
  - Input
  - Oracle
- Test 2
  - Input
  - Oracle
- Test 3
  - Input
  - Oracle
- Test 4
  - Input
  - Oracle
- Test 5
  - Input
  - Oracle
- Test 6
  - Input
  - Oracle
- Test 7
  - Input
  - Oracle
Software Testing

• The most common way of measuring & ensuring correctness

• Key Issues:
  – Are the tests adequate?
Software Testing

- The most common way of measuring & ensuring correctness

- Key Issues:
  - Are the tests adequate?
  - Automated input generation
Software Testing

• The most common way of measuring & ensuring correctness

  Input → Program → Observed Behavior → Oracle → Outcome

• Key Issues:
  – Are the tests adequate?
  – Automated input generation
  – Automated oracles

Test Suite

| Test 1 | Input | Oracle |
| Test 2 | Input | Oracle |
| Test 3 | Input | Oracle |
| Test 4 | Input | Oracle |
| Test 5 | Input | Oracle |
| Test 6 | Input | Oracle |
| Test 7 | Input | Oracle |
Software Testing

- The most common way of measuring & ensuring correctness

- Key Issues:
  - Are the tests adequate?
  - Automated input generation
  - Automated oracles
  - Robustness / flakiness / maintainability
Software Testing

- The most common way of measuring & ensuring correctness

![Diagram showing input, program, observed behavior, and oracle with outcome]

- Key Issues:
  - Are the tests adequate?
  - Automated input generation
  - Automated oracles
  - Robustness / flakiness / maintainability
  - ...

Test Suite

<table>
<thead>
<tr>
<th>Test 1</th>
<th>Input</th>
<th>Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 2</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 3</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 4</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 5</td>
<td>Input</td>
<td>Oracle</td>
</tr>
<tr>
<td>Test 6</td>
<td>Input</td>
<td>Oracle</td>
</tr>
</tbody>
</table>
Test Suite Adequacy

• Questions
  – Is a test suite good enough?
  – What parts of software need to be tested better?
Test Suite Adequacy

• Questions
  – Is a test suite good enough?
  – What parts of software need to be tested better?

• Metrics
  – Statement Coverage

Is each statement executed by at least one test in the test suite?

```
def my_lovely_fun(a, b, c):
    if (a && b) || c:
        ...
    else:
        ...
    print('awesome')
```

\[
\text{score} = \frac{\# \text{ covered}}{\# \text{ statements}}
\]
Test Suite Adequacy

- Questions
  - Is a test suite good enough?
  - What parts of software need to be tested better?

- Metrics
  - Statement Coverage
  - Branch Coverage

\[
\text{score} = \frac{\text{# covered}}{\text{# branches}}
\]

```python
def my_lovely_fun(a, b, c):
    if (a && b) || c:
        ...
    else:
        ...
    print('awesome')
```
Test Suite Adequacy

• Questions
  – Is a test suite good enough?
  – What parts of software need to be tested better?

• Metrics
  – Statement Coverage
  – Branch Coverage
  – MC/DC Coverage

More common in safety critical systems where full coverage may be required.

```python
def my_lovely_fun(a, b, c):
    if (a & b) | c:
        ...
    else:
        ...
    print('awesome')
```
Test Suite Adequacy

- **Questions**
  - Is a test suite good enough?
  - What parts of software need to be tested better?

- **Metrics**
  - Statement Coverage
  - Branch Coverage
  - MC/DC Coverage
  - Mutation Coverage

```python
def my_lovely_func(a, b, c):
    if (a & b) | c:
        ...
    else:
        ...
    print('awesome')
```
Test Suite Adequacy

• Questions
  – Is a test suite good enough?
  – What parts of software need to be tested better?

• Metrics
  – Statement Coverage
  – Branch Coverage
  – MC/DC Coverage
  – Mutation Coverage

$$\text{score} = \frac{\# \text{covered/killed}}{\# \text{non-equivalent mutants}}$$
Test Suite Adequacy

• Questions
  – Is a test suite good enough?
  – What parts of software need to be tested better?

• Metrics
  – Statement Coverage
  – Branch Coverage
  – MC/DC Coverage
  – Mutation Coverage
  – Path Coverage
Test Suite Adequacy

• Questions
  – Is a test suite good enough?
  – What parts of software need to be tested better?

• Metrics
  – Statement Coverage
  – Branch Coverage
  – MC/DC Coverage
  – Mutation Coverage
  – Path Coverage

Can apply EPP!
Test Suite Adequacy

• Questions
  – Is a test suite good enough?
  – What parts of software need to be tested better?

• Metrics
  – Statement Coverage
  – Branch Coverage
  – MC/DC Coverage
  – Mutation Coverage
  – Path Coverage
  – ...

Test Suite Adequacy

• Questions
  – Is a test suite good enough?
  – What parts of software need to be tested better?

• Metrics
  – Statement Coverage
  – Branch Coverage
  – MC/DC Coverage
  – Mutation Coverage
  – Path Coverage

BUT reducing test suites for St, Br, MC/DC coverage decrease defect detection!
Generating Inputs

• Sample all possible inputs

```python
for test in allPossibleInputs:
    check_test(test)
```
Generating Inputs

• Sample all possible inputs

```python
for test in allPossibleInputs:
    check_test(test)
```

• Target specific goals

```python
for s in statements:
    test = findTestThrough(s)
    check_test(test)
```
or other coverage criteria
Generating Inputs

- Sample all possible inputs
  
  ```python
  for test in allPossibleInputs:
      check_test(test)
  ```

- Target specific goals
  
  ```python
  for s in statements:
      test = findTestThrough(s)
      check_test(test)
  ```

  ```python
  or other coverage criteria
  ```

  ```python
  for i in inputModel:
      test = findRepresentative(i)
      check_test(test)
  ```
Generating Inputs

- Usually broken into black box & white box approaches
Generating Inputs

• Usually broken into black box & white box approaches
  – **Black Box** – Treat the program as opaque / unknown
    
    e.g. specification based, naive fuzzing, boundary value analysis, ...
Generating Inputs

- Usually broken into black box & white box approaches
  - **Black Box** – Treat the program as opaque / unknown
  - **White Box** – Program structure & semantics can be used

  e.g. symbolic execution, call chain synthesis, white box fuzzing, boundary value analysis, ...
Generating Oracles
Generating Oracles

- Likely invariants?
- Careful variable selection & monitoring?
- Differential Testing
- Metamorphic Testing

A very open (hard) problem.
Interesting Problems

- Random Testing
- Test Suite Adequacy
- Test Suite Minimization
- Test Generation
- Oracle Generation
- Test Maintenance

- Performance Testing
- Field Testing
- Power Testing
- Test Prioritization
- ...
Interesting Problems

- Random Testing
- Test Suite Adequacy
- Test Suite Minimization
- Test Generation
- Oracle Generation
- Test Maintenance
- Performance Testing
- Field Testing
- Power Testing
- Test Prioritization
- ...

Test generation techniques have also proven to be critical in security research.
Symbolic Execution

- An approach for generating test inputs.

```plaintext
x ← input()
y ← input()

if x == 2*y

if x > y+10
```
Symbolic Execution

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

\[
\begin{align*}
x & \leftarrow \text{symbolic()} \\
y & \leftarrow \text{symbolic()} \\
\text{if } x &= 2*y \\
\text{if } x &> y + 10
\end{align*}
\]
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

\[
\begin{align*}
x & \leftarrow \text{symbolic()} \\
y & \leftarrow \text{symbolic()} \\
\text{if } x &= 2*y \\
\text{if } x > y+10
\end{align*}
\]
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

```
x ← symbolic()
y ← symbolic()
if x == 2*y
if x > y+10
x = 2*y
y > 10
```

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.

*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.
- Solve for the symbolic symbols to find inputs that yield the path.

Cadar & Sen, 2013

```
x ← symbolic()
y ← symbolic()

if x == 2*y

if x > y+10

x=30
y=15
```
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.

\[
x \leftarrow \text{symbolic()}
\]
\[
y \leftarrow \text{symbolic()}
\]
\[
\text{if } x == 2 \times y
\]
\[
\text{if } x > y + 10
\]
\[
x = 30
\]
\[
y = 15
\]
\[
x = 2
\]
\[
y = 1
\]

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.
- Solve for the symbolic symbols to find inputs that yield the path.
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

\[
\begin{align*}
\text{declare-const x Int} \\
\text{declare-const y Int} \\
\text{assert (= x (* 2 y))} \\
\text{assert (> y 10)} \\
\text{(check-sat)} \\
\text{(get-model)}
\end{align*}
\]
How Can We Solve Constraints?

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

\[
\begin{align*}
  x &= 2 \times y \\
  y &> 10
\end{align*}
\]

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

Z3
How Can We Solve Constraints?

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

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

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

**Z3**

```
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

\[
\begin{align*}
\text{x} & = 2\times y \\
y & > 10
\end{align*}
\]

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

Z3  
sat  
(model  
(define-fun y () Int 11)  
(define-fun x () Int 22)  
)

Try it online:  
Useful Questions

• If \( \varphi \) holds after a statement, what must have been true at the point before?
  – \textit{weakest precondition}
Useful Questions

- If $\varphi$ holds after a statement, what must have been true at the point before?  
  - weakest precondition

- If $\varphi$ holds before a statement, what can we guarantee to be true after?  
  - strongest poscondition
Useful Questions

• If φ holds after a statement, what must have been true at the point before?
  – *weakest precondition*

• If φ holds before a statement, what can we guarantee to be true after?
  – *strongest postcondition*

e.g. Given two versions of a program v1, v2 and assertions on output φi in each from an input I

What is wp(φ1) ∧ ¬wp(φ2)?
Useful Questions

- If $\phi$ holds after a statement, what must have been true at the before?
  - weakest precondition

- If $\phi$ holds before a statement, what can we guarantee to be true after?
  - strongest poscondition

e.g. Given two versions of a program $v_1, v_2$ and assertions on output $\phi_i$ in each from an input $I$

What is $wp(\phi_1) \wedge \neg wp(\phi_2)$?   $wp(\phi_1) \Rightarrow wp(\phi_2)$
Useful Questions

• If $\varphi$ holds after a statement, what must have been true at the before?
  – weakest precondition

• If $\varphi$ holds before a statement, what can we guarantee to be true after?
  – strongest poscondition

e.g. Given two versions of a program $v_1, v_2$ and assertions on output $\varphi_i$ in each from an input $I$

What is $wp(\varphi_1) \land \neg wp(\varphi_2)$? $wp(\varphi_1) \Rightarrow wp(\varphi_2)$

What is the intuitive meaning?
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.

```plaintext
x ← input()
y ← input()

if x == 2*y
    if x > y+10

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.

```plaintext
x ← input()
y ← input()
if x == 2*y
    if 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)

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)

\[
(x=2*y) \land (x>y+10)
\]

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)

\[
\begin{align*}
x & \leftarrow \text{input()} \\
y & \leftarrow \text{input()} \\
\text{if } x & = 2 \times y \\
\text{if } x & > y+10 \\
\text{(x=2*y) } & \land \neg(x>y+10)
\end{align*}
\]
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

Execution on this side is concrete from this point on.

Cadar & Sen, 2013
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/