A Brief Intro to Automated Test Generation

Nick Sumner
Our Test Suites Are Still Limited

- There is only so much we can include
- Even covering interesting interactions is a challenge
Our Test Suites Are Still Limited

- There is only so much we can include
- Even covering interesting interactions is a challenge

Our first naive solution may not have been naive!

```
for test in allPossibleInputs:
    run_program(test)
```
Our Test Suites Are Still Limited

- There is only so much we can include
- Even covering interesting interactions is a challenge

Our first naive solution may not have been naive!

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

How might this be pragmatically useful?
Automated Test Generation

- We can *continuously* run new tests
Automated Test Generation

- We can *continuously* run new tests
  - But manual testing this way won't work!
Automated Test Generation

- We can continuously run new tests
  - But manual testing this way won't work!

- **Automated Test Generation**
  - Use program analysis to derive new tests without the user
Automated Test Generation

- We can continuously run new tests
  - But manual testing this way won't work!
- Automated Test Generation
  - Use program analysis to derive new tests without the user
- 2 approaches are increasingly common
Automated Test Generation

- We can continuously run new tests
  - But manual testing this way won't work!
- Automated Test Generation
  - Use program analysis to derive new tests without the user
- 2 approaches are increasingly common
  - Fuzz Testing
Automated Test Generation

- We can continuously run new tests
  - But manual testing this way won't work!

- Automated Test Generation
  - Use program analysis to derive new tests without the user

- 2 approaches are increasingly common
  - Fuzz Testing
  - Symbolic Execution
Fuzz Testing

- An approach for generating test inputs
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

```
./grep "02d6..." RandomFile
```
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

```
./grep "02d6..." RandomFile
```

It was distressingly effective at finding buffer overflows (25%-33% of programs).
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

- Now 2 main types
  1. *Generational* (model based)
     - Creates entirely new inputs
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
     - Creates entirely new inputs
     - Needs a *model* for the input
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
     - Creates entirely new inputs
     - Needs a *model* for the input
     $a^*bc(d|e)c^*$
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

- Now 2 main types
  1) *Generational* (model based)
     - Creates entirely new inputs
     - Needs a *model* for the input

\[
a^*bc(d|e)c^* \\
A \rightarrow aAb \\
A \rightarrow cA \\
A \rightarrow \varepsilon
\]
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
     - Creates entirely new inputs
     - Needs a *model* for the input

\[
\begin{align*}
a^*bc(d|e)c^* \\
A &\rightarrow aAb \\
A &\rightarrow cA \\
A &\rightarrow \epsilon \\
\end{align*}
\]
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

- Now 2 main types
  1) Generational (model based)
  2) Mutational (heuristic change based)
     - Modify an existing test suite
Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

- Now 2 main types
  1) *Generational* (model based)
  2) *Mutational* (heuristic change based)
    - Modify an existing test suite
    - Seeing a resurgence via *AFL* & *libFuzzer*
American Fuzzy Lop

- Increasingly used mutational fuzzer
  - Effective at finding buffer overflows
**American Fuzzy Lop**

- Increasingly used mutational fuzzer

```
<table>
<thead>
<tr>
<th>process timing</th>
<th>overall results</th>
</tr>
</thead>
<tbody>
<tr>
<td>run time</td>
<td>cycles done : 0</td>
</tr>
<tr>
<td>last new path</td>
<td>total paths : 2448</td>
</tr>
<tr>
<td>last uniq crash</td>
<td>uniq crashes : 111</td>
</tr>
<tr>
<td>last uniq hang</td>
<td>uniq hangs : 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>map coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>now processing</td>
<td>map density : 3702 (5.65%)</td>
</tr>
<tr>
<td>paths timed out</td>
<td>count coverage : 5.83 bits/tuple</td>
</tr>
<tr>
<td></td>
<td>findings in depth</td>
</tr>
<tr>
<td></td>
<td>favored paths : 221 (9.03%)</td>
</tr>
<tr>
<td></td>
<td>new edges on : 401 (16.38%)</td>
</tr>
<tr>
<td></td>
<td>total crashes : 427 (111 unique)</td>
</tr>
<tr>
<td></td>
<td>total hangs : 0 (0 unique)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>path geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>now trying</td>
<td>levels : 3</td>
</tr>
<tr>
<td>stage execs</td>
<td>pending : 2420</td>
</tr>
<tr>
<td>total execs</td>
<td>pend fav : 213</td>
</tr>
<tr>
<td>exec speed</td>
<td>own finds : 2350</td>
</tr>
<tr>
<td></td>
<td>imported : n/a</td>
</tr>
<tr>
<td>fuzzing strategy yields</td>
<td>variable : 0</td>
</tr>
<tr>
<td>bit flips</td>
<td></td>
</tr>
<tr>
<td>byte flips</td>
<td></td>
</tr>
<tr>
<td>arithmetics</td>
<td></td>
</tr>
<tr>
<td>known ints</td>
<td></td>
</tr>
<tr>
<td>dictionary</td>
<td></td>
</tr>
<tr>
<td>havoc</td>
<td></td>
</tr>
<tr>
<td>trim</td>
<td></td>
</tr>
</tbody>
</table>
```

CPU: 40%
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

\[ x \leftarrow \text{symbolic}() \]
\[ y \leftarrow \text{symbolic}() \]

if \( x = 2y \)

if \( x > y+10 \)

Cadar & Sen, 2013
Symbolic Execution

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

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

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

Path Constraint

Cadar & Sen, 2013
Symbolic Execution

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

\[
x \leftarrow \text{symbolic()}
\]
\[
y \leftarrow \text{symbolic()}
\]
\[
\text{if } x \geq 2 \cdot y
\]
\[
\text{if } x > y + 10
\]
\[
x = 30
\]
\[
y = 15
\]
Symbolic Execution

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

\[
x \leftarrow \text{symbolic()}
\]
\[
y \leftarrow \text{symbolic()}
\]
\[
\text{if } x \geq 2 \times y
\]
\[
\text{if } x > y + 10
\]
\[
\begin{align*}
x &= 30 \\
y &= 15
\end{align*}
\]
\[
\begin{align*}
x &= 2 \\
y &= 1
\end{align*}
\]
Symbolic Execution

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

```python
x ← symbolic()
y ← symbolic()
if x == 2*y
if x > y+10
  x=30
  y=15
  x=2
  y=1
  x=0
  y=1
```
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
```

(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

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

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

```
(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
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

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

Z3

x = 2*y
y > 10

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**.

```
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.

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)*

```python
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)

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

```python
x ← input()
y ← input()
if x == 2*y
   if x > y+10
      \( (x=2*y) \land \neg(x>y+10) \)
```

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)

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 used beyond just testing.

Try it out:
https://github.com/klee/klee
Where They Fit in the Process

• Automated test generation is a continual process.
Where They Fit in the Process

- Automated test generation is a continual process.
- Just as much a part of modern QA as continuous integration
Where They Fit in the Process

- Automated test generation is a continual process.
- Just as much a part of modern QA as continuous integration
- Especially crucial as part of maintaining security (more on this later!)