CMPT 745
Software Engineering

Symbolic Execution

Nick Sumner
wsumner@sfu.ca
Symbolic Execution

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

- As we have seen, building constraints that model code can be useful.
- With care, we can even try 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 try 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]

```
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

[Cadar & Sen, 2013]

```plaintext
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 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 along a path using the symbolic values to build a formula over the input symbols.

[Cadar & Sen, 2013]

$x \leftarrow \text{symbolic}()$
$y \leftarrow \text{symbolic}()$

If $x = 2y$

If $x > y + 10$

Path Constraint

$x = 2y$
$y > 10$
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

[Cadar & Sen, 2013]

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

[Cadar & Sen, 2013]

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

```
x=30
y=15
```

```
x=2
y=1
```
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.
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.
  - What we saw before was essentially static symbolic execution.
Using SymEx to solve problems

- Note that we described SymEx over *traces*.
  - This is dynamic symbolic execution.
  - What we saw before was essentially static symbolic execution.

- 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.
  - What we saw before was essentially static symbolic execution.

- 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.
  - What we saw before was essentially static symbolic execution.

- 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 $\varphi_i$ in each from an input $I$
Using SymEx to solve problems

- Note that we described SymEx over traces.
  - This is dynamic symbolic execution.
  - What we saw before was essentially static symbolic execution.

- 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 $\varphi_i$ in each from an input $I$

$$\text{What is } wp(\varphi_1) \land \neg wp(\varphi_2)?$$
How Can We Solve Constraints?

- **SMT Solvers**
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMPLIB2
How Can We Solve Constraints?

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

\[
\begin{align*}
x &= 2y \\
y &> 10
\end{align*}
\]
How Can We Solve Constraints?

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

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

\[ x = 2 \times y \]
\[ y > 10 \]
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)
```

$x = 2y$
$y > 10$

Z3
How Can We Solve Constraints?

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

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

\[ x = 2y \]
\[ y > 10 \]

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

**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
```

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

```
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

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

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

\[
\begin{align*}
\text{sat} \\
\text{(model)} \\
\text{(define-fun y () Int 11)} \\
\text{(define-fun x () Int 22)}
\end{align*}
\]

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.
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)

```
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
  - Concolic (dynamic symbolic)

[Cadar & Sen, 2013]

```
x ← input()
y ← input()

if x == 2*y
  ...
else:
  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)

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

[Concolic example]

```
x ← input()
y ← input()
if x == 2*y
  if x > y+10
  (x=2*y) ∧ ¬(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]

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

[Çadar & Sen, 2013]

```
x ← input()
y ← input()
if x == 2*y
X= ?
y=10
```

```
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)
  - Execution Generated Testing

[Cadar & Sen, 2013]

```
x ← input()
y ← input()
if x == 2*y
  X=20
  y=10
if x > y+10
  X=?≠20
  y=10
```
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.
(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

Overflow!
(Some) Applications

- Constructing test suites
- Targeted tests
- Automated exploit discovery & synthesis
(Some) Applications

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

Input $\models$ Overflow $\land$ 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)
- ...

Application: Test Driven Model Checking

- While traditional testing is *sampling*,

Input Domain
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program.
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program.
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program.
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program.
- Carefully choosing which questions to ask can allow us to prove properties of programs!
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program.

- Carefully choosing which questions to ask can allow us to prove properties of programs!

```c
if (p == p1) return
if (p == p2) return
*p1 = 0; ...;
*p = 1
assert(*p1 != 1 && ...)
```

Can the assertion here fail?
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program.

- Carefully choosing which questions to ask can allow us to prove properties of programs!

```
Execution Tree

```

```c
if (p == p1) return
if (p == p2) return
*p1 = 0; ...;
*p = 1
assert(*p1 !=1 && ...)
```

Can the assertion here fail?
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program.

- Carefully choosing which questions to ask can allow us to prove properties of programs!

Execution Tree

```c
if (p == p1) return
if (p == p2) return
*p1 = 0; ...;
*p = 1
assert(*p1 !=1 && ...)
```

Can the assertion here fail?
Application: Test Driven Model Checking

- While traditional testing is sampling, SymEx enables targeted tests that answer questions about a program.
- Carefully choosing which questions to ask can allow us to prove properties of programs!

Do you see any potential problems with this approach as given?

```
if (p == p1) return
if (p == p2) return
*p1 = 0; ...;
*p = 1
assert(*p1 != 1 && ...)
```

Can the assertion here fail?
Challenges

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

- Loops

```
while i < j

while i < j

while i < j

...
```
Path Explosion

- Loops
- Combinatorial Explosion
Path Explosion

- Loops
- Combinatorial Explosion
- Strategies
Path Explosion

- Loops
- Combinatorial Explosion
- Strategies
  - Search heuristics (DFS, BFS, Targeted, Merged, ...)

62
Path Explosion

- Loops
- Combinatorial Explosion

**Strategies**
- Search heuristics
- Memoization (Have we already analyzed this?)
Challenging Constraints

- Intuitively, we cannot solve all constraints

```python
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

```python
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

```python
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?
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
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

- Can we use more constrained problems than SAT/SMT?
  - Many problems can use simpler constrained Horn clauses
Interesting Directions

- How can we speed up or remove symbolic operations?
Interesting Directions

- How can we speed up or remove symbolic operations?
  - Neural strategies
Interesting Directions

- How can we speed up or remove symbolic operations?
  - Neural strategies
  - Indexing & memoization
Interesting Directions

• How can we speed up or remove symbolic operations?
  – Neural strategies
  – Indexing & memoization
  – Aggressively using theory of uninterpreted functions
Interesting Directions

- How can we speed up or remove symbolic operations?
  - Neural strategies
  - Indexing & memoization
  - Aggressively using theory of uninterpreted functions

- Probabilistic Symbolic Execution
  - How *likely* is one path vs another?
  - Can that direct our search toward more interesting areas?
Interesting Directions

- How can we speed up or remove symbolic operations?
  - Neural strategies
  - Indexing & memoization
  - Aggressively using theory of uninterpreted functions

- Probabilistic Symbolic Execution
  - How *likely* is one path vs another?
  - Can that direct our search toward more interesting areas?

- Decomposing goals into smaller problems
  - How can we analyze systems like Linux, Chrome, & Firefox well? [Brown 2020]
Carefully choosing which questions to ask can allow us to prove properties of programs!

```c
if (p == p1) return
if (p == p2) return
*p1 = 0; ...;
*p = 1
assert(*p1 !=1 && ...)
Can the assertion here fail?
```
Carefully choosing which questions to ask can allow us to prove properties of programs!

- Some relationships may be hard or missing

```
if (p == p1) return

if (p == p2) return

*p1 = 0; ...;
*p = 1

assert(*p1 != 1 && ...)
```

Can the assertion here fail?
Revisit: Test Driven Model Checking

- Carefully choosing which questions to ask can allow us to prove properties of programs!
  - Some relationships may be hard or missing
  - Full combinatorial search will not scale

```c
if (p == p1) return
if (p == p2) return
*p1 = 0; ...;
*p = 1
assert(*p1 != 1 && ...)
```

Can the assertion here fail?
Carefully choosing which questions to ask can allow us to prove properties of programs!
- Some relationships may be hard or missing
- Full combinatorial search will not scale
- We still want a proof of correctness

```
if (p == p1) return

if (p == p2) return

*p1 = 0; ...;
*p = 1

assert(*p1 != 1 && ...)
```

Can the assertion here fail?
Carefully choosing which questions to ask can allow us to prove properties of programs!

[Beckman, TSE 2016]
Carefully choosing which questions to ask can allow us to prove properties of programs!

[Beckman, TSE 2016]
Carefully choosing which questions to ask can allow us to prove properties of programs!

```c
if (p == p1) return
if (p == p2) return
*p1 = 0; ...;
*p = 1
```

Can the assertion here fail?

```
if (p == p1) return
if (p == p2) return
*p1 = 0; ...;
*p = 1
assert(*p1 !=1 && ...)
```

[Beckman, TSE 2016]
Carefully choosing which questions to ask can allow us to prove properties of programs!

![Diagram](image.png)
Carefully choosing which questions to ask can allow us to prove properties of programs!

Revisit: Test Driven Model Checking

The error state is now unreachable!

Can the assertion here fail?
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://an gr.io/