Random Testing

Nick Sumner
wsumner@sfu.ca
Our test suites are intrinsically limited

- Test suites are limited
Our test suites are intrinsically limited

- Test suites are limited
  - A test suite typically contains enough tests to instill confidence but no more
Our test suites are intrinsically limited

- Test suites are limited
  - A test suite typically contains enough tests to instill confidence but no more
  - Test suite adequacy measures help us quantify that confidence
Our test suites are intrinsically limited

- Test suites are limited
  - A test suite typically contains enough tests to instill confidence but no more
  - Test suite adequacy measures help us quantify that confidence
  - Program analysis can help to uncover interesting bugs in a given test suite
Our test suites are intrinsically limited

- Test suites are limited
  - A test suite typically contains enough tests to instill confidence but no more
  - Test suite adequacy measures help us quantify that confidence
  - Program analysis can help to uncover interesting bugs in a given test suite

- How can we hope to uncover new & unexpected bugs?
Our test suites are intrinsically limited

- Test suites are limited
  - A test suite typically contains enough tests to instill confidence but no more
  - Test suite adequacy measures help us quantify that confidence
  - Program analysis can help to uncover interesting bugs in a given test suite

- How can we hope to uncover new & unexpected bugs?
  - Static analysis provides one direction but is still challenging
Our test suites are intrinsically limited

- Test suites are limited
  - A test suite typically contains enough tests to instill confidence but no more
  - Test suite adequacy measures help us quantify that confidence
  - Program analysis can help to uncover interesting bugs in a given test suite

- How can we hope to uncover new & unexpected bugs?
  - Static analysis provides one direction but is still challenging
  - Maybe our first naive solution was not naive...
Our test suites are intrinsically limited

- Test suites are limited
  - A test suite typically contains enough tests to instill confidence but no more
  - Test suite adequacy measures help us quantify that confidence
  - Program analysis can help to uncover interesting bugs in a given test suite

- How can we hope to uncover new & unexpected bugs?
  - Static analysis provides one direction but is still challenging
  - Maybe our first naive solution was not naive...

```python
for test in allPossibleInputs:
    run_program(test)
```
Our test suites are intrinsically limited

- Test suites are limited
  - A test suite typically contains enough tests to instill confidence but no more
  - Test suite adequacy measures help us quantify that confidence
  - Program analysis can help to uncover interesting bugs in a given test suite

- How can we hope to uncover new & unexpected bugs?
  - Static analysis provides one direction but is still challenging
  - Maybe our first naive solution was not naive...

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

How might this be pragmatically useful?
Random Testing

- We can continuously run new tests
  - Doing this manually / with manually constructed tests is clearly wrong
Random Testing

- We can continuously run new tests
  - Doing this manually / with manually constructed tests is clearly wrong

- Random Testing
  - Use program analysis to randomly sample new tests without user interaction
Random Testing

- We can continuously run new tests
  - Doing this manually / with manually constructed tests is clearly wrong

- Random Testing
  - Use program analysis to randomly sample new tests without user interaction

- Several directions have arisen
Random Testing

- We can continuously run new tests
  - Doing this manually / with manually constructed tests is clearly wrong

- Random Testing
  - Use program analysis to randomly sample new tests without user interaction

- Several directions have arisen
  - **Fuzz Testing**
    Generating new inputs from a model or existing suite
Random Testing

- We can continuously run new tests
  - Doing this manually / with manually constructed tests is clearly wrong

- Random Testing
  - Use program analysis to randomly sample new tests without user interaction

- Several directions have arisen
  - Fuzz Testing
    Generating new inputs from a model or existing suite
  - Feedback Directed Random Testing
    Generating OOP unit tests as a sequence of method calls
Random Testing

- We can continuously run new tests
  - Doing this manually / with manually constructed tests is clearly wrong

- Random Testing
  - Use program analysis to randomly sample new tests without user interaction

- Several directions have arisen
  - Fuzz Testing
    Generating new inputs from a model or existing suite
  - Feedback Directed Random Testing
    Generating OOP unit tests as a sequence of method calls
  - Property based testing
  - Chaos Engineering
Random Testing

- We can continuously run new tests
  - Doing this manually / with manually constructed tests is clearly wrong

- Random Testing
  - Use program analysis to randomly sample new tests without user interaction

- Several directions have arisen
  - Fuzz Testing
    Generating new inputs from a model or existing suite
  - Feedback Directed Random Testing
    Generating OOP unit tests as a sequence of method calls
  - Property based testing
  - Chaos Engineering

We’ll discuss these more later.
The need not be random.
Fuzz Testing

- Historically, fuzz testing was naive:
Fuzz Testing

- Historically, fuzz testing was naive:
  1) Generate random file/string
  2) Pass random string/file to program
  3) Look for crash
Fuzz Testing

- Historically, fuzz testing was naive:
  1) Generate random file/string
  2) Pass random string/file to program
  3) Look for crash

- But it was alarmingly effective even then
  ```
  ./grep "02d6..." RandomFile
  Found buffer overflows (25%-33% of programs).
  ```
Fuzz Testing

• Historically, fuzz testing was naive:
  1) Generate random file/string
  2) Pass random string/file to program
  3) Look for crash

• But it was alarmingly effective even then

  ./grep “02d6…” RandomFile

  Found buffer overflows (25%-33% of programs).

• Techniques have evolved along several dimensions
  – Is an initial test suite required?
  – How are new tests generated?
  – How does the success / failure of previous tests affect test generation?
  – What kinds of bugs can be found?
Fuzz Testing

- Can be classified along many dimensions
  - Each of those previous points and more that we will consider
Fuzz Testing

- Can be classified along many dimensions
  - Each of those previous points and more that we will consider

- 2 major ways to generate inputs:
Fuzz Testing

- Can be classified along many dimensions
  - Each of those previous points and more that we will consider

- 2 major ways to generate inputs:
  - *Generational*
    - Creates entirely new inputs
    - Needs a model of the possible input space
Fuzz Testing

- Can be classified along many dimensions
  - Each of those previous points and more that we will consider

- 2 major ways to generate inputs:
  - *Generational*
    - Creates entirely new inputs
    - Needs a model of the possible input space
  - *Mutational*
    - Modifies an existing suite of inputs
    - Seeing a resurgence in tools like
Fuzz Testing

• Can be classified along many dimensions
  – Each of those previous points and more that we will consider

• 2 major ways to generate inputs:
  – *Generational*
    • Creates entirely new inputs
    • Needs a model of the possible input space
  – *Mutational*
    • Modifies an existing suite of inputs
    • Seeing a resurgence in tools like
  – Even more state of the art approaches blend generation & mutation further
Generational Fuzz Testing

- Sample inputs from a model of the input space
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?

```
a*bc(d|e)c*
```
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?

\[ a^*bc(d|e)c^* \]

\[
\begin{align*}
A & \rightarrow aAb \\
A & \rightarrow cA \\
A & \rightarrow \varepsilon
\end{align*}
\]
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?

\[ a^*bc(d|e)c^* \]

\[ A \rightarrow aAb \]
\[ A \rightarrow cA \]
\[ A \rightarrow \varepsilon \]
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?

We can randomly rewrite nonterminals to sample:

A

\[ a^*bc(d|e)c^* \]

\[ A \rightarrow aAb \]
\[ A \rightarrow cA \]
\[ A \rightarrow \epsilon \]
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?

\[
\begin{align*}
a^*bc(d|e)c^* \\
\cdots
\end{align*}
\]

We can randomly rewrite nonterminals to sample:

\[
A \rightarrow aA \rightarrow aAb
\]
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?
    
    $a^*bc(d|e)c^*$

    $\begin{align}
    A & \rightarrow aAb \\
    A & \rightarrow cA \\
    A & \rightarrow \epsilon \\
    \end{align}$

    $\ldots$

    We can randomly rewrite nonterminals to sample:
    $A \rightarrow aAb \rightarrow aaAbb$
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?

\[
a^*bc(d | e)c^*
\]

\[
A \rightarrow aAb
\]
\[
A \rightarrow cA
\]
\[
A \rightarrow \varepsilon
\]

...
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?

\[ a^*bc(d|e)c^* \]

\[ A \rightarrow aAb \]
\[ A \rightarrow cA \]
\[ A \rightarrow \epsilon \]

- Simple textual grammars may not suffice.
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?

\[
a^*bc(d|e)c^*
\]

\[
\begin{align*}
A & \rightarrow aAb \\
A & \rightarrow cA \\
A & \rightarrow \epsilon
\end{align*}
\]

- Simple textual grammars may not suffice.
  - What about binary file formats? Wire protocols?
Generational Fuzz Testing

- Sample inputs from a model of the input space
  - What might a model be in this case?
    
    \[ a^*bc(d|e)c^* \]
    
    \[ A \rightarrow aAb \]
    \[ A \rightarrow cA \]
    \[ A \rightarrow \epsilon \]
    
    \[ \ldots \]

- Simple textual grammars may not suffice.
  - What about binary file formats? Wire protocols?
  - Specifications may include richer information about values, structure, and dependences
Generational Fuzz Testing

- Example: Peach Fuzzer (peachfuzzer.com)
Generational Fuzz Testing

- **Example: Peach Fuzzer (peachfuzzer.com)**
  - Specifications are provided through “peach pits”
  - XML specifications of both protocols & data
Generational Fuzz Testing

- **Example: Peach Fuzzer (peachfuzzer.com)**
  - Specifications are provided through “peach pits”
  - XML specifications of both protocols & data
  - e.g.
    (https://github.com/MozillaSecurity/peach/blob/master/Pits/Files/WebVTT/vtt.xml)

```xml
<DataModel name="_Timestamp">
  <String name="Hour">
    <Hint name="NumericalString" value="true"/>
  </String>
  <String name="Separator" value=":" token="true"/>
  <String name="Minute">
    <Hint name="NumericalString" value="true"/>
  </String>
  <String name="Period" value="." token="true"/>
  <String name="Second">
    <Hint name="NumericalString" value="true"/>
  </String>
</DataModel>
```
Mutational Fuzz Testing

- Given a corpus of inputs, *evolve* new inputs using fitness heuristics
Mutational Fuzz Testing

- **Given a corpus of inputs, evolve new inputs using fitness heuristics**
  - Even an empty corpus may suffice:
Mutational Fuzz Testing

- Given a corpus of inputs, evolve new inputs using fitness heuristics
  - Even an empty corpus may suffice:

  Pulling JPEGS out of thin air
  [Zalewski, 2014]
Mutational Fuzz Testing

- Given a corpus of inputs, **evolve** new inputs using fitness heuristics
  - Even an empty corpus may suffice:
    **Pulling JPEGS out of thin air**
    [Zalewski, 2014]
  - The power comes from the fitness heuristics
Mutational Fuzz Testing

• Given a corpus of inputs, evolve new inputs using fitness heuristics
  – Even an empty corpus may suffice:
    Pulling JPEGs out of thin air
    [Zalewski, 2014]
  – The power comes from the fitness heuristics

• Coverage Guided Fuzzing (CGF)
  – Use some notion of test coverage
  – Evolve a test suite toward more coverage
Mutational Fuzz Testing

- Given a corpus of inputs, evolve new inputs using fitness heuristics

```
S <- initial corpus
total_coverage <- {}
repeat
  for i in S:
    if sample P(i) then
      i' <- mutate(i)
      coverage <- execute(i')
      if coverage not in total_coverage:
        S <- S and {i'}
        total_coverage.add(coverage)
  until timeout
return S
```

This is just the big picture. Many optimizations complicate an implementation.
Mutational Fuzz Testing

- Given a corpus of inputs, **evolve** new inputs using fitness heuristics

```python
S <- initial corpus
total_coverage <- {}
repeat
    for i in S:
        if sample P(i) then
            i' <- mutate(i)
            coverage <- execute(i')
            if coverage not in total_coverage:
                S <- S and {i'}
                total_coverage.add(coverage)
until timeout
return S
```

This is just the big picture. Many optimizations complicate an implementation.
Mutational Fuzz Testing

- Given a corpus of inputs, **evolve** new inputs using fitness heuristics

```python
S <- initial corpus
total_coverage <- {}
repeat
    for i in S:
        if sample P(i) then
            i' <- mutate(i)
            coverage <- execute(i')
            if coverage not in total_coverage:
                S <- S and {i'}
                total_coverage.add(coverage)
    until timeout
return S
```

This is just the big picture. Many optimizations complicate an implementation.
Mutational Fuzz Testing

- Given a corpus of inputs, evolve new inputs using fitness heuristics

\[ S \leftarrow \text{initial corpus} \]
\[ \text{total\_coverage} \leftarrow \{\} \]
repeat
  for i in S:
    if sample P(i) then
      i' \leftarrow \text{mutate}(i)
      coverage \leftarrow \text{execute}(i')
      if coverage not in total\_coverage:
        S \leftarrow S \text{ and } \{i'\}
        total\_coverage.add(coverage)
  until timeout
until timeout
return S
### Mutational Fuzz Testing

- **Given a corpus of inputs,** *evolve* new inputs using fitness heuristics

```python
S <- initial corpus
total_coverage <- {}
repeat
    for i in S:
        if sample P(i) then
            i' <- mutate(i)
            coverage <- execute(i')
            if coverage not in total_coverage:
                S <- S and {i'}
                total_coverage.add(coverage)
    until timeout
return S
```
Mutational Fuzz Testing

• Given a corpus of inputs, *evolve* new inputs using fitness heuristics

```
S <- initial corpus
total_coverage <- {}
repeat
  for i in S:
    if sample P(i) then
      i' <- mutate(i)
      coverage <- execute(i')
      if coverage not in total_coverage:
        S <- S and {i'}
        total_coverage.add(coverage)
  until timeout
return S
```
Mutational Fuzz Testing

- Given a corpus of inputs, evolve new inputs using fitness heuristics

\[
\begin{align*}
S & \leftarrow \text{initial corpus} \\
total\_coverage & \leftarrow \{\} \\
\text{repeat} \\
\quad \text{for } i \text{ in } S: \\
\quad \quad \text{if sample } P(i) \text{ then} \\
\quad \quad \quad i' & \leftarrow \text{mutate}(i) \\
\quad \quad \text{coverage} & \leftarrow \text{execute}(i') \\
\quad \quad \text{if coverage not in total\_coverage:} \\
\quad \quad \quad S & \leftarrow S \text{ and } \{i'\} \\
\quad \text{total\_coverage}.\text{add}(coverage) \\
\text{until timeout} \\
\text{return } S
\end{align*}
\]
Mutational Fuzz Testing

- Given a corpus of inputs, evolve new inputs using fitness heuristics

```python
S <- initial corpus
total_coverage <- {}
repeat
  for i in S:
    if sample P(i) then
      i' <- mutate(i)
      coverage <- execute(i')
      if coverage not in total_coverage:
        S <- S and {i'}
        total_coverage.add(coverage)
  until timeout
return S
```
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```java
void foo(char a, char b) {
    if (a > 127) {
        ...
    } else {
        ...
    }
    if (b > 127) {
        ...
    } else {
        ...
    }
}
```

I1: (0,0)  I2: (200,200)
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```c
void foo(char a, char b) {
    if (a > 127) {
        ...
        else {
            ...
        }
    } else {
        ...
    }
    if (b > 127) {
        ...
    } else {
        ...
    }
}
```

I1: (0,0)  I2: (200,200)  I3: (0,200)
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```c
void foo(char a, char b) {
    if (a > 127) {  
        ...
    } else { 
        ...
    }
    if (b > 127) {  
        ...
    } else {  
        ...
    }
}
```
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```java
void foo(long a, long b) {
    if (a == 112358) {
        ...
    } else {
        ...
    }
    if (b == 4879235) {
        ...
    } else {
        ...
    }
}
```

Covering both true branches feels like finding a needle in a haystack!

What can we do?
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```c
void foo(long a, long b) {
    if (a == 112358) {
        ...
    } else {
        ...
    }
    if (b == 4879235) {
        ...
    } else {
        ...
    }
}
```
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```c
void foo(long a, long b) {
    if (a == 112358) {
        ...
    } else {
        ...
    }
    if (b == 4879235) {
        ...
    } else {
        ...
    }
}
```

Adding notions of coverage can steer the evolution however we desire.
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```c
void foo(long a, long b) {
    if (a == 112358) {
        ...
    } else {
        ...
    }
    if (b == 4879235) {
        ...
    } else {
        ...
    }
}
```

I1: 48 bits 63 bits
I2: 37 bits 54 bits
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```c
void foo(long a, long b) {
    if (a == 112358) {
        ...
    } else {
        ...
    }
    if (b == 4879235) {
        ...
    } else {
        ...
    }
}
```

```
I1: 48 bits 63 bits 64 bits similar
I2: 37 bits 54 bits 64 bits similar
I3: 
```
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```c
void foo(long a, long b) {
    if (a == 112358) {
        ...
    } else {
        ...
    }
    ...
}
```

Compilers can transform a program to make it amenable to testing!
Even simple coverage heuristics are powerful

- Let us consider just statement coverage

```c
void foo(long a, long b) {
    if (a == 112358) {
        ...
    } else {
        ...
    }
}
```

```c
void foo(long a, long b) {
    if (byte0(a) == 0xE6 && byte1(a) == 0xB6 && byte2(a) == 0x01 && byte4(a) == 0x00) {
        ...
    } else {
        ...
    }
}
```

Compilers can transform a program to make it amenable to testing!
Domain specific heuristics enable custom fuzzers

- Computational overhead/denial of service
  - Count per instruction frequency in coverage
Domain specific heuristics enable custom fuzzers

- Computational overhead/denial of service
  - Count per instruction frequency in coverage

- Memory consumption
  - Count allocated memory per allocation site
  - Automatically generates PNG bombs in practice!
Domain specific heuristics enable custom fuzzers

- Computational overhead/denial of service
  - Count per instruction frequency in coverage

- Memory consumption
  - Count allocated memory per allocation site
  - Automatically generates PNG bombs in practice!

- Energy consumption?
  - Measure power consumption over, e.g. tasks
Domain specific heuristics enable custom fuzzers

- Computational overhead/denial of service
  - Count per instruction frequency in coverage
- Memory consumption
  - Count allocated memory per allocation site
  - Automatically generates PNG bombs in practice!
- Energy consumption?
  - Measure power consumption over, e.g. tasks
- REST API invocations
  - Measure diversity of requests fed to server
Domain specific heuristics enable custom fuzzers

- Computational overhead/denial of service
  - Count per instruction frequency in coverage

- Memory consumption
  - Count allocated memory per allocation site
  - Automatically generates PNG bombs in practice!

- Energy consumption?
  - Measure power consumption over, e.g. tasks

- REST API invocations
  - Measure diversity of requests fed to server

- ...

American Fuzzy Lop

- (AFL) is one commonly used fuzzer that was supported by Google

Let's see an example.
The Oracle Problem

- We have referred to this as random testing, but what are our oracles?
The Oracle Problem

- We have referred to this as random testing, but what are our oracles?
- **Common universal oracles**
  - Never crash
  - No undefined behavior
  - No failures from dynamic analysis tools X, Y, or Z
The Oracle Problem

- We have referred to this as random testing, but what are our oracles?

- Common universal oracles
  - Never crash
  - No undefined behavior
  - No failures from dynamic analysis tools X, Y, or Z

- Differential Testing
  - Feed input into N different implementations & vote
  - Feed input into N configurations of one implementation & vote
  - This is a major approach in modern compiler testing!
The Oracle Problem

- We have referred to this as random testing, but what are our oracles?

- Common universal oracles
  - Never crash
  - No undefined behavior
  - No failures from dynamic analysis tools X, Y, or Z

- Differential Testing
  - Feed input into N different implementations & vote
  - Feed input into N configurations of one implementation & vote
  - This is a major approach in modern compiler testing!

- Metamorphic Testing
  - Identify key properties that enable correct results to be known relative to mutations (e.g. graphics drivers, machine learning, ...
Other challenges in fuzzing

- Highly structured inputs require more care
  - Grammar + CGF hybrids
  - Input generators
  - ...

Other challenges in fuzzing

- Highly structured inputs require more care
  - Grammar + CGF hybrids
  - Input generators
  - ...

- Making use of nuanced oracles can be challenging in practice
Other challenges in fuzzing

- Highly structured inputs require more care
  - Grammar + CGF hybrids
  - Input generators
  - ...

- Making use of nuanced oracles can be challenging in practice

- It can be most effective at a whole program or single function level
Feedback Directed Random Testing

- In practice, *input* fuzzing may not apply
Feedback Directed Random Testing

- In practice, *input* fuzzing may not apply
  - What if the thing we want to test is an API rather than a program?
  - What if it is an object oriented API?
Feedback Directed Random Testing

- In practice, *input* fuzzing may not apply
  - What if the thing we want to test is an API rather than a program?
  - What if it is an object oriented API?

- It can be preferable to generate some other model of behavior
Feedback Directed Random Testing

• In practice, *input* fuzzing may not apply
  – What if the thing we want to test is an API rather than a program?
  – What if it is an object oriented API?

• It can be preferable to generate some other model of behavior

• Feedback Directed Random Testing
Feedback Directed Random Testing

- In practice, *input* fuzzing may not apply
  - What if the thing we want to test is an API rather than a program?
  - What if it is an object oriented API?

- It can be preferable to generate some other model of behavior

- **Feedback Directed Random Testing**
  - Consider a unit test with *Arranging, Acting, and Asserting*
Feedback Directed Random Testing

- In practice, *input* fuzzing may not apply
  - What if the thing we want to test is an API rather than a program?
  - What if it is an object oriented API?

- It can be preferable to generate some other model of behavior

- **Feedback Directed Random Testing**
  - Consider a unit test with *Arranging*, *Acting*, and *Asserting*
  - Generate a sequence of such operations randomly to explore API behavior
Feedback Directed Random Testing

- In practice, *input* fuzzing may not apply
  - What if the thing we want to test is an API rather than a program?
  - What if it is an object oriented API?

- It can be preferable to generate some other model of behavior

- **Feedback Directed Random Testing**
  - Consider a unit test with *Arranging*, *Acting*, and *Asserting*
  - Generate a sequence of such operations randomly to explore API behavior
  - Use coverage feedback again to guide the process
Feedback Directed Random Testing

- In practice, *input* fuzzing may not apply
  - What if the thing we want to test is an API rather than a program?
  - What if it is an object oriented API?

- It can be preferable to generate some other model of behavior

- Feedback Directed Random Testing
  - Consider a unit test with *Arranging*, *Acting*, and *Asserting*
  - Generate a sequence of such operations randomly to explore API behavior
  - Use coverage feedback again to guide the process

- Available through such tools as Randoop, GRT, ...
Feedback Directed Random Testing

TEST(...,...) {
    Triangle t{1,1,1};
}


Feedback Directed Random Testing

```cpp
TEST(...,...) {
    TEST(...,...) {
        Triangle t{1,1,1};
        t.isEquilateral();
    }
}
```
Feedback Directed Random Testing

```c
TEST(..., ...) {
    TEST(...) {
        TEST(..., ...) {
            Triangle t{1,1,1};
            t.isEquilateral();
            Triangle t2{1,2,1};
        }
    }
}
```
Feedback Directed Random Testing

```
TEST(...,...) {
    Triangle t{1,1,1};
    t.isEquilateral();
}

TEST(...,...) {
    Triangle t{1,2,1};
    t2.contains(t1);
}
```
Feedback Directed Random Testing

```cpp
TEST(...,...) {
    Triangle t{1,1,1};
    t.isEquilateral();
}
```

```cpp
TEST(...,...) {
    Triangle t{1,1,1};
    t.isEquilateral();
    Triangle t2{1,2,1};
    t2.contains(t1);
}
```

```cpp
TEST(...,...) {
    Triangle t{1,1,1};
    t.isEquilateral();
    Triangle t2{1,2,1};
    t2.contains(t1);
    ...
}
```
Challenges in Feedback Directed Random Testing

- What notions of coverage are good?
  - Sometimes a sequence extension does not add value
Challenges in Feedback Directed Random Testing

- What notions of coverage are good?
  - Sometimes a sequence extension does not add value

- Oracles, again
  - Simple contracts & exceptions are easy
  - Invariant violation?
  - Near invariants?
  - Alternate schedules?
Summary

- Random testing strategies provide a means of continuous testing
Summary

- Random testing strategies provide a means of continuous testing
- They can be surprisingly effective in practice
Summary

- Random testing strategies provide a means of continuous testing
- They can be surprisingly effective in practice
- **Effective application to a specific problem may require tailoring a tool**