Software Testing, Quality Assurance & Maintenance—Lecture 4

Patrick Lam
University of Waterloo

January 15, 2025

Part I

When to stop?
Idea 2: Mutation Analysis

How many tests?

Do you have enough tests? How do you know?

Let's fuzz the test suite.

How? Modifying the program and seeing if the test suite notices.

Mutants

A mutant is a modified version of the program being tested.

Usually we change an operator or identifier:

$$x + 5 \Rightarrow x - 5$$

Killing Mutants

The test suite should fail on the mutant.
Then the mutant is killed.

Remember: arrange, act, assert.
Mutant might trigger errors during act;
or it may detect different output during assert.

Example Mutants

Use language grammar to create mutants (code/L04/minval.c).

```
// original
int min(int a, int b)
{
  int minVal;
  minVal = a;

  if (b < a) {


    minVal = b;




  }
  return minVal;
}
```

```
// with mutants
int min(int a, int b)
{
  int minVal;
  minVal = a;
  minVal = b;                          // Δ 1
  if (b < a) {
  if (b > a) {                         // Δ 2
  if (b < minVal) {                    // Δ 3
    minVal = b;
    BOMB();                            // Δ 4
    minVal = a;                        // Δ 5
    minVal = failOnZero(b);            // Δ 6
  }
  return minVal;
}
```

Testing on the mutants

Here's a test suite. How do the mutants do?

	Δ 1	Δ 2	Δ 3	Δ 4	Δ 5	Δ 6
$\langle a = 0, b = 1, \exp = 0 \rangle$	kill		–			
$\langle a = 1, b = 0, \exp = 0 \rangle$	–		–			
$\langle a = 1, b = 1, \exp = 1 \rangle$			–			
$\langle a = 1, b = 349, \exp = 1 \rangle$			–			

Observe: $\Delta 3$ not killable.

Key idea for Mutation Analysis

Idea: use mutation analysis to evaluate test suite quality/improve test suites.

Good test suites ought to be effective at killing mutants.

Why should this work? (1/2)

Competent Programmer Hypothesis:

programmers usually are almost right, except for “subtle, low-level faults”.

Mutation analysis tries to mimic this.
(Exceptions?)

Why should this work? (2/2)

Coupling Effect Hypothesis:

complex faults are the result of simple faults combining.

Hence, detecting all simple faults will detect many complex faults.

Implication: test suites that are good at ensuring program quality also good at killing mutants.

Mutation analysis in context

Hard to apply by hand, and automation is complicated.

Mutation is a “gold standard”
against which to test other testing criteria.

Consider test suite T which ensures statement coverage.
What does mutation analysis say about T?

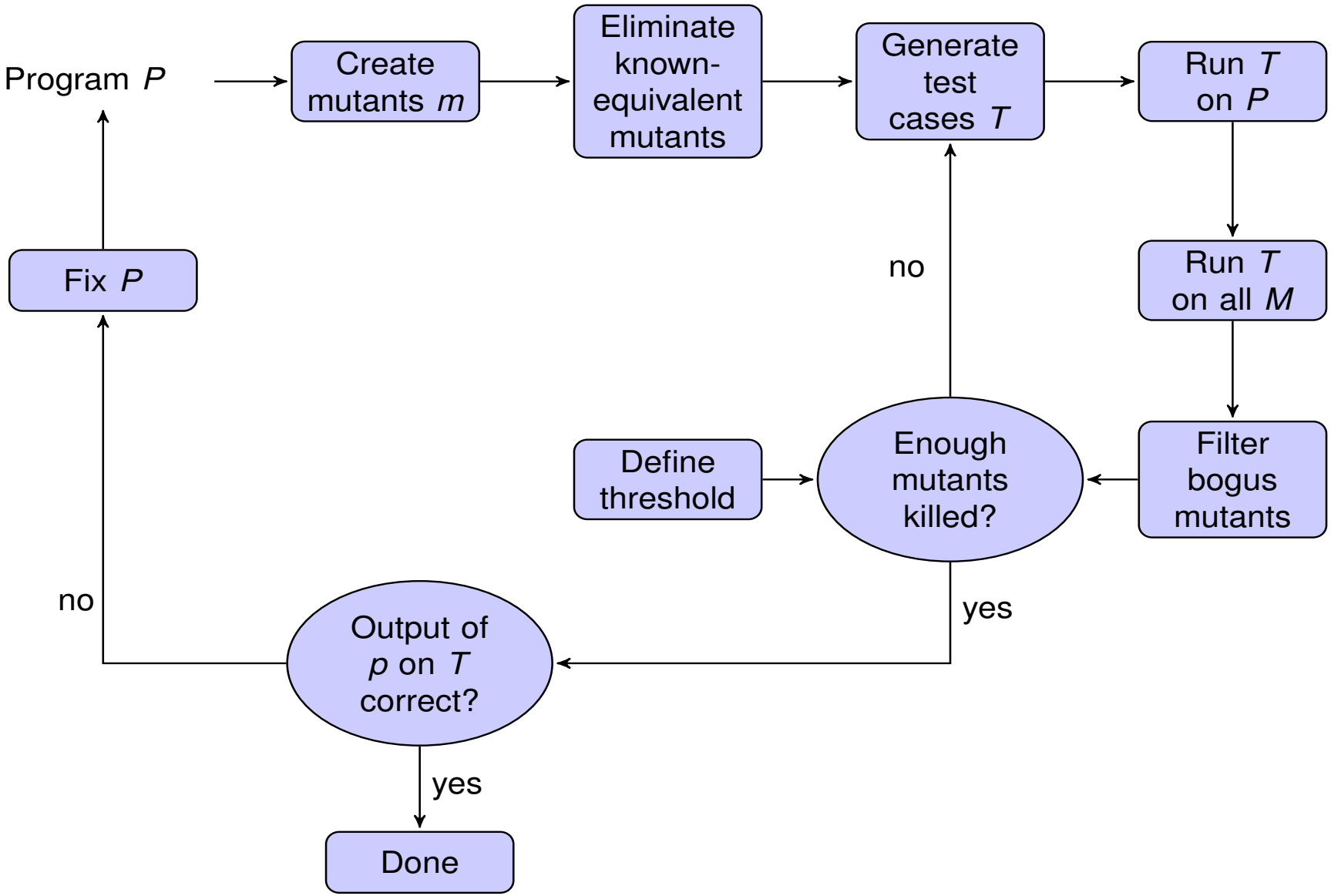
Using Mutation Analysis

Three steps:

1. Generate mutants (usually with a tool)
2. Execute mutants (computationally expensive)
3. Classify (manual)

Then, create new test cases to kill remaining mutants.

Mutation Analysis Diagram



Basic Block Definition

A basic block is a sequence of instructions in the control-flow graph that has one entry point and one exit point.

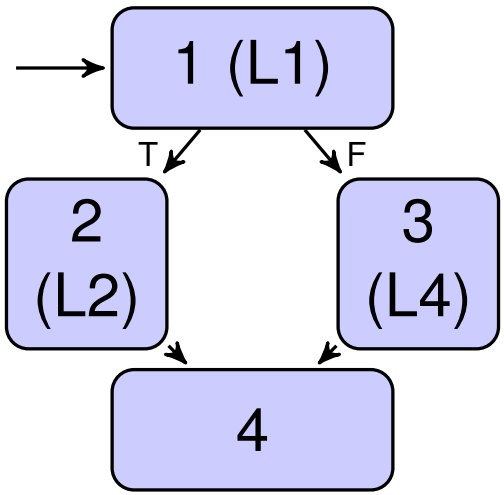
Usually want maximal basic blocks.

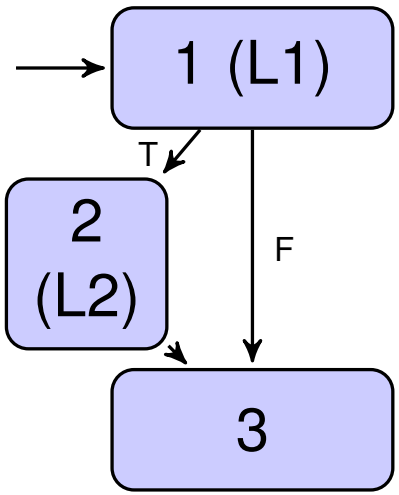
May have multiple successors.

No jumps into the middle of a basic block.

Constructing Basic Blocks: if

```
if (z < 17)
  print (x);
else
  print (y);
```



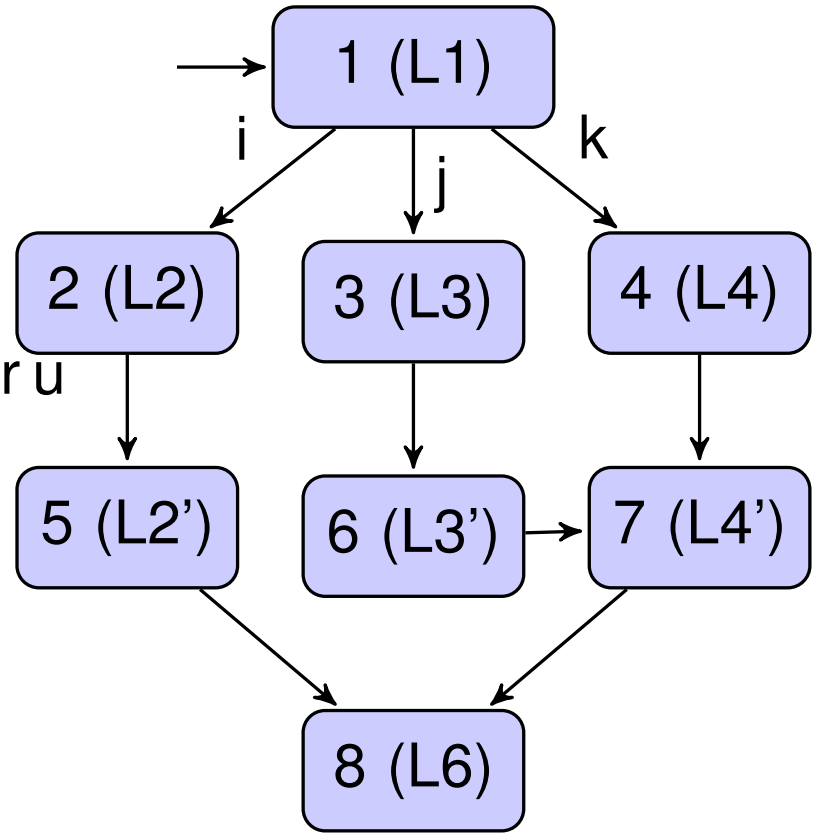



```
if (z < 17)
  print(x);
```

Not talking about short-circuit evaluation.

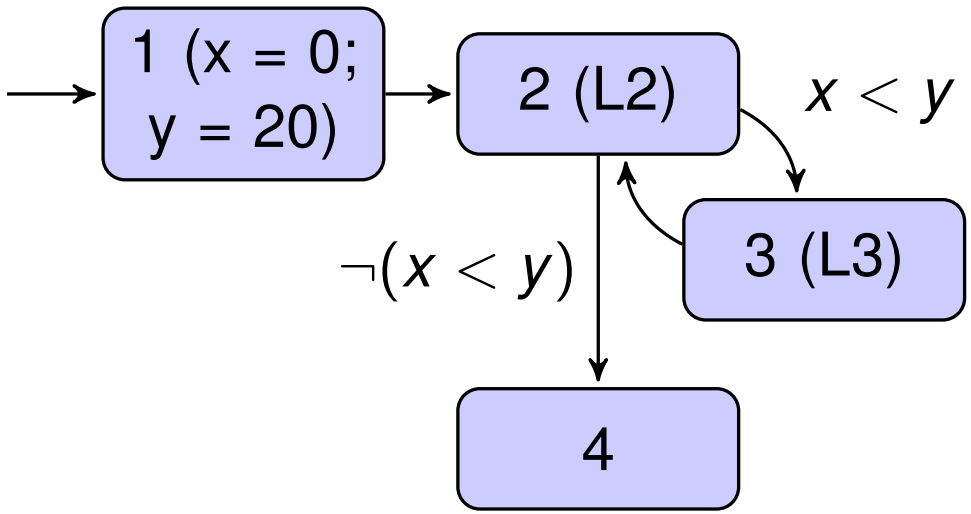
Constructing Basic Blocks: case/switch

```
switch (n) {
  case 'I': ...; break;
  case 'J': ...; //fallthru
  case 'K': ...; break;
}
// ...
```



Constructing Basic Blocks: while

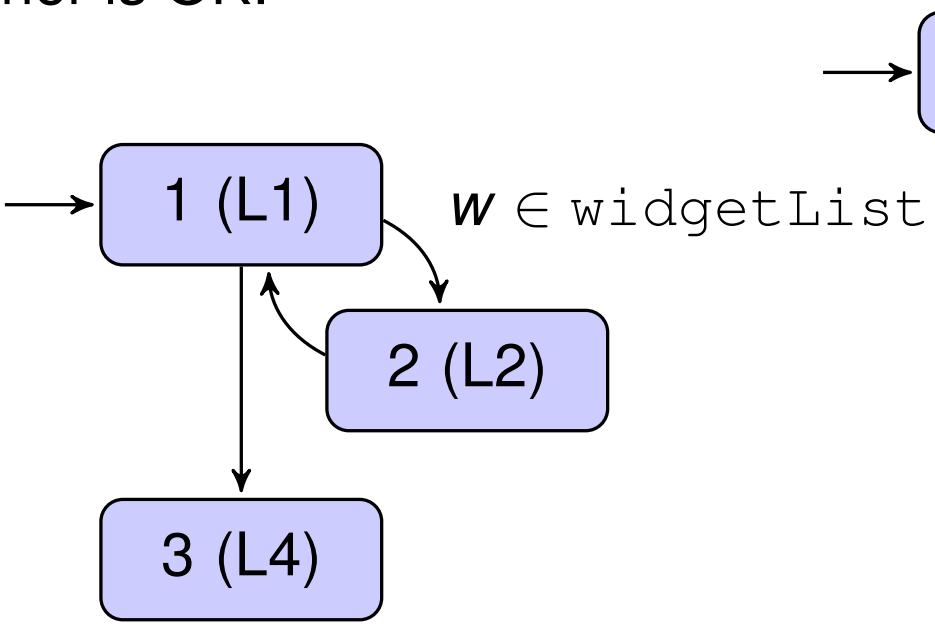
```
x = 0; y = 20;
while (x < y) {
  x ++; y --;
}
```

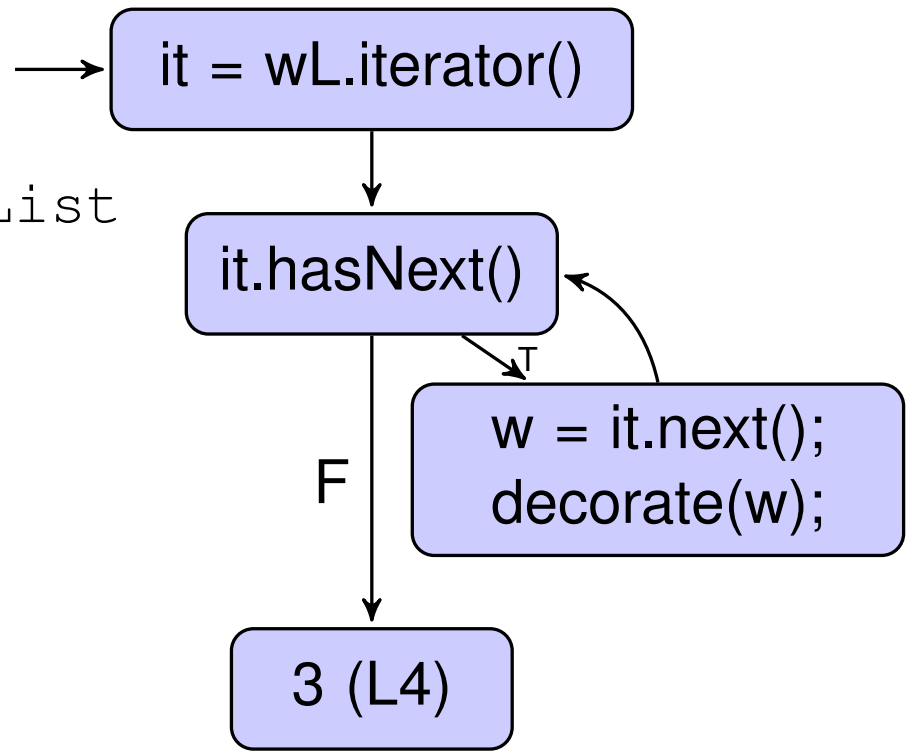


Constructing Basic Blocks: for

```
for (Widget w : widgetList) {
  decorate(w);
}
```

Either is OK:





Back to Statement and Branch Coverage

Given a test suite and a program,
instrument the program to:

- count whether each statement (CFG node) is executed;
- count whether each branch (CFG edge) is taken.

Statement coverage is the fraction of statements (nodes) that are executed by the test suite.

Branch coverage is the fraction of branches (edges) that are executed.

Example Code

```
class Foo:
    def m(self, a, b):
        if a < 0 and b < 0:
            return 4
        elif a < 0 and b > 0:
            return 3
        elif a > 0 and b < 0:
            return 2
        elif a >= 0 and b >= 0:
            return a/b
        raise Exception("I didn't think things through
```

Example Test Suite

```
import unittest

from .foo import Foo

class CoverageTests(unittest.TestCase):
    def test_one(self):
        f = Foo()
        f.m(1, 2)

    def test_two(self):
        f = Foo()
        f.m(1, -2)

    def test_three(self):
        f = Foo()
        f.m(-1, 2)
```

Coverage Report

```
Name                    Stmts   Miss  Branch  BrPart  Cover   Missi
-------------------------------------------------------------------
l03/foo.py                 11      2       8       2    79%   4, 11
l03/test_suite.py          12      0       0       0   100%
-------------------------------------------------------------------
TOTAL                     124     98      46       2    21%
```

HTML report also available.

On Coverage

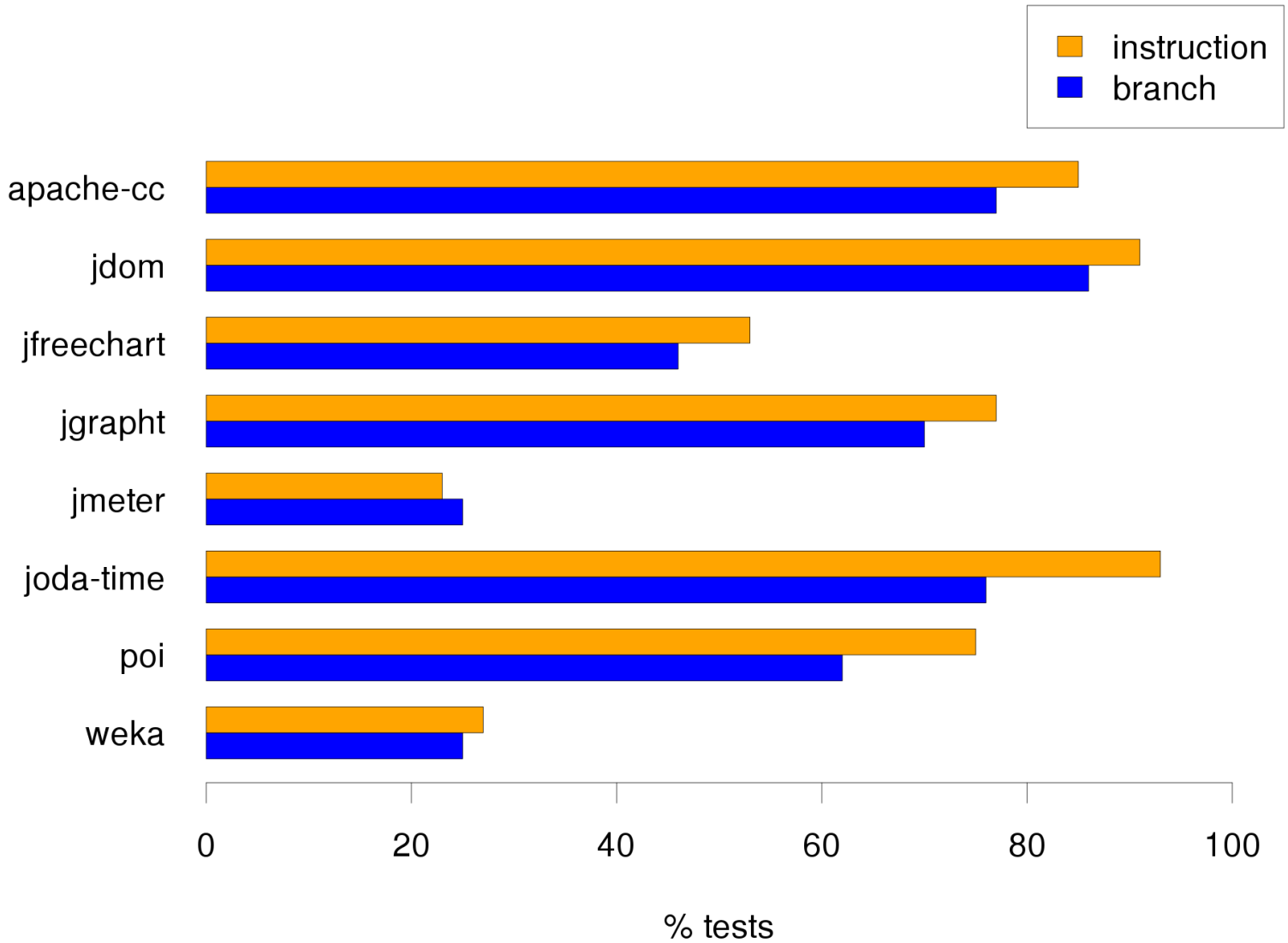
Can add missing test cases to visit all lines.

Even with 100% branch coverage,
one is missing an important behaviour: what if `b` is 0?

Infeasible Test Requirements

Infeasible to reach 100% coverage on real programs.
How much is enough, and why is there a gap?

Some Real Coverage Data



Case Study: JUnit (4.11) the Artifact

https://avandeursen.com/2012/12/21/line-coverage-lessons-from-junit/

JUnit Measurements

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	221	84%	2970/3513	81%	859/1060	1.727
junit.extensions	6	82%	52/63	87%	7/8	1.25
junit.framework	17	76%	399/525	90%	139/154	1.605
junit.runner	3	49%	77/155	41%	23/56	2.225
junit.textui	2	76%	99/130	76%	23/30	1.686
org.junit	14	85%	196/230	75%	68/90	1.655
org.junit.experimental	2	91%	21/23	83%	5/6	1.5
org.junit.experimental.categories	5	100%	67/67	100%	44/44	3.357
org.junit.experimental.max	8	85%	92/108	86%	26/30	1.969
org.junit.experimental.results	6	92%	37/40	87%	7/8	1.222
org.junit.experimental.runners	1	100%	2/2	N/A	N/A	1
org.junit.experimental.theories	14	96%	119/123	88%	37/42	1.674
org.junit.experimental.theories.internal	5	88%	98/111	92%	39/42	2.29
org.junit.experimental.theories.suppliers	2	100%	7/7	100%	2/2	2
org.junit.internal	11	94%	149/157	94%	53/56	1.947
org.junit.internal.builders	8	98%	57/58	92%	13/14	2
org.junit.internal.matchers	4	75%	40/53	0%	0/18	1.391
org.junit.internal.requests	3	96%	27/28	100%	2/2	1.429
org.junit.internal.runners	18	73%	306/415	63%	82/130	2.155
org.junit.internal.runners.model	3	100%	26/26	100%	4/4	1.5
org.junit.internal.runners.rules	1	100%	35/35	100%	20/20	2.111
org.junit.internal.runners.statements	7	97%	92/94	100%	14/14	2
org.junit.matchers	1	9%	1/11	N/A	N/A	1
org.junit.rules	20	89%	203/226	96%	31/32	1.444
org.junit.runner	12	93%	150/161	88%	30/34	1.378
org.junit.runner.manipulation	9	85%	36/42	77%	14/18	1.632
org.junit.runner.notification	12	100%	98/98	100%	8/8	1.162
org.junit.runners	16	98%	321/327	96%	95/98	1.737
org.junit.runners.model	11	82%	163/198	73%	73/100	1.918

Report generated by Cobertura 1.9.4.1 on 12/22/12 2:25 PM.

JUnit Stats

Overall instruction coverage: 85%.

13,000 lines of code, 15,000 lines of test.

Consistent with industry average.

What's not covered? Deprecation

- deprecated code: 65% instruction coverage
- nondeprecated code: 93% instruction coverage

- newer code (in `org.junit.*`): 90% instruction coverage
- older code (in `junit.*`): 70% instruction coverage

(Why is this? Perhaps the coverage decreased over time for the deprecated code, since no one is really maintaining it anymore, and failing test cases just get removed.)

A Whole Untested Class

Blogpost author found one class that was completely untested!

There were tests.
But the tests never got run, because they were never added to CI.

They also failed when run. (You don't run it, it doesn't work.)

The Usual Suspects 1: Too Simple to Test

```
public static void assumeFalse(boolean b) {
  assumeTrue(!b);
}

/**
* Override to set up your specific external reso
*
* @throws if setup fails (which will disable {@c
*/
protected void before() throws Throwable {
  // do nothing
}
```

The Usual Suspects 2: Dead by Design

```
/**
* Protect constructor since it is a static only
*/
protected Assert() { }

// should never be executed:
catch (InitializationError e) {
  throw new RuntimeException(
  "Bug in saff's brain: " +
  "Suite constructor, called as above, should alw
}

// unreachable
try {
  ...
} catch (InitializationError e) {
  return new ErrorReportingRunner(null, e); // u
}
```

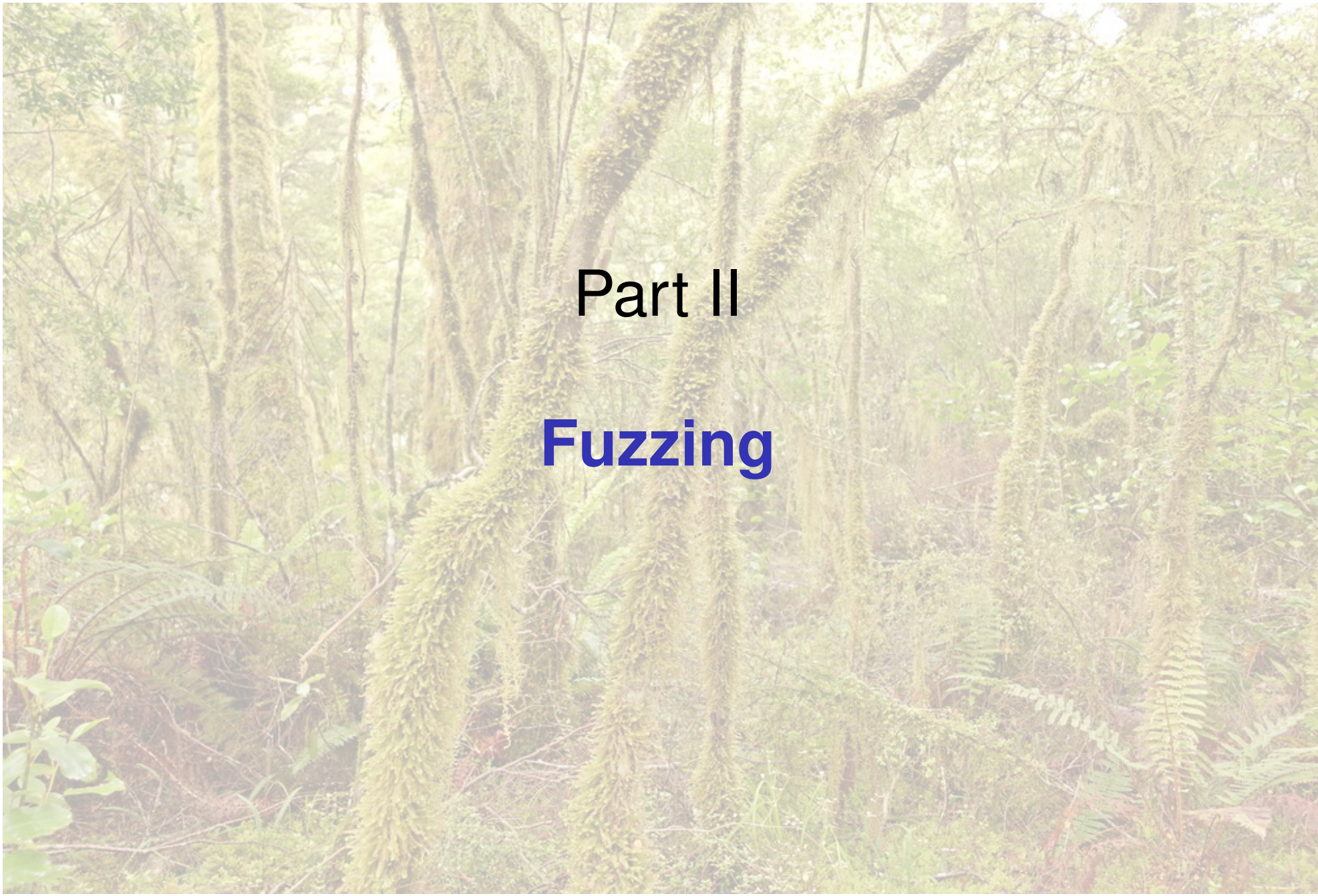
Thoughts on JUnit Coverage

JUnit: written by people who care about testing.

Non-deprecated code: 93% instruction coverage,
i.e. $\leq$ 2–3 untested lines of code per method.

Probably OK to have lower coverage for deprecated code.

Don't forget that what is in the tests matters too!



Part II

Fuzzing

Some JavaScript Code

```
function test() {
    var f = function g() {
        if (this != 10) f();
    };
    var a = f();
}
test();
```

Huh?

- this code used to crash WebKit (`https://bugs.webkit.org/show_bug.cgi?id=116853`).
- automatically generated by the Fuzzinator tool, based on a grammar for JavaScript.

Fuzzing effectively finds software bugs, especially security-based bugs (e.g. insufficient input validation.)

Fuzzing Origin Story

- 1988.
- Prof. Barton Miller was using a modem,
  on a dark and stormy night.
- Line noise caused UNIX utilities to crash!

Fuzzing Origin Story Part 2

- he got grad students in his Advanced Operating Systems class to write a fuzzer
  (generating unstructured ASCII random inputs)
- result: 25%-33% of UNIX utilties crashed on random inputs[1]

[1] http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html

(An earlier use of Fuzzing)

- 1983: Apple's "The Monkey[2]"
- Generated random events for MacPaint, MacWrite.
- Found lots of bugs,
  but eventually the monkey hit the Quit command.

- Solution: "MonkeyLives" system flag, ignore Quit.

[2] http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt

How Fuzzing Works

Two kinds of fuzzing:

- mutation-based: start with existing, randomly modify
- generation-based: start with grammar, generate inputs

What you do:

- feed randomly-generated inputs to the program;
- look for crashes or assertion errors;
- or run under a dynamic analysis tool (e.g. Valgrind) and observe runtime errors.

Level 0 Fuzzing

Generation-based testing for HTML5.

Use the regular expression:

```
.*
```

that is: “any character”, “0 or more times”.

Found a WebKit assertion failure:
`https://bugs.webkit.org/show_bug.cgi?id=132179`.
Process:

- Take the regular expression and generate random strings from it.
- Feed them to the browser and see what happens.
- Find an assertion failure/crash.

Hierarchy of inputs: C

1. sequence of ASCII characters;
2. sequence of words, separators, and white space (gets past the lexer);
3. syntactically correct C program (gets past the parser);
4. type-correct C program (gets past the type checker);
5. statically conforming C program (starts to exercise optimizations);
6. dynamically conforming C program;
7. model conforming C program.

Each level is a subset of previous level, but more likely to find interesting inputs specific to the system.

Operate at all the levels.

Mutation-based Fuzzing

Develop a tool that randomly modifies existing inputs:

- totally randomly, by flipping bytes in the input; or,
- parse the input and then change some of the nonterminals.

If you flip bytes, you also need to update any applicable checksums if you want to see anything interesting (similar to level 3 above).

Quote from Fuzzinator author

More than a year ago, when I started fuzzing, I was mostly focusing on mutation-based fuzzer technologies since they were easy to build and pretty effective. Having a nice error-prone test suite (e.g. LayoutTests) was the warrant for fresh new bugs. At least for a while.

As expected, the test generator based on the knowledge extracted from a finite set of test cases reached the edge of its possibilities after some time and didn't generate essentially new test cases anymore.

At this point, a fuzzer girl can reload her gun with new input test sets and will probably find new bugs. This works a few times but she will soon find herself in a loop testing the common code paths and running into the same bugs again and again.[3]

[3] `http://webkit.sed.hu/blog/20141023/fuzzinator-reloaded`

Fuzzing Summary

Fuzzing finds interesting test cases.

Works best at interfaces between components.

Advantages: it runs automatically and really works.
Disadvantages: without significant work, it won't find sophisticated domain-specific issues.

Related: Chaos Monkey

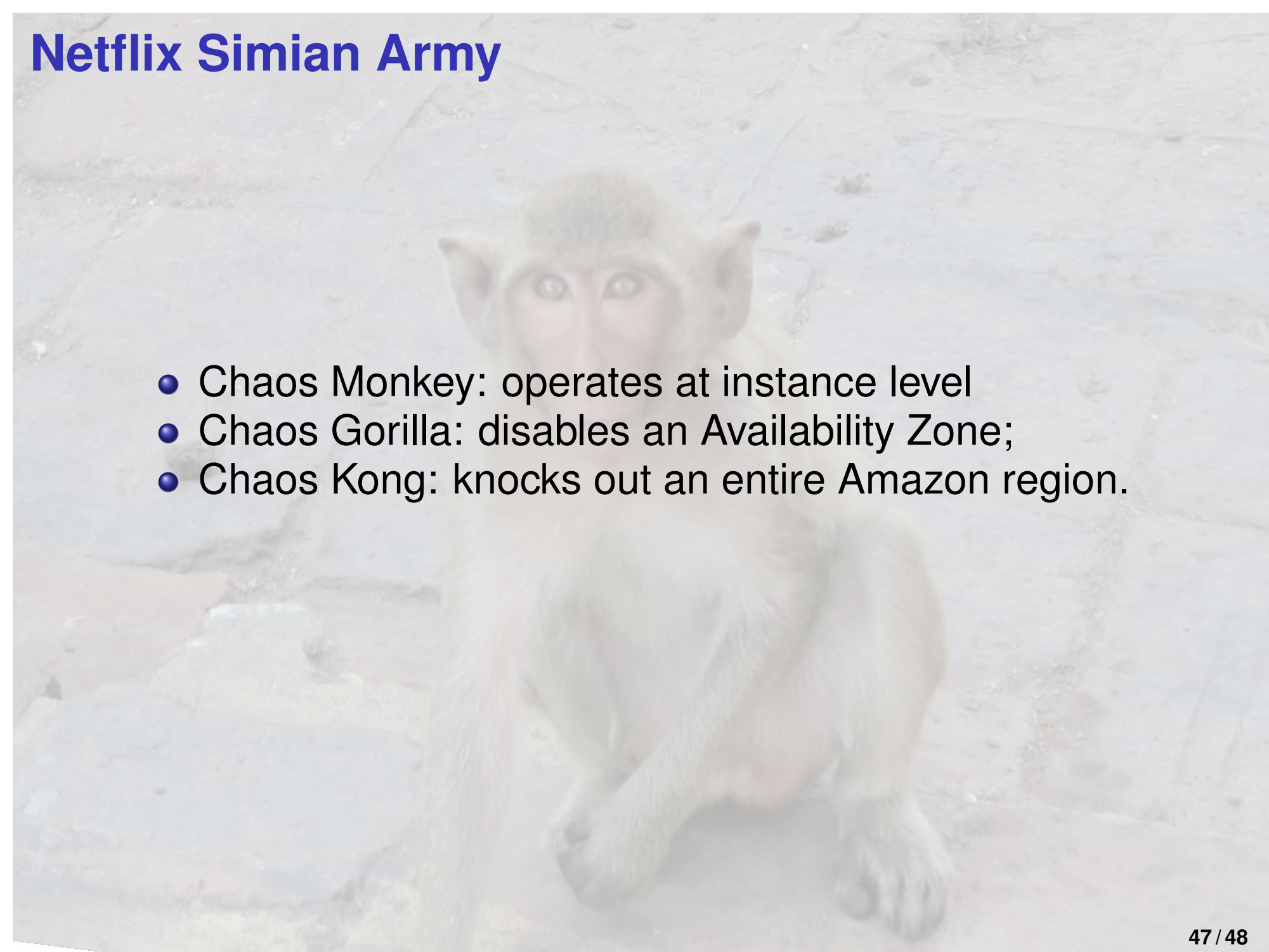
Instead of inputs, think distributed systems.

Some instances (components) randomly fail
(because of bogus inputs, or . . .).

Ideally: system continues to work.

Failures are inevitable, need a strategy to deal with,
better to encounter not-at-4am.

Netflix Simian Army



- Chaos Monkey: operates at instance level
- Chaos Gorilla: disables an Availability Zone;
- Chaos Kong: knocks out an entire Amazon region.

Jeff Atwood Quotes

Why inflict such a system on yourself?
“Sometimes you don’t get a choice; the Chaos Monkey chooses you.”

Software engineering benefits:

- “Where we had one server performing an essential function, we switched to two.”
- “If we didn’t have a sensible fallback for something, we created one.”
- “We removed dependencies all over the place, paring down to the absolute minimum we required to run.”
- “We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available.”