

Testing

Program testing can be used to show the presence of bugs, but never to show their absence! -- Edsger Dijkstra



The Many Purposes of Testing

Not only are tests used to drive software design, but **we design our software for testing** (later in this lecture).

- **Find bugs**

- Hard to prove of the absence of bugs (Dijkstra)

- **Prevent bugs** from sneaking in during enhancement
(Regression Testing)

- Loose synchronization among developers/teams can result in incorrect use or enhancement of existing code

- Give **high confidence** in the integrity of your product

- **Explore class/method design** (Test-First/Test-Driven Development and/or DbC)

- **Specification of expected behavior**

The THREE BIG IDEAS of Software Testing

Coverage: Seek to execute all possibilities.

Done naïvely, that's too many so...

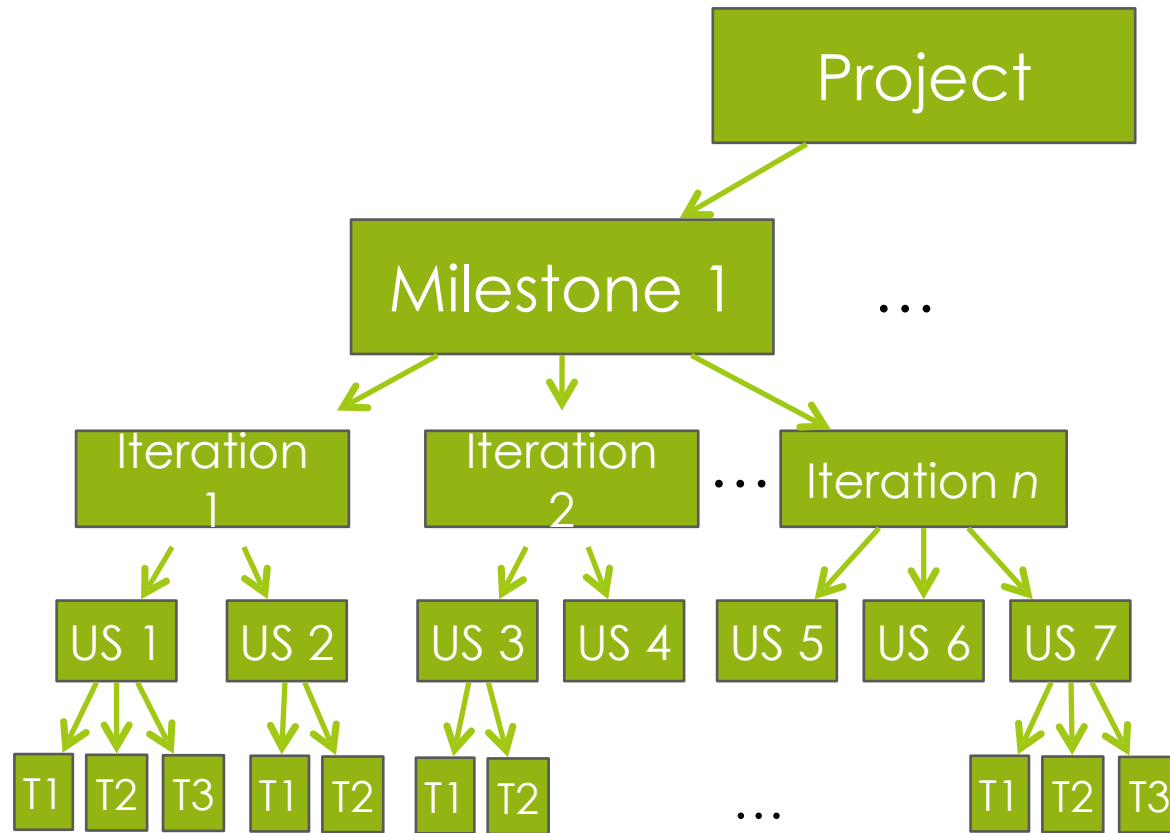
Test Equivalence Classes: No need to write/run two tests that expose the same bug. If you have a “theory” for the bug, then you can tell if two tests reveal the same bug – i.e., from the same class. **Cover all eq. classes, not inputs.**

That's still too many, so...

Bottom-Up Testing: When testing if something works, its parts should already be tested. **We test just the current level,** reducing the explosion of combinations.

Bottom-Up Testing and the Hierarchical Structure of Agile Planning and Delivery

For example, Iteration testing assumes that the individual Stories/Features work, and tests how the Stories glue together.



- ← User, System Testing
(perf, robustness, user experience)
(i.e., End-to-End Scenarios + Personas)
- ← Acceptance Testing
(customer demo, End-to-End Scenarios)
- ← Story Testing (features)
(BDD acceptance tests)
- ← Unit Testing (methods)
(black/gray/white box)

Each level of testing assumes all the lower levels of tests have passed. Only test for the “current-level” risk.

Agile Testing: Hierarchical, Diverse (80/20)

- Write (and run ;) three kinds of tests, *bottom up*:
 1. Task level: Unit tests for critical units (b-box and/or w-box)
 2. Story/Iteration-level: BDD scenario tests (in unit or BDD tester)
 - Automating all could be expensive; some by hand
 3. Iteration/Milestone-level: End-to-end Scenario tests (“run” by hand - already done, from product design)
 - Additionally consider Personas, platforms/configurations, real people
- Diversification beyond the hierarchy:
 - Asserts from DbC
 - Logging for hard-to-test code (grey-box)

Include time for testing during Planning

Write tests for high-risk units

For each Story, have a testing Task (BDD Scenario Tests)

- ▣ Could have two: one for writing tests, one for passing

For an Iteration, have a testing Story or “loose” Task

- ▣ This is a “Developer Story”: As a developer, I want...
- ▣ End-to-End Scenarios, e.g.

For Milestone, have a testing Iteration or loose Story/Task

- ▣ longer End-to-End Scenarios, e.g.

Testing early-stage software

- You want to test module A
- But A depends on module B.
- Module B isn't ready yet.
- What do?



Another situation

- Want to test code that depends on the current time
- Or the network
- Or the disk
- Now what?

Solution: mocking

■ New class: MockCalendar

```
class MockCalendar extends Calendar {  
    long millis;  
    MockCalendar(long millis) {this.millis = millis;}  
    static MockCalendar getInstance()  
        { return new MockCalendar(millis); }  
    long getTimeInMillis() { return millis; }  
    void setTimeInMillis(long ms) { millis = ms; }  
    ... // Lots of stubbed methods that we don't use  
}
```

Pass MockCalendar instance into code to be tested.

Advanced Testing

Or: how to avoid writing tests manually (sometimes)

Puzzle: Find x such $p1(x)$ returns True

```
def p1(x):  
    if x * x - 10 == 15:  
        return True  
    return False
```

Puzzle: Find x such p2(x) returns True

```
def p2(x):  
    if x > 0 and x < 1000:  
        if ((x - 32) * 5/9 == 100):  
            return True  
    return False
```

Puzzle: Find x such p3(x) returns True

```
def p3(x):  
    if x > 3 and x < 100:  
        z = x - 2  
        c = 0  
        while z >= 2:  
            if z ** (x - 1) % x == 1:  
                c = c + 1  
                z = z - 1  
            if c == x - 3:  
                return True  
        return False
```

Fuzz Testing

Security and Robustness

Barton P. Miller, Lars Fredriksen and Bryan So

Study of the Reliability of UNIX Utilities

COMMUNICATIONS OF THE ACM / December 1990 / Vol.33, No.12

33

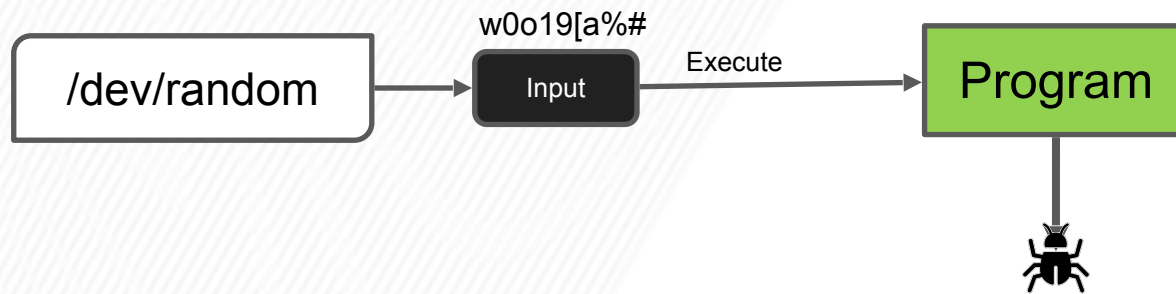
Communications of the ACM (1990)

“

On a dark and stormy night one of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. The author had to race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash.

”

Fuzz Testing



A 1990 study found crashes in:

adb, as, bc, cb, col, diction, emacs, eqn, ftp, indent, lex, look, m4, make, nroff, plot, prolog, ptx, refer!, spell, style, tsort, uniq, vgrind, vi

Common Fuzzer-Found Bugs in C/C++

Causes: incorrect arg validation, incorrect type casting, executing untrusted code, etc.

Effects: buffer-overflows, memory leak, division-by-zero, use-after-free, assertion violation, etc. (“crash”)

Impact: security, reliability, performance, correctness

How do you make programs “crash” when a bug is encountered?

Automatic Oracles: Sanitizers

- Address Sanitizer (ASAN) ***
- LeakSanitizer (comes with ASAN)
- Thread Sanitizer (TSAN)
- Undefined-behavior Sanitizer (UBSAN)

<https://github.com/google/sanitizers>

AddressSanitizer

Compile with `clang -fsanitize=address`

```
int get_element(int* a, int i) {  
    return a[i];  
}
```

Is it null?

```
int get_element(int* a, int i) {  
    if (a == NULL) abort();  
    return a[i];  
}
```

Is the access out of bounds?

```
int get_element(int* a, int i) {  
    if (a == NULL) abort();  
    region = get_allocation(a);  
    if (in_heap(region)) {  
        low, high = get_bounds(region);  
        if ((a + i) < low || (a + i) > high) {  
            abort();  
        }  
    }  
    return a[i];  
}
```

Is this a reference to a stack-allocated variable after return?

```
int get_element(int* a, int i) {  
    if (a == NULL) abort();  
    region = get_allocation(a);  
    if (in_stack(region)) {  
        if (popped(region)) abort();  
        ...  
    }  
    if (in_heap(region)) { ... }  
    return a[i];  
}
```

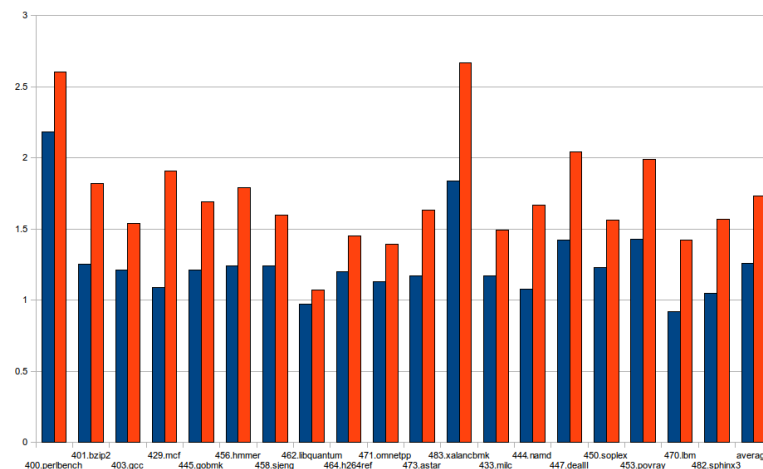
AddressSanitizer

<https://github.com/google/sanitizers/wiki/AddressSanitizer>

Asan is a memory error detector for C/C++. It finds:

- Use after free (dangling pointer dereference)
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

Slowdown about 2x on SPEC CPU 2006



Strengths and Limitations

Strengths:

- Cheap to generate inputs

- Easy to debug when a failure is identified

Limitations:

- Randomly generated inputs don't make sense most of the time.

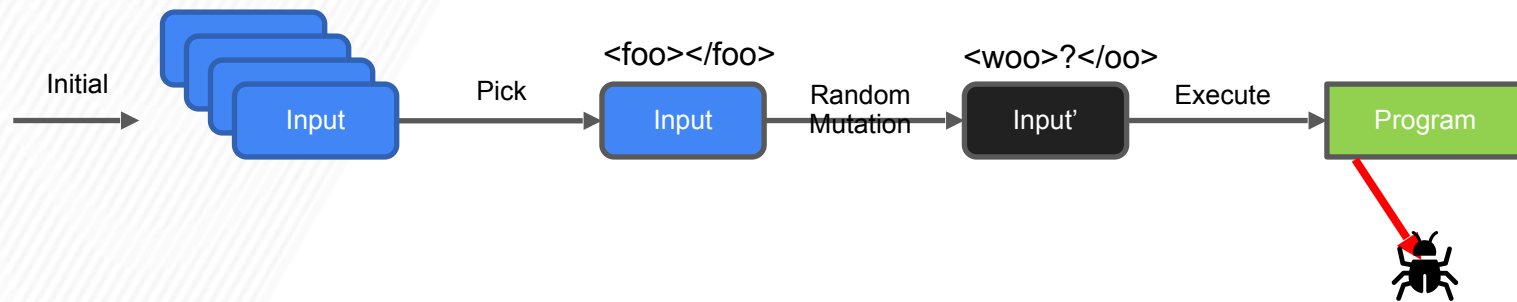
 - E.g. Imagine testing a browser and providing some "input" HTML randomly:

 - dgsad5135o gsd;gj lsdkg3125j@!T%#(W+123sd asf j**

- Unlikely to exercise interesting behavior in the web browser

- Can take a long time to find bugs. Not sure when to stop.

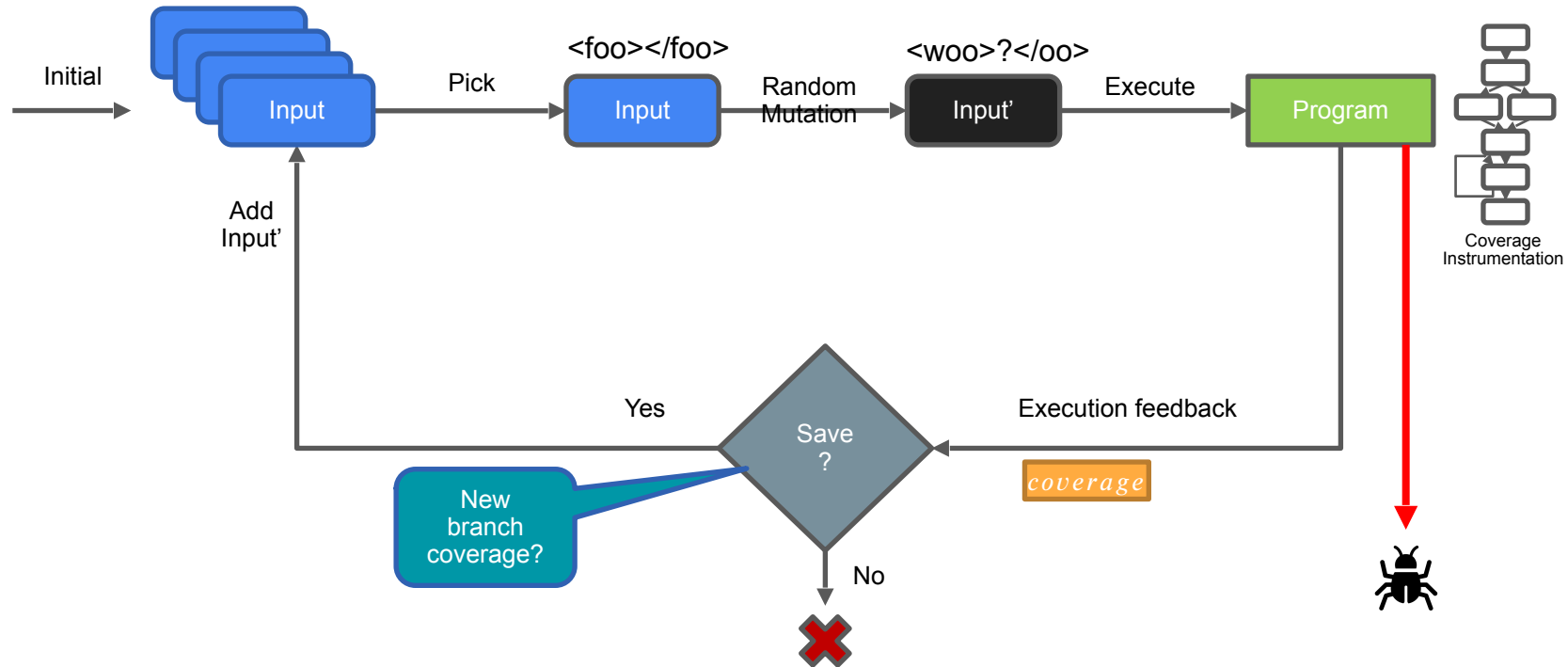
Mutation-Based Fuzzing (e.g. Radamsa)




Mutation Heuristics

- Binary input
 - Bit flips, byte flips
 - Change random bytes
 - Insert random byte chunks
 - Delete random byte chunks
 - Set randomly chosen byte chunks to *interesting* values e.g. INT_MAX, INT_MIN, 0, 1, -1, ...
- Text input
 - Insert random symbols relevant to format (e.g. "<" and ">" for xml)
 - Insert keywords from a dictionary (e.g. "<project>" for Maven POM.xml)
- GUI input
 - Change targets of clicks
 - Change type of clicks
 - Select different buttons
 - Change text to be entered in forms
 - ... Much harder to design

Coverage-Guided Fuzzing (e.g. AFL)




ClusterFuzz @ Chromium

 bugs

chromium ▾

New issue

All issues ▾



label:ClusterFuzz -status:Duplicate

1 - 100 of 25423

Next ▸

List

ID ▾	Pri ▾	M ▾	Stars ▾	ReleaseBlock ▾	Component ▾	Status ▾	Owner ▾
1133812	1	----	2	----	Blink>GetUserMedia Webcam	Untriaged	----
1133763	1	----	1	----	----	Untriaged	----
1133701	1	----	1	----	Blink>JavaScript	Untriaged	----
1133254	1	----	2	----	----	Untriaged	----
1133124	1	----	1	----	----	Untriaged	----
1133024	2	----	3	----	Internals>Network	Started	dmcardle@ch
1132958	1	----	2	----	UI>Accessibility, Blink>Accessibility	Assigned	sin...@chromi
1132907	2	----	2	----	Blink>JavaScript>GC	Assigned	dinfuehr@chr

Property-based testing

- Manually writing tests:

- - work

- - requires creativity

- - biased toward your expectations of where bugs are

- + precise (test relevant use cases)

- + can test basically anything

Can we generate lots of tests?

First, write down a property that a function should have, and a range:

```
@given(s.integers(min_value=-(10 ** 6), max_value=10 ** 6))

def test_factorize_multiplication_property(n):

    """The product of the integers returned by factorize(n) needs to be n."""

    factors = factorize(n)

    product = 1

    for factor in factors:

        product *= factor

    assert product == n, f"factorize({n}) returned {factors}"
```

Then, run Hypothesis, which searches the space...

```

===== test session starts =====
platform linux -- Python 3.8.4, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /home/moose/GitHub/MartinThoma/algorithms/medium/property-based-testing
plugins: hypothesis-5.23.8
collected 9 items

test_factorize_parametrize.py ..... [ 88%]
test_factorize_property.py F [100%]

===== FAILURES =====
_____ test_factorize_multiplication_property _____

    @given(s.integers(min_value=-(10 ** 6), max_value=10 ** 6))
> def test_factorize_multiplication_property(n):

test_factorize_property.py:10:
-----
n = 5

    @given(s.integers(min_value=-(10 ** 6), max_value=10 ** 6))
    def test_factorize_multiplication_property(n):
        """The product of the integers returned by factorize(n) needs to be n."""
        factors = factorize(n)
        product = 1
        for factor in factors:
            product *= factor
> assert product == n, f"factorize({n}) returned {factors}"
E   AssertionError: factorize(5) returned []
E   assert 1 == 5

test_factorize_property.py:16: AssertionError
----- Hypothesis -----
Falsifying example: test_factorize_multiplication_property(
    n=5,
)

===== short test summary info =====
FAILED test_factorize_property.py::test_factorize_multiplication_property - AssertionEr...
===== 1 failed, 8 passed in 0.12s =====

```

Oops! factorize(5)
returned an empty list of
factors!