Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 11 — February 9, 2026

Patrick Lam *version 1*

I write scientific content in LaTeX, which is somewhat user-unfriendly. If one posts in the relevant StackExchange with a problem, a Minimal Working Example (MWE) is pretty much required[1]. We need to reproduce a bug before we can fix it—hence a *working example*. To the point of this lecture, a *minimal* working example saves a lot of time for the person who is charged with fixing the bug.

The relevant *Fuzzing Book* content is at `https://www.fuzzingbook.org/html/Reducer.html`.

Specifically in our context of fuzzing: fuzzers produce potentially large inputs. Often, the created input contains more than is needed to reproduce a bug, and that makes it hard, as a human oracle, to understand what is going on.

In this lecture, we show how to *reduce* a failing input—that is, "to identify those circumstances of a failure that are relevant for the failure to occur, and to *omit* (if possible) those parts that are not". The *Fuzzing Book* quotes Kernighan and Pike in *The Practice of Programming*:

> For every circumstance of the problem, check whether it is relevant for the problem to occur. If it is not, remove it from the problem report or the test case in question.

The *Fuzzing Book* provides an example, where they have obscured the problem, adding some mystery to our lives.

```
class MysteryRunner(Runner):
    def run(self, inp: str) -> Tuple[str, Outcome]:
        x = inp.find(chr(0o17 + 0o31))
        y = inp.find(chr(0o27 + 0o22))
        if x >= 0 and y >= 0 and x < y:
            return (inp, Runner.FAIL)
        else:
            return (inp, Runner.PASS)
```

This `Runner` fails on some inputs. At this point, we have a number of fuzzing techniques at our disposal, but we can use plain old `RandomFuzzer` to find the failure.

```
def fuzz_mystery_runner():
    mystery = MysteryRunner()
    random_fuzzer = RandomFuzzer()
    while True:
        inp = random_fuzzer.fuzz()
        result, outcome = mystery.run(inp)
        if outcome == mystery.FAIL:
            break
    print (result)
```

OK, so we do that. It works—or fails, actually.

[1] Discussion: `https://tex.meta.stackexchange.com/questions/6255/why-does-tex-require-such-elaborate-mwes`

```
$ python3 mystery_runner.py
(%*50  1)-&7,;49:4?%:43*(-.
```

But the cause of the failure is not exactly clear from this input.

Manual Input Reduction

Before we write some code to do it, let's see how we can reduce an input manually. Kernighan and Pike continue by suggesting a divide and conquer process:

> Proceed by binary search. Throw away half the input and see if the output is still wrong; if not, go back to the previous state and discard the other half of the input.

Does this work?

```
>>> from mystery_runner import *
>>> failing_input = "(%*50  1)-&7,;49:4?%:43*(-."
>>> mystery = MysteryRunner()
>>> mystery.run(failing_input)
('(%*50  1)-&7,;49:4?%:43*(-.', 'FAIL')
>>> half_length = len(failing_input) // 2 # integer division
>>> first_half = failing_input[:half_length]
>>> mystery.run(first_half)
('(%*50  1)-&7,', 'FAIL')
```

That's progress. We now have a string that's half as long as the original and still triggers the failure. Let's try the same trick again, on the first half.

```
>>> quarter_length = len(first_half) // 2
>>> first_quarter = first_half[quarter_length:]
>>> mystery.run(first_quarter)
(' 1)-&7,', 'PASS')
>>> second_quarter = first_half[:quarter_length]
>>> mystery.run(second_quarter)
(' 1)-&7,', 'PASS')
```

Halving doesn't quite work this time. We need both the first quarter and the second quarter to trigger the failure—looking at the code, it's looking for two characters, but the characters don't have to be in the same half.

Delta Debugging

There are other ways to do binary searches. What we tried above was directly searching for the offending part of the input, but that didn't work. *Delta debugging* is another way. The insight here is to try to *remove* smaller and smaller parts of the input, and see whether the input still triggers the failure. Contrast that to trying to run on smaller and smaller parts of the input. Intuitively, it's more likely that removing parts keeps the input still-broken.

Let's see an example of one step of delta debugging: we next *remove* quarters of our failing input. First, the first quarter.

```
>>> quarter_length=len(failing_input)//4
>>> input_without_first_quarter=failing_input[quarter_length:]
>>> mystery.run(input_without_first_quarter)
(' 1)-&7,;49:4?%:43*(-.', 'PASS')
```

Because we're looking for a failure, we can see that we have to keep the first quarter to get the failure. Similarly, we can try to remove the second quarter.

```
>>> input_without_second_quarter=failing_input[:quarter_length]+failing_input[
                                                quarter_length*2:]
>>> mystery.run(input_without_second_quarter)
('(%*50 ,;49:4?%:43*(-.', 'PASS')
```

Again, removing the second quarter doesn't trigger the failure. From earlier, we would expect that we can remove the third and fourth quarters, so let's do that, in keeping with running an algorithm.

```
>>> input_without_third_quarter=failing_input[:quarter_length*2]+failing_input[
                                                quarter_length*3:]
>>> mystery.run(input_without_third_quarter)
('(%*50  1)-&7?%:43*(-.', 'FAIL')
```

Indeed, we can remove the third quarter. What about the fourth quarter?

```
>>> input_without_fourth_quarter=failing_input[:quarter_length*3]
>>> mystery.run(input_without_fourth_quarter)
('(%*50  1)-&7,;49:4', 'FAIL')
```

At some level, we're no further ahead yet than before. But the approach is different: there is a clear next step, which is to remove eighths from the first failing input we encountered.

The actual algorithm isn't quite like that, but it's close. The *Fuzzing Book* includes a `Reducer` base class.

```
class Reducer:
    """Base class for reducers."""

    def __init__(self, runner: Runner, log_test: bool = False) -> None:
        """Attach reducer to the given `runner`"""
        self.runner = runner
        self.log_test = log_test
        self.reset()
```

and an abstract `reduce` implementation. Also `test`.

There is also a `CachingReducer` which remembers what has been previously tested.

```
class CachingReducer(Reducer):
    def test(self, inp):
        if inp in self.cache:
            return self.cache[inp]

        outcome = super().test(inp)
        self.cache[inp] = outcome
        return outcome
```

The crux is the `DeltaDebuggingReducer`:

```
class DeltaDebuggingReducer(CachingReducer):
    """Reduce inputs using delta debugging."""

    def reduce(self, inp: str) -> str:
        """Reduce input `inp` using delta debugging. Return reduced input."""

        self.reset()
        assert self.test(inp) != Runner.PASS

        n = 2     # Initial granularity
        while len(inp) >= 2:
            start = 0.0
            subset_length = len(inp) / n
            some_complement_is_failing = False

            while start < len(inp):
                complement = inp[:int(start)] + \
                    inp[int(start + subset_length):]

                if self.test(complement) == Runner.FAIL:
                    inp = complement
                    n = max(n - 1, 2)
                    some_complement_is_failing = True
                    break

                start += subset_length

            if not some_complement_is_failing:
                if n == len(inp):
                    break
                n = min(n * 2, len(inp))

        return inp
```

It's not actually halving the size every time—it removes a chunk of size $1/n$, doubling n after running through all the chunks, but it decreases n by 1 when there is a test failure.

One can run the delta debugger:

```
dd_reducer = DeltaDebuggingReducer(mystery, log_test=True)
dd_reducer.reduce(failing_input)
```

and there is an example run in the *Fuzzing Book*, which I'll show excerpts from:

```
Test #1 ' 7:,>((/$$-/->.;.=;(.%!:50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(
Test #2 '\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(3%&5\'\'>#' 49 PASS
Test #3 " 7:,>((/$$-/->.;.=;(.%!:50#7*8=$&&=$9!%6(4=&69':" 48 PASS
Test #4 '50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(3%&5\'\'>#' 73 FAIL
Test #5 "50#7*8=$&&=$9!%6(4=&69':<7+1<2!4$>92+$1<(3%&5''>#" 49 PASS
Test #6 '50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+' 48 FAIL
...
Test #23 '(460)' 5 FAIL
Test #24 '460)' 4 PASS
```

```
Test #25 '(0)' 3 FAIL
Test #26 '0)' 2 PASS
Test #27 '(' 1 PASS
Test #28 '()' 2 FAIL
Test #29 ')' 1 PASS
'()'
```

I wouldn't want to do this manually on this test input. Since it's a random input it's harder to understand than a human-generated one, but, assuming that the system is deterministic, we can run the algorithm and get the answer. We also assume that test cases can run quickly enough that we can afford dozens of iterations. These are the same conditions as for fuzzing to work well.

In this case, the answer is that the system fails on an input with a `(` and then a `)`.

Delta debugging yields a 1-minimal test case: removing any character is guaranteed to not fail. In the example, we see that the single-paren cases pass. This is a local minimum: in principle, there might be some other smaller test case that one would reach with different choices, though there isn't in this case.

The *Fuzzing Book* points out the following advantages of reduced test cases:

- reduces cognitive load for the programmer: no irrelevant details, easier to understand what's happening.
- easier to communicate: we can say "`MysteryRunner` fails on "`()`"" rather than "`MysteryRunner` fails on 4100-character input (attached)" (or worse, not attached).
- helps identifying duplicates (to some extent—assuming that a failure has a single cause).

In terms of efficiency, delta debugging is best-case $O(\log n)$ and worst-case $O(n^2)$.

Note also that the `DeltaDebugging` implementation checks that the initial test case does fail.