Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 15 — March 2, 2026

Patrick Lam *version 1*

American Fuzzy Lop[1] is a mutation-based fuzzing tool widely used in practice. It's easy to use and finds real vulnerabilities. We'll talk about the basic principles behind it.

Mutation-based fuzzing

In mutation-based fuzzing (confusingly named, but not at all the same as mutation analysis), you use/develop a tool that randomly modifies existing inputs. You could do this totally randomly by flipping bytes in the input, or you could 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.

Here's a description of a mutation-based fuzzing workflow by the author of Fuzzinator[2].

> 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]

Generating random inputs: not very successful. Let's work through an example of fuzzing a URL parser[4].

So, let's first think about our domain—URLs. There is a definition of a valid URL[5]. A program that accepts URLs should do something useful with valid URLs and reject invalid URLs. Ideally, we would test some number of valid URLs and different kinds of invalid URLs.

A valid URL looks like this:

```
scheme://netloc/path?query#fragment
```

We are going to talk about the `scheme` part. There are fixed number of valid `scheme`s, including `http`, `https`, `file`, etc.

Let's use a library function to parse URLs.

[1] `http://lcamtuf.coredump.cx/afl/`

[2] `https://github.com/renatahodovan/fuzzinator`

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

[4] Much of today's lecture is based on `https://www.fuzzingbook.org/html/MutationFuzzer.html`

[5] More generally, RFC 3986 defines Uniform Resource Identifiers, or URIs: `https://datatracker.ietf.org/doc/html/rfc3986`

```
>>> from typing import Tuple, List, Callable, Set, Any
>>> from urllib.parse import urlparse

>>> urlparse("http://www.google.com/search?q=fuzzing")
ParseResult(scheme='http', netloc='www.google.com', path='/search',
    params='', query='q=fuzzing', fragment='')
```

And here's an example function that uses `urlparse`. How can we test this function?

```
def url_consumer(url: str) -> bool:
    supported_schemes = ["http", "https"]
    result = urlparse(url)
    if result.scheme not in supported_schemes:
        raise ValueError("Scheme must be one of " +
                         repr(supported_schemes))
    if result.netloc == '':
        raise ValueError("Host must be non-empty")

    # Do something with the URL
    return True
```

This function tries to parse its input and enforce the `scheme` being either `http` or `https`. If it's a valid URL with allowed scheme, it returns `True`. Otherwise it raises an error.

Let's see what happens if we run this 1000 times with random inputs, using the `fuzzer()` from last time to generate these inputs, rather than mutating existing inputs:

```
  for i in range(1000):
    try:
        url = fuzzer()
        result = url_consumer(url)
        print("Success!")
    except ValueError:
        pass
```

How likely is it that this is going to ever print success? The *Fuzzing Book* has a calculation, but basically, not very likely at all.

So, `fuzzer()` can find errors in the library function `urlparse()`, but it'll basically never get beyond that to test behaviour on any valid inputs. That is, if `url_consumer` were to actually do anything on the `True`-returning branch, we'd never be able to test that with inputs from `fuzzer()`.

Mutating inputs

If we do want a more dense set of valid inputs, there are basically two things we can do: mutate existing inputs, or use a grammar to generate inputs. (As a variant of generating with a grammar, one can also parse an input with the grammar, mutate the parse tree, and unparse.) We'll talk about mutating inputs, or mutational fuzzing.

Today's code is in the repo in the `L15/mutation_fuzzer.py` file.

When our input is a string, then we can insert a random character, delete a character, or change an existing character.

```python
def delete_random_character(s: str) -> str:
    """Returns s with a random character deleted"""
    if s == "":
        return s

    pos = random.randint(0, len(s) - 1)
    #print("Deleting", repr(s[pos]), "at", pos)
    return s[:pos] + s[pos + 1:]

def insert_random_character(s: str) -> str:
    """Returns s with a random character inserted"""
    pos = random.randint(0, len(s))
    random_character = chr(random.randrange(32, 127))
    #print("Inserting", repr(random_character), "at", pos)
    return s[:pos] + random_character + s[pos:]

def flip_random_character(s):
    """Returns s with a random bit flipped in a random position"""
    if s == "":
        return s

    pos = random.randint(0, len(s) - 1)
    c = s[pos]
    bit = 1 << random.randint(0, 6)
    new_c = chr(ord(c) ^ bit)
    #print("Flipping", bit, "in", repr(c) + ", giving", repr(new_c))
    return s[:pos] + new_c + s[pos + 1:]
```

We can run these functions:

```python
seed_input = "A quick brown fox"
for i in range(10):
    x = delete_random_character(seed_input)
    print(repr(x))

for i in range(10):
    print(repr(insert_random_character(seed_input)))

for i in range(10):
    print(repr(flip_random_character(seed_input)))
```

Or we can randomly choose one of the three functions to call:

```python
def mutate(s: str) -> str:
    """Return s with a random mutation applied"""
    mutators = [
        delete_random_character,
        insert_random_character,
        flip_random_character
    ]
    mutator = random.choice(mutators)
    # print(mutator)
    return mutator(s)
```

and call that function:

```
for i in range(10):
    print(repr(mutate("A quick brown fox")))
```

Back to URLs

In terms of its API, it's a bit inconvenient that our earlier `url_consumer()` function raises an error. Let's fit it into a function that returns `True` or `False`:

```
def is_valid_url(url: str) -> bool:
    try:
        result = url_consumer(url)
        return True
    except ValueError:
        return False

assert is_valid_url("http://www.google.com/search?q=fuzzing")
assert not is_valid_url("xyzzy")
```

We can now use our `mutate` function:

```
seed_input = "http://www.google.com/search?q=fuzzing"
valid_inputs = set()
trials = 20

for i in range(trials):
    inp = mutate(seed_input)
    if is_valid_url(inp):
        valid_inputs.add(inp)
```

and if you evaluate `len(valid_inputs)/trials`, you can see the proportion of your mutations that are valid inputs.

The *Fuzzing Book* talks about the probability of randomly mutating from `http` to `https` in an input, and works out that it's actually possible in reasonable time (3656 trials, 0.0049s in their example).

Multiple mutations. The setup for using multiple mutations in the book is somewhat contrived. For mutation analysis as discussed in Lecture 4, we only applied one mutation. But sometimes one does want to apply multiple mutations.

Let's see what happens when we apply multiple mutations.

```
seed_input = "http://www.google.com/search?q=fuzzing"
mutations = 50
inp = seed_input
for i in range(mutations):
    if i % 5 == 0:
        print(i, "mutations:", repr(inp))
    inp = mutate(inp)
```

After 45 mutations we see that we get something quite different from the original string:

```
45 mutations: " htP&)5q>-3ww.oo0lB_e/sca3ujdtzi'"
```

Implementation of a mutation fuzzer. In the `code/L15` directory, you'll find a `MutationFuzzer` class along with its dependencies. This class's constructor takes a seed and a minimum and maximum number of mutations to apply. Here is a base `Fuzzer` class, along with a `MutationFuzzer` class.

```
class MutationFuzzer(Fuzzer):
    """Base class for mutational fuzzing"""

    def __init__(self, seed: List[str],
                 min_mutations: int = 2,
                 max_mutations: int = 10) -> None
        # ...

    def reset(self) -> None:
        """Set population to initial seed.
        To be overloaded in subclasses."""
        self.population = self.seed
        self.seed_index = 0

    def create_candidate(self) -> str:
        """Create a new candidate by mutating a population member"""
        candidate = random.choice(self.population)
        trials = random.randint(self.min_mutations, self.
           max_mutations)
        for i in range(trials):
            candidate = self.mutate(candidate)
        return candidate

    def fuzz(self) -> str:
        if self.seed_index < len(self.seed):
            # Still seeding
            self.inp = self.seed[self.seed_index]
            self.seed_index += 1
        else:
            # Mutating
            self.inp = self.create_candidate()
        return self.inp
```

Basically, the important method here, `fuzz()`, returns the seeds the first few times it's called, and then calls `create_candidate` to obtain a randomlmy-chosen population member, mutated the appropriatet number of times. The population is currently populated with the seeds.

We can try it:

```
seed_input = "http://www.google.com/search?q=fuzzing"
mutation_fuzzer = MutationFuzzer(seed=[seed_input])
print(mutation_fuzzer.fuzz())
print(mutation_fuzzer.fuzz())
print(mutation_fuzzer.fuzz())
```

and we get the seed first and then its mutations, most of which aren't valid URLs—but more are valid than when we took randomly-generated strings.

Hierarchies

Before we started talking about mutation, we were generating pretty much purely-random inputs and fed them to the programs that we were testing. How effective is that going to be? We saw that most random inputs to `bc` didn't do anything interesting. Let's still use randomness, but generate inputs in a more directed way. I've alluded to this above, but we are going to say a bit more now.

Say that we're trying to generate C programs rather than URLs. They have a lot more structure! One could propose the following hierarchy of inputs[6]:

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 of these levels contains a subset of the inputs from previous levels. However, as the level increases, we are more likely to find interesting bugs that reveal functionality specific to the system (rather than simply input validation issues).

How do we generate inputs at higher levels of the hierarchy? There are two choices: grammars will get you up to some levels of the hierarchy, but then you need more smarts than you can encode in

[6] `http://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf`

a context-free grammar to generate type-correct programs; or, you can modify existing inputs, as we've seen above.

While the example above is specific to C, the concept applies to all generational fuzzing tools. Of course, the system under test shouldn't ever crash on random ASCII characters. But it's hard to find the really interesting cases without incorporating knowledge about correct syntax for inputs. Increasing the level should also increase code coverage.

John Regehr discusses this issue at greater length[7] and concludes that generational fuzzing tools should operate at all levels, rather than restricting themselves to only some of the levels.

Guiding by Coverage

Time to introduce a new idea—one that has been made popular in practice by AFL. We've previously talked about coverage in terms of evaluating how good a test suite is. Now, we're going to use coverage to guide test generation.

Let's build some infrastructure first. We have an abstract `Runner` class with a `run()` function. By default, the abstract class just says everything is "unresolved". Here is a subclass that runs a function it is given during instantiation, and returns `PASS` if the function returns sucessfully and `FAIL` if the function raises an exception.

```
class FunctionRunner(Runner):
    def __init__(self, function: Callable) -> None:
        """Initialize.  ‘function‘ is a function to be executed"""
        self.function = function

    def run_function(self, inp: str) -> Any:
        return self.function(inp)

    def run(self, inp: str) -> Tuple[Any, str]:
        try:
            result = self.run_function(inp)
            outcome = self.PASS
        except Exception:
            result = None
            outcome = self.FAIL

        return result, outcome
```

We can create and invoke this runner, with the original exception-raising function:

```
http_runner = FunctionRunner(url_consumer)
http_runner.run("https://foo.bar/")
```

Back in Lecture 3, we talked about measuring coverage programmatically. Now we can use that to guide fuzzing. We'll also use the notation of populations. First, measuring coverage. We can create

[7] `blog.regehr.org/archives/1039`

a FunctionCoverageRunner which subclasses FunctionRunner but defines this run_function() implementation:

```
class FunctionCoverageRunner(FunctionRunner):
    def run_function(self, inp: str) -> Any:
        with Coverage() as cov:
            try:
                result = super().run_function(inp)
            except Exception as exc:
                self._coverage = cov.coverage()
                raise exc

        self._coverage = cov.coverage()
        return result

    def coverage(self) -> Set[Location]:
        return self._coverage
```

Running it and calling getter function coverage() gives a list of program points, which are function/line tuples.

Now, the idea is to add an input to the population of source inputs whenever that input adds to coverage. When the runner next asks for an input, it will mutate something that has been added to the population in a previous iteration.

```
class MutationCoverageFuzzer(MutationFuzzer):
    """Fuzz with mutated inputs based on coverage"""

    def reset(self) -> None:
        super().reset()
        self.coverages_seen: Set[frozenset] = set()
        # Now empty; we fill this with seed in the first fuzz runs
        self.population = []

    def run(self, runner: FunctionCoverageRunner) -> Any:
        """Run function(inp) while tracking coverage.
           If we reach new coverage,
           add inp to population and its coverage to
              population_coverage
        """
        result, outcome = super().run(runner)
        new_coverage = frozenset(runner.coverage())
        if outcome == Runner.PASS and new_coverage not in self.
           coverages_seen:
            # We have new coverage
            self.population.append(self.inp)
            self.coverages_seen.add(new_coverage)

        return result
```

After running this fuzzer for a number of trials, we can look at the population and see how it consists of a number of valid inputs.

```
['http://www.google.com/search?q=fuzzing', 'http://wwwgo\x7fgie.c*om/{earchq=fuzzing',
'http://\\www=go\x7fgie.c*nm{erchq=fuzzig', 'http://www.google.com/ear#h?q=fuzzing',
'http://\\www=g/\x7fgienc*nKmer;chq=nuzzig', 'http://wwv.goog?le.com/?ear#h?q9fuzzing',
'http://\\www=g/\x7fgienC*nKmer;#hq=nuzzig', 'http://\\www=g/\x7fgiec*n(ier;c/hq=nuZrig.',
'http://\\wwiw=ag/\x7fgienC*nKmer;iq=nuzzi?g']
```

We need additional machinery to see how coverage over time increases using this strategy, and there's a plot of that in the *Fuzzing Book*, but maybe you can take my word for it.

In any case, coverage-guided fuzzing definitely explores new parts of the program's behaviour as it runs. Of course, as with any type of fuzzing, it will eventually hit diminishing returns.