Software Testing, Quality Assurance & Maintenance—Lecture 8

Patrick Lam
University of Waterloo

January 30, 2026

Part I

Mutation-Based Fuzzing

Putting things together

Goal: generate many test cases automatically.

When we talked about helping human oracles, we mentioned starting from known inputs.

Mutation-based fuzzing: automatically modify known inputs.

Mutation-based fuzzing in practice

Could just flip bytes in the input.

Or, parse the input and change some nonterminals in the AST.

Note: Also need to update checksums to see anything interesting.

Example: URLs

A valid URL looks like this:

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

There is a definition of valid vs invalid URLs (RFC 3986).

A program should do something useful with valid URLs and reject invalid URLs.

Let's use fuzzing to generate valid and invalid URLs.

schemes

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

There are a fixed number of valid `scheme`s:
`http`, `https`, `file`, etc.

Using the urllib library

```
>>> from typing import Tuple, List
>>> from typing import 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='')
```

urllib in ur 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
```

How to test?

Naive input generation

In `code/L08/random_inputs.py`:

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

You'd be very lucky indeed to see Success!.

Basically, this fuzzing won't test anything past validation.

Being less naive

Basically two alternatives:

- mutate existing inputs; or,
- generate inputs using a grammar.

(As mentioned earlier, can also parse/mutate/unparse).

Mutating existing inputs (strings)

```
import random

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:]
```

Mutating existing inputs (strings)

```
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:]
```

Running the mutation code

```
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)))
```

Choose randomness randomly

```
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)

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

Back to URLs: retrofitting `url_consumer`

```
from random_inputs import url_consumer

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")
```

Easier to test with this wrapper.

Using the mutation fuzzer

```
from mutation_fuzzer import MutationFuzzer

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

mutation_fuzzer = MutationFuzzer([])
for i in range(trials):
    inp = mutation_fuzzer.mutate(seed_input)
    if is_valid_url(inp):
        valid_inputs.add(inp)

print (len(valid_inputs)/trials)
```

What do you observe when you run this?

Exercise: `http` → `https`

How long should you expect to wait before randomly mutating `http` to `https` and getting a valid input?

Multiple mutations

Not for mutation analysis, but useful here.

```
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 = mutation_fuzzer.mutate(inp)
```

Encapsulating fuzzing in a 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:
        # ...
```

Useful functions

```
    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
```

Using MutationFuzzer

```
>>> 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())
http://www.google.com/search?q=fuzzing
http+:R/'ww.google.com/serchql=fuzing
htEtp://wwwgoogld.coi/earch?qn=fung
```

Part II

Intermission: Guiding by Coverage

On Hierarchies

We know that randomly changing bytes won't exercise much interesting functionality.

It can cause crashes, though, at least for a while.

Let's continue to use randomness, but in a more directed way.

Hierarchy of inputs: C

C programs are way more structured than URLs.

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.

Generating higher-level inputs

Two choices:

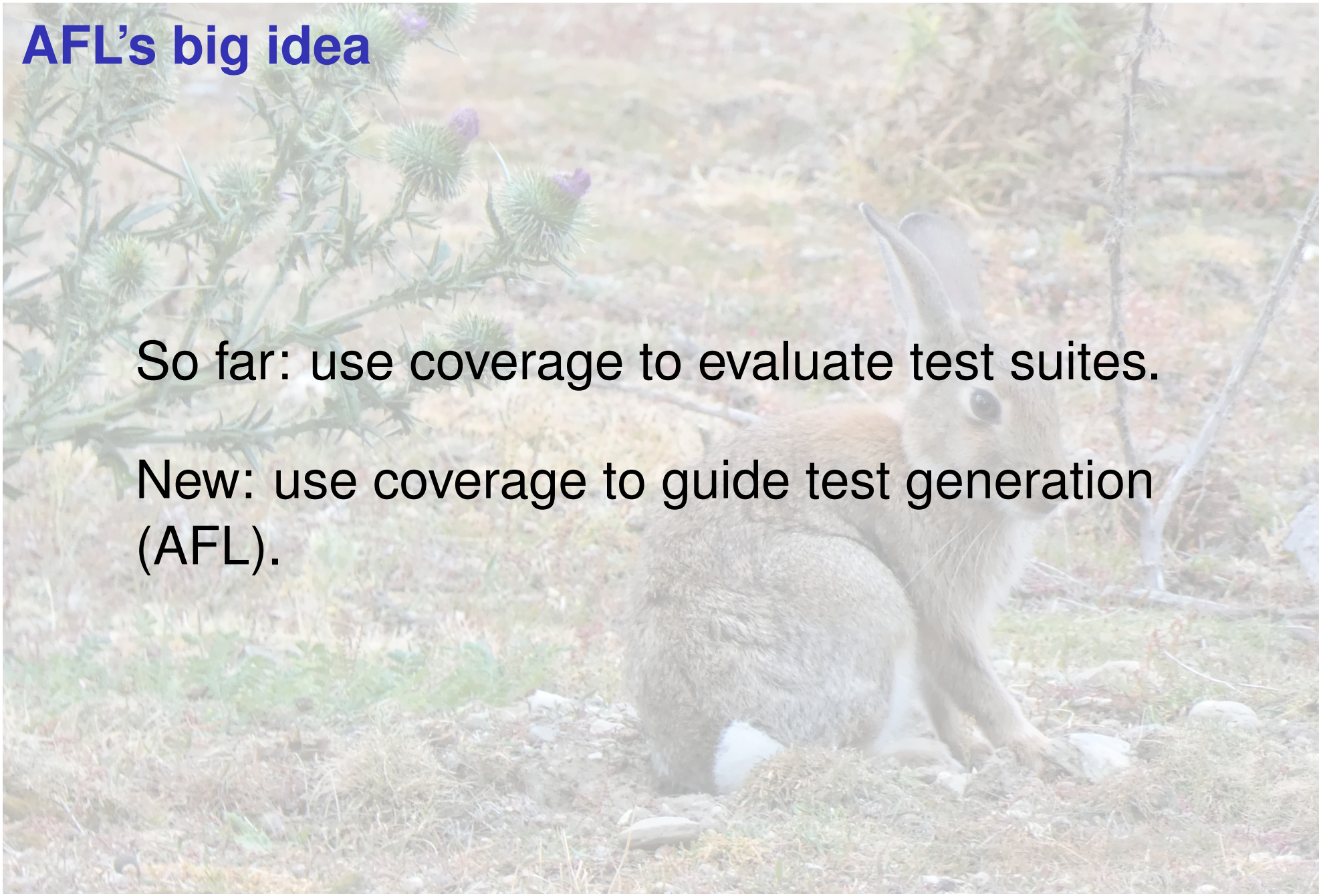
1. use grammars (context-free grammars still don't satisfy all constraints)
2. modify existing inputs (as seen above)

This is true for all generational fuzzing tools. Need to incorporate knowledge about correct syntax.

Part III

Guiding by Coverage

AFL's big idea



So far: use coverage to evaluate test suites.

New: use coverage to guide test generation (AFL).

Infrastructure

```
class Runner:
    """Base class for testing inputs."""

    # Test outcomes
    PASS = "PASS"
    FAIL = "FAIL"
    UNRESOLVED = "UNRESOLVED"

    def __init__(self) -> None:
        """Initialize"""
        pass

    def run(self, inp: str) -> Any:
        """Run the runner with the given input"""
        return (inp, Runner.UNRESOLVED)
```

Instantiating infrastructure

```
class FunctionRunner(Runner):
    def __init__(self, function: Callable) -> None:
        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
```

Running the FunctionRunner

```
from fuzzer import Runner
from random_inputs import url_consumer
from urllib.parse import urlparse

if __name__ == "__main__":
    # view output from urlconsumer_runner:
    urlconsumer_runner = FunctionRunner(url_consumer)
    print (urlconsumer_runner.run("https://foo.bar"))
```

Output: `(True, 'PASS')`

Measuring Coverage in the Runner

```
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 `function_coverage_runner.py`

```
if __name__ == "__main__":
    from urllib.parse import urlparse

    # view output from urlconsumer_runner:
    urlconsumer_runner = FunctionCoverageRunner(
                                          url_consumer)
    urlconsumer_runner.run("https://foo.bar")

    print(list(urlconsumer_runner.coverage())[:5])
```

prints a slice of the coverage:

```
[('url_consumer', 7), ('_splitnetloc', 416),
 ('_splitnetloc', 419), ('urlsplit', 502),
 ('urlsplit', 499)]
```

Putting the AFL Idea into Practice

Maintain a population of source inputs.

Add an input to the population when the input adds to coverage.

The mutation fuzzer (from Part I) mutates inputs in the population to generate new candidate inputs.

MutationCoverageFuzzer implementation

```
class MutationCoverageFuzzer(MutationFuzzer):
    def reset(self) -> None:
        super().reset()
        self.coverages_seen: Set[frozenset] = set()
        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
```

The population

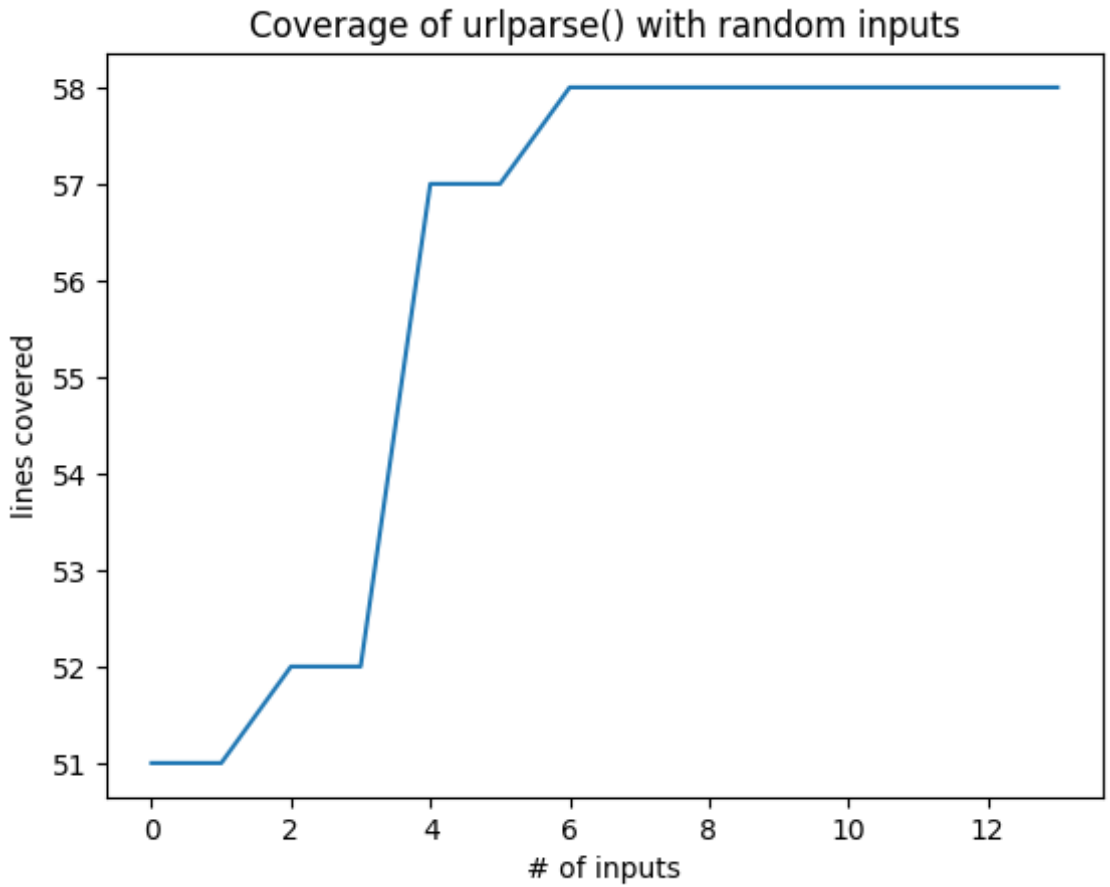
```
if __name__ == "__main__":
    seed_input = "http://www.google.com/search?q=fuzzing"
    mutation_coverage_fuzzer = MutationCoverageFuzzer(seed=[
                                        seed_input])
    urlconsumer_runner = FunctionCoverageRunner(url_consumer)
    mutation__coverage_fuzzer.runs(urlconsumer_runner, trials=
                                        10000)
    print (mutation_fuzzer.population)
```

We aim to increase coverage of `url_consumer` and functions it calls. The population after 10,000 trials:

```
['http://www.google.com/search?q=fuzzing',
 'http://www.google|.com/search\x7fq=fuZzing',
 'http://ww;w.google|.com/searc\x7f=fuZzing#',
 'http://ww;w?.gogle|com/sEarc\x7f=f,uZzig#',
 "http://www.googla|.com'sarch\x7fq9fuZzi!ng",
 'http://ww;wgoole|/com/sear;c\x7ffuZzing#',
 'http://wg;wgoole|m/cnmb/suar;cwfuZzing\x03',
 'http://wg;wgoole|m/cnmb/suar;cwfuZzing\x03',
 'http://wgW;wgoole|m/cnmb/s}ar;cwfuZz-ing\x03/:',
 'http://wgW;wgoole|m/cnmb/s}ar;cwfuZz?-qing\x03/:',
 'Http://wg5W;\x7fgoorle|amcmb/S}ar;cwfuZz?-qing#\x03/:',
 'Http://wg5W;\x7fgoOrle|!mcmb/S}ap;cwfuZj/-qing#/:']
```

Coverage increases

It is possible to plot coverage-over-time using this strategy; see the *Fuzzing Book* for details, but here's a picture from there.



Code comprehension exercise

There's a lot of inheritance in `MutationCoverageFuzzer`.

Exercise: How does `MutationCoverageFuzzer.runs()` work? Trace the execution and form an understanding of how the classes fit together.

Coverage-guided fuzzing summary

Coverage-guided fuzzing (AFL) definitely explores new parts of the program's behaviour as it runs.

Eventually, it hits diminishing returns.