Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 8 — January 30, 2026

Patrick Lam *version 2*

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 some of the principles behind it, though this lecture misses some important afl components (maybe later).

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. The Fuzzinator experience report we mentioned last time was mutation-based fuzzing. 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.

Generating random inputs: not very successful. Let's work through an example of fuzzing a URL parser[2].

So, let's first think about our domain—URLs. There is a definition of a valid URL[3]. 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.

```
>>> 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?

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

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

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

```python
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 (in `code/L08/random_inputs.py`):

```python
  for i in range(1000):
    try:
        fuzzer = Fuzzer()
        url = fuzzer.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 `L08/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

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)
```

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

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/L08` 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 randomly-chosen population member, mutated the appropriate 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[4]:

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 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.

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

John Regehr discusses this issue at greater length[5] and concludes that generational fuzzing tools should operate at all levels, rather than restricting themselves to only some of the levels.

Guiding by Coverage: basic idea

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. Using coverage to guide test generation is known as *greybox* fuzzing: it uses some information about the system, but less than whitebox fuzzing.

There are more technical details about AFL here: `https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt`.

Let's build some infrastructure first. We introduce an abstract `Runner` class with a `run()` function (`code/L08/fuzzer.py`). 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)
print(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 a `FunctionCoverageRunner` which subclasses `FunctionRunner` but defines this `run_function()` implementation:

```
class FunctionCoverageRunner(FunctionRunner):
```

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

```
    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. The mutation fuzzer mutates inputs in the population to generate new candidate inputs.

In Python, we can just measure coverage using built-in language features. From what I understand, the preferred mode of operation for AFL is to take the assembly code generated by the compiler and to add instrumentation to record coverage information—in particular, branch counts. AFL can also collect coverage information from a virtual machine (QEMU) or dynamic instrumentation (Pintools.

```
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.

Exercise. How does `MutationCoverageFuzzer.runs()` work?

There is a lot of inheritance in this collection of classes. You should be able to inspect the code and understand it (CS247 skills), but it's a good exercise to do so in any case, both for really understanding this concept, and in terms of code comprehension skills.

The API to `MutationCoverageFuzzer` (MCF) is the `runs()` method; you tell it to do N runs. The inherited implementation, from `Fuzzer`, calls `self.run()` N times.

Then, MCF.`run()` does one superclass call to `MutationFuzzer.run()` to run on some input i that the `MutationFuzzer` generates. If i leads to new coverage (not in `.coverages_seen`), MCF adds i to the `.population`.

The `MutationFuzzer` uses the `.population` to generate new inputs by fuzzing something drawn randomly from there. But, until the seeds are used up, it starts by returning the seeds as the first inputs—on they way, the seeds would get added to the `.population` if they add to coverage.

Example: Guiding by Coverage for the win. Here is some code that is designed to be resistant to normal fuzzing but susceptible to guided fuzzing, found in `code/L08/crashme.py`:

```
def crashme(s: str) -> None:
    if len(s) > 0 and s[0] == 'b':
        if len(s) > 1 and s[1] == 'a':
            if len(s) > 2 and s[2] == 'd':
                if len(s) > 3 and s[3] == '!':
                    raise Exception()
```

If we run it with input `''good''` and ask for coverage from a `FunctionCoverageRunner`, we'll see something like

```
[('crashme', 4), ('run_function', 13)]
```

Let's take a step back and measure how well a (blackbox) mutation-based fuzzer works on `crashme`. The *Fuzzing Book* includes an `AdvancedMutationFuzzer`, found in `code/L08/advanced_mutation_fuzzer.py`. I'm not going to discuss it, because it's almost exactly the same as the `MutationFuzzer` above, except that it does between 1 and 5 mutations when it draws a seed from the population to return a candidate. The `AdvancedMutationFuzzer` is blackbox because it does not, in particular, consider coverage.

```python
import time

n = 30000
seed_input = "good"

blackbox_fuzzer = AdvancedMutationFuzzer([seed_input], Mutator(),
                                          PowerSchedule())

start = time.time()
blackbox_fuzzer.runs(FunctionCoverageRunner(crashme), trials=n)
end = time.time()

print ("It took the blackbox mutation-based fuzzer %0.2f seconds to generate
                                          and execute %d inputs." % (end -
                                          start, n))

_, blackbox_coverage = population_coverage(blackbox_fuzzer.inputs, crashme)
bb_max_coverage = max(blackbox_coverage)

print ("The blackbox mutation-based fuzzer achieved a maximum coverage of %d
                                          statements." % bb_max_coverage)

print ([seed_input] + \
[\
    blackbox_fuzzer.inputs[idx] for idx in range(len(blackbox_coverage))\
    if blackbox_coverage[idx] > blackbox_coverage[idx - 1]\
])
```

I ran this and got the output:

```
It took the blackbox mutation-based fuzzer 0.57 seconds to generate and execute 30000 inputs.
The blackbox mutation-based fuzzer achieved a maximum coverage of 2 statements.
['good', 'boo']
```

On this run, the fuzzer never gets past the second `if` statement. (I re-ran it a couple of times and it got past the second statement once.)

There is also a `GreyboxFuzzer` which extends `AdvancedMutationFuzzer`, but is almost entirely like the `MutationCoverageFuzzer` above. In file `code/L08/compare_blackbox_greybox.py`, we add lines to also run `GreyboxFuzzer`, and observe this output:

```
It took the blackbox mutation-based fuzzer 0.65 seconds to generate and execute 30000 inputs.
The blackbox mutation-based fuzzer achieved a maximum coverage of 2 statements.
['good', 'bgodI']
It took the greybox mutation-based fuzzer 0.73 seconds to generate and execute 30000 inputs.
Our greybox mutation-based fuzzer covers 3 more statements
[good, bgod, bao]Cd, badS, bad!]
```

That is, adding coverage-based feedback allows the fuzzer to reach the otherwise-unlikely nested branches.

The *Fuzzing Book* also shows graphs of coverage over time for the black-box and grey-box fuzzer.

Summary. 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.

Guiding by Coverage: AFL's refinements—Power Schedules

To date, we've just put paths that increase coverage into the population, and drawn seeds from the population uniformly at random. We can do better: some paths are more important than others, and we want the important paths to come up more often when generating new paths[6].

To improve this, we introduce the concept of a *power schedule*. The power schedule assigns an energy value (floating-point) to each seed in the population, allowing the fuzzer to prioritize higher-energy and thus presumably higher-value seeds, and can randomly select a seed from the population consistent with the energy distribution.

There is a `Seed` class in `code/L08/power_schedule.py`, but the only important thing for our purposes is its `energy` and `data` properties. The power schedule is more interesting:

```python
class PowerSchedule:
    """Define how fuzzing time should be distributed across the population."""

    def __init__(self) -> None:
        """Constructor"""
        self.path_frequency: Dict = {}

    def assignEnergy(self, population: Sequence[Seed]) -> None:
        """Assigns each seed the same energy"""
        for seed in population:
            seed.energy = 1

    def normalizedEnergy(self, population: Sequence[Seed]) -> List[float]:
        """Normalize energy"""
        energy = list(map(lambda seed: seed.energy, population))
        sum_energy = sum(energy)  # Add up all values in energy
        assert sum_energy != 0
        norm_energy = list(map(lambda nrg: nrg / sum_energy, energy))
        return norm_energy

    def choose(self, population: Sequence[Seed]) -> Seed:
        """Choose weighted by normalized energy."""
        self.assignEnergy(population)
        norm_energy = self.normalizedEnergy(population)
        seed: Seed = random.choices(population, weights=norm_energy)[0]
        return seed
```

both of which can be found in `code/L08/power_schedule.py`. We can check to see that the default implementation chooses more or less uniformly:

```python
    population = [Seed("A"), Seed("B"), Seed("C")]
    schedule = PowerSchedule()
    hits = { "A": 0, "B": 0, "C": 0 }
    for i in range(10000):
        seed = schedule.choose(population)
```

[6] *Fuzzing Book* reference: `https://www.fuzzingbook.org/html/GreyboxFuzzer.html`

```
        hits[seed.data] += 1
    print (repr(hits))
```

results in:

```
{'A': 3372, 'B': 3249, 'C': 3379}
```

It turns out that there is one more difference in the `AdvancedMutationFuzzer`: it chooses asks the power schedule to choose the seed, presumably using probabilities weighted according to the power schedule. So far, we haven't assigned any energy, so all seeds have energy 1.

But now, let's give unusual paths—those not exercised very often—more energy. We'll put this code in `code/L08/counting_grey_box_fuzzer.py`. First, path IDs.

```
import pickle   # serializes an object by producing a byte array from all the
                                                   information in the object
import hashlib  # produces a 128-bit hash value from a byte array

def getPathID(coverage: Any) -> str:
    """Returns a unique hash for the covered statements"""
    pickled = pickle.dumps(sorted(coverage))
    return hashlib.md5(pickled).hexdigest()
```

The original AFL assigns an energy that is constant in the number of times a seed has been chosen $s(i)$; AFLFast assigns an energy exponential in $s(i)$. The AFLFast paper [BPR19] writes:

> When the seed is fuzzed for the first time, very low energy is assigned. Every time the seed is chosen thereafter, exponentially more inputs are generated up to a certain bound. This allows to rapidly approach the minimum energy required to discover a new path.

Hence:

```
class AFLFastSchedule(PowerSchedule):
    """Exponential power schedule as implemented in AFLFast"""

    def __init__(self, exponent: float) -> None:
        self.exponent = exponent

    def assignEnergy(self, population: Sequence[Seed]) -> None:
        """Assign exponential energy inversely proportional to path frequency"""
        for seed in population:
            seed.energy = 1 / (self.path_frequency[getPathID(seed.coverage)] **
                                                   self.exponent)
```

so we can see that this power schedule assigns energy inversely proportional to frequency, exponentiated. The counting greybox fuzzer adds path frequency to the paths that are run.

```
    def run(self, runner: FunctionCoverageRunner) -> Tuple[Any, str]:
        """Inform scheduler about path frequency"""
        result, outcome = super().run(runner)

        path_id = getPathID(runner.coverage())
        if path_id not in self.schedule.path_frequency:
            self.schedule.path_frequency[path_id] = 1
        else:
            self.schedule.path_frequency[path_id] += 1

        return(result, outcome)
```

We can compare this counting greybox fuzzer to the original. Nothing special here:

```
    import time
    n = 10000
    seed_input = "good"
    fast_schedule = AFLFastSchedule(5)
    fast_fuzzer = CountingGreyboxFuzzer([seed_input], Mutator(), fast_schedule)
    start = time.time()
    fast_fuzzer.runs(FunctionCoverageRunner(crashme), trials=n)
    end = time.time()

    print ("It took the fuzzer w/ exponential schedule %0.2f seconds to generate
                                          and execute %d inputs." % (end -
                                          start, n))
    _, counting_greybox_coverage = population_coverage(fast_fuzzer.inputs, crashme
                                          )
    cgb_max_coverage = max(counting_greybox_coverage)
    print ("Our fuzzer w/exponential schedule covers %d statements." % (
                                          cgb_max_coverage))
    print("             path id 'p'           : path frequency 'f(p)'")
    print (fast_schedule.path_frequency)

    seed_input = "good"
    orig_schedule = PowerSchedule()
    orig_fuzzer = CountingGreyboxFuzzer([seed_input], Mutator(), orig_schedule)
    start = time.time()
    orig_fuzzer.runs(FunctionCoverageRunner(crashme), trials=n)
    end = time.time()

    print ("It took the fuzzer w/ original schedule %0.2f seconds to generate and
                                          execute %d inputs." % (end - start, n
                                          ))
    print("             path id 'p'           : path frequency 'f(p)'")
    print (orig_schedule.path_frequency)
```

Running this a few times, I observed that the exponential schedule is often a bit slower than the original schedule, but it also more consistently hits 5 statements/paths, with a decent number of hits to the fifth statement. When the original schedule hits the fifth statement, the count is sometimes as low as 4.

```
It took the fuzzer w/ exponential schedule 0.46 seconds to generate and execute 10000 inputs.
Our fuzzer w/exponential schedule covers 5 statements.
             path id 'p'           : path frequency 'f(p)'
{'26b4becfdd3a8aacb81607a627bd1854': 5468, 'bc2fa870f15bb877d04c81e7bef87bac': 2694,
 '6f2492ce0367e22be7f8327c8e333853': 1119, 'f7b00fe99a9c688bbd30df9e747557a8': 452,
 '8669eeece2269c362bfa8d147565bbb5': 267}
It took the fuzzer w/ original schedule 0.30 seconds to generate and execute 10000 inputs.
             path id 'p'           : path frequency 'f(p)'
{'26b4becfdd3a8aacb81607a627bd1854': 7538, 'bc2fa870f15bb877d04c81e7bef87bac': 2121,
 '6f2492ce0367e22be7f8327c8e333853': 327, 'f7b00fe99a9c688bbd30df9e747557a8': 14}
```

We can also examine the normalized energy assigned to the paths under both schedules (see code):

```
fast schedule:
'26b4becfdd3a8aacb81607a627bd1854', 0.00000, 'good'
'bc2fa870f15bb877d04c81e7bef87bac', 0.00000, 'boodVP'
'6f2492ce0367e22be7f8327c8e333853', 0.00003, 'baooDvP'
'f7b00fe99a9c688bbd30df9e747557a8', 0.03650, 'badvP'
'8669eeece2269c362bfa8d147565bbb5', 0.96347, 'bad!t8D'
original schedule:
'26b4becfdd3a8aacb81607a627bd1854', 0.25000, 'good'
'bc2fa870f15bb877d04c81e7bef87bac', 0.25000, 'bgoodQ'
'6f2492ce0367e22be7f8327c8e333853', 0.25000, 'baJ g6poodP'
'f7b00fe99a9c688bbd30df9e747557a8', 0.25000, 'badN g6poodP'
```

We see that the unusual path gets the vast majority of the energy under the fast schedule, while all paths have the same energy under the original schedule.

Another Example: HTMLParser. We can also evaluate different fuzzers on the Python library's HTML parser:

```python
from html.parser import HTMLParser

def my_parser(inp:str) -> None:
    parser = HTMLParser()
    parser.feed(inp)
```

Starting with $n = 5000$ and a single seed input containing one space (see `code/L08/fuzz_htmlparser.py`):

```
It took all three fuzzers 14.77 seconds to generate and execute 5000 inputs.
Maximum coverages: 65, 165, 168.
Last 10 blackbox:
[' 0', '\x00', '', '', ' /', ' +', '', 'X ', '', '\x00']
Last 10 greybox:
['', '6uJ(', '&6D+G<\x1b!G(', '1&x<$<<n>', '~\x0ek\n#\\<', 'z<<`', '="8!\x01', '1:&9<[8)<?!x',
 '\x15L:a&$T<', '<|']
Last 10 counting greybox:
['>W//<', 'W!<E-/~><?<V', ']\nCNL\x0eD>j<v', '\x1cZ./8\x1f5?$YIN)', 'N$<><Y1!Ie',
 "z|i\x0c6'}><", 'N,\x1c/?5><!I', '%V^Mo5&n<>+.j<', '>N\rP5!G>', 'PXRCnb+/']
```

We can see that the counting greybox fuzzer does a bit better than the greybox fuzzer in terms of coverage, which does a lot better than the blackbox fuzzer. We can also see that the blackbox fuzzer doesn't find much, while the greybox fuzzer starts to include things like brackets. The counting greybox fuzzer includes longer inputs.

Still, even the counting greybox fuzzer doesn't have keywords like `<html>`. We'll need to involve grammars to do that.

References

[BPR19] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.