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: generate new inputs automatically, by modifying 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, defined in the RFC: `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: higher-level mutation).

Mutating existing inputs (strings)

```python
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)

```python
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

```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)))
```

Choose randomness randomly

```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)

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

(code/L08/mutator.py)

Back to URLs: retrofitting `url_consumer`

```python
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:
wrapper returns `True`/`False`.

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` $\rightarrow$ `https`

How many trials to expect before randomly mutating `http` to get `https`?

Applying multiple mutations to one input

Not for mutation analysis, but critical for mutation fuzzing.

```
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

```python
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: Hierarchies

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: greybox fuzzing = use coverage to guide test generation (used in AFL, with some more twists).

How AFL gathers coverage information

In Python: use language features to measure coverage.

AFL: rewrite assembly code, adding instrumentation to collect branch counts.

AFL can also collect coverage information from a virtual machine (QEMU) or dynamic instrumentation (pintools).

Infrastructure

```python
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

```python
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

```python
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: Greybox Fuzzing

Maintain a population of source inputs.

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

Greybox Fuzzing: Add an input to the population when that input adds to coverage.

Why greybox?

Blackbox: don’t look at the implementation at all.

Whitebox: use the implementation to guide testing.

Greybox: use coverage to guide testing, but don’t look at the implementation itself.

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:
            self.population.append(self.inp)
            self.coverages_seen.add(new_coverage)

        return result
```

The population

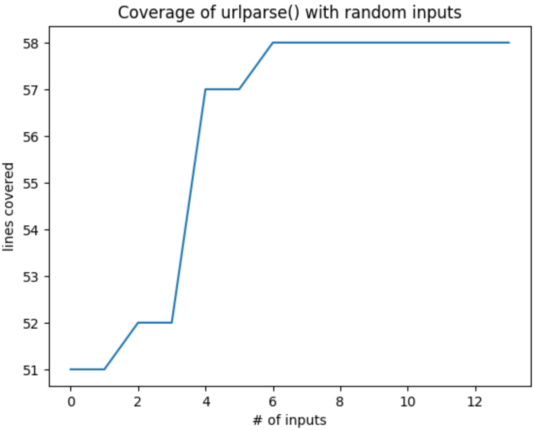
```
if __name__ == "__main__":
    seed_input = "http://www.google.com/search?q=fuzzing"
    mutation_coverage_fuzzer = MutationCoverageFuzzer(seed=[
                                                seed_input])
    urlconsumer_runner = FunctionCoverageRunner(url_consumer)
    mutation__coverage_fuzzer.runs(urlconsumer_runner,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

We aim to increase coverage of `url_consumer` and the 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.

Guiding by Coverage for the win

Consider this code to be tested:

```
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()
```

Resistant to normal mutation fuzzing.

Trying Mutation Fuzzing

```
n=30000
seed_input="good"
bb_fuzzer=AdvancedMutationFuzzer([seed_input], Mutator(),
                                  PowerSchedule())

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

print ("Blackbox mutation-based: %0.2fs for %d inputs." %
                                  (end - start, n))
_, blackbox_coverage = population_coverage(
                                 blackbox_fuzzer.inputs,
                                 crashme)
bb_max_coverage = max(blackbox_coverage)
print ("Blackbox mutation-based: max coverage %d." %
                                 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]\
])
```

Mutation Fuzzing Results

```
Blackbox mutation-based: 0.57s for 30000 inputs.
Blackbox mutation-based fuzzer: max coverage 2.
['good', 'boo']
```

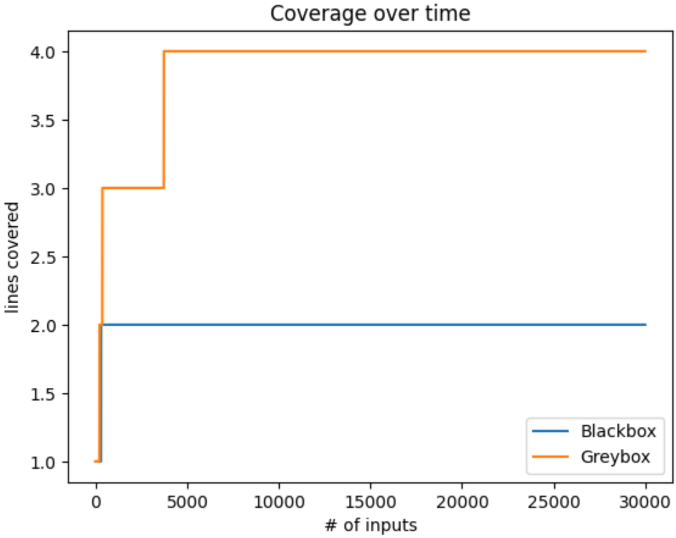
Adding Greybox: Results

You'll find `GreyboxFuzzer` in the repo also.

```
Blackbox mutation-based: 0.65s for 30000 inputs.
Blackbox mutation-based fuzzer: max coverage 2.
['good', 'bgodI']
Greybox mutation-based: 0.73s for 30000 inputs.
Greybox mutation-based fuzzer: 3 more.
[good, bgod, bao]Cd, badS, bad!]
```

That's promising: it reaches otherwise-unlikely nested branches.

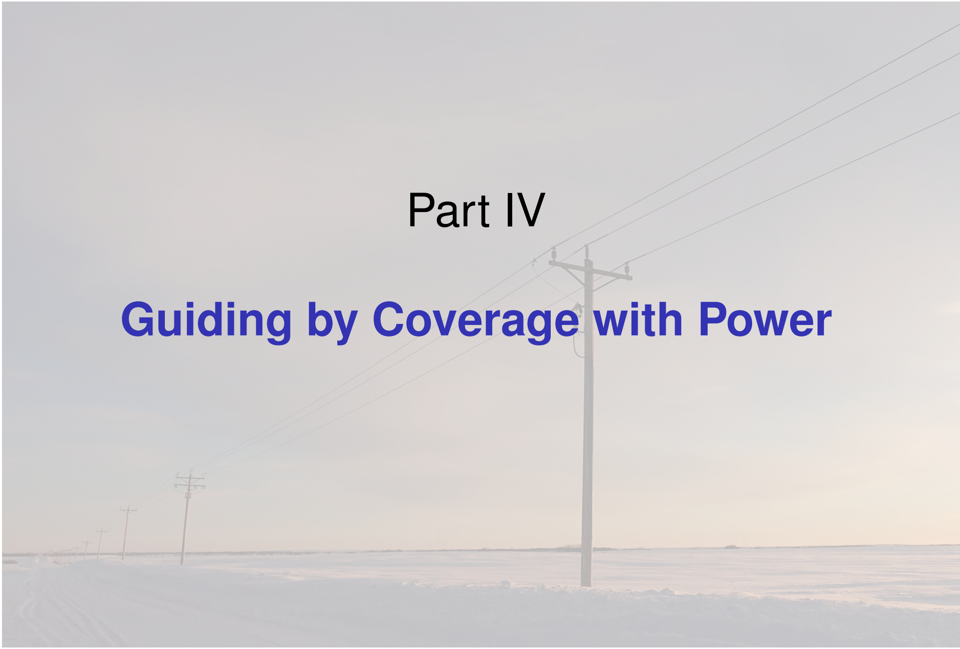
Coverage over time: blackbox vs greybox (from Fuzzing Book)



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.



Part IV

Guiding by Coverage with Power

AFL does a bit more than we've said so far

So far: draw seeds (paths) uniformly at random from population.

But: some paths more important than others. Important paths should come up more often.

Concept: Power schedule

Power schedule: assigns energy value (floating-point) to each seed in the population.

Allows the fuzzer to prioritize higher-energy (presumably higher-value) seeds—
it randomly selects a seed from the population consistent with energy distribution.

Power Schedule

```
class PowerSchedule:
    def __init__(self) -> None:
        self.path_frequency: Dict = {}

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

    def normalizedEnergy(self, population: Sequence[Seed]) ->
                                                  List[float]:
        """omitted for space"""
        pass

    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
```

Default Power Schedule

```python
if __name__ == "__main__":
    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)
        hits[seed.data] += 1
    print (repr(hits))
```

yields:

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

AdvancedMutationFuzzer

```
def create_candidate(self) -> str:
    """Returns an input generated by fuzzing a seed in the
        population"""
    seed = self.schedule.choose(self.population)  # <--!

    # Stacking: Apply multiple mutations to generate the
        candidate
    candidate = seed.data
    trials = min(len(candidate), 1 << random.randint(1, 5))
    for i in range(trials):
        candidate = self.mutator.mutate(candidate)
    return candidate
```

We haven’t assigned any energy, so all seeds have energy 1.

Path IDs

Want to give unusual paths (those not exercised often) more energy.

First, define 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()
```

AFL vs AFLFast

Original AFL: energy is constant in # times seed chosen $s(i)$.

AFLFast: energy exponential in $s(i)$; from that paper:

> *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.*

```
class AFLFastSchedule(PowerSchedule):
  def assignEnergy(self, population) -> None:
    for seed in population:
      seed.energy = 1 / (self.path_frequency[getPathID(
                                    seed.coverage)] **
                                    self.exponent)
```

Counting Greybox Fuzzer

Adds to path frequency when a path is run:

```
    def run(self, runner: FunctionCoverageRunner) ->
                                            Tuple[Any, str]:
        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)
```

Counting Greybox Fuzzer: Results

```
fuzzer w/ exponential schedule 0.46s for 10000 inputs.
Our fuzzer w/exponential schedule covers 5 statements.
             path id 'p'                 : path frequency 'f(p)'
{'26...1854': 5468, 'bc...7bac': 2694,
 '6f...3853': 1119, 'f7...57a8': 452,
 '86...bbb5': 267}
fuzzer w/ original schedule 0.30s for 10000 inputs.
             path id 'p'                 : path frequency 'f(p)'
{'26...1854': 7538, 'bc...7bac': 2121,
 '6f...3853': 327,  'f7...57a8': 14}
```

Exponential is a bit slower than original, but more consistently hits 5 paths. Original schedule has low count for 5th path.

Counting Greybox Fuzzer: Normalized Energy

```
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'
```

Unusual path gets the vast majority of the energy under fast schedule;
all paths have same energy under the original schedule.

Another example: HTMLParser

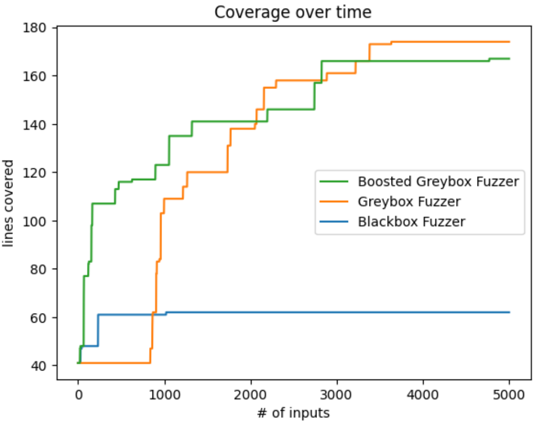
```python
from html.parser import HTMLParser

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

Results

Starting with $n = 5000$; single seed with one space.

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

Coverage over time (from Fuzzing Book)



HTMLParser: comments

Coverage:

- counting greybox does a bit better than the greybox;
- greybox does a lot better than the blackbox;

Inputs:

- blackbox doesn't find much;
- greybox includes brackets;
- counting greybox includes longer inputs.

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