

American Fuzzy Lop¹ 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. 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 parser².

So, let's first think about our domain—URLs. There is a definition of a valid URL³. 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 `schemes`, including `http`, `https`, `file`, etc.

Let's use a library function to parse URLs.

```
1 >>> from typing import Tuple, List, Callable, Set, Any
2 >>> from urllib.parse import urlparse
3
4 >>> urlparse("http://www.google.com/search?q=fuzzing")
5 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 def url_consumer(url: str) -> bool:
```

¹<http://lcamtuf.coredump.cx/afl/>

²Much of today's lecture is based on <https://www.fuzzingbook.org/html/MutationFuzzer.html>

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

```

2     supported_schemes = ["http", "https"]
3     result = urlparse(url)
4     if result.scheme not in supported_schemes:
5         raise ValueError("Scheme must be one of " +
6                           repr(supported_schemes))
7     if result.netloc == '':
8         raise ValueError("Host must be non-empty")
9
10    # Do something with the URL
11    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:

```

1  for i in range(1000):
2      try:
3          url = fuzzer()
4          result = url_consumer(url)
5          print("Success!")
6      except ValueError:
7          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.

```

1 def delete_random_character(s: str) -> str:
2     """Returns s with a random character deleted"""
3     if s == "":
4         return s
5
6     pos = random.randint(0, len(s) - 1)

```

```

7     # print (" Deleting ", repr(s[pos]), " at ", pos)
8     return s[:pos] + s[pos + 1:]
9
10 def insert_random_character(s: str) -> str:
11     """Returns s with a random character inserted"""
12     pos = random.randint(0, len(s))
13     random_character = chr(random.randrange(32, 127))
14     # print (" Inserting ", repr(random_character), " at ", pos)
15     return s[:pos] + random_character + s[pos:]
16
17 def flip_random_character(s):
18     """Returns s with a random bit flipped in a random position"""
19     if s == "":
20         return s
21
22     pos = random.randint(0, len(s) - 1)
23     c = s[pos]
24     bit = 1 << random.randint(0, 6)
25     new_c = chr(ord(c) ^ bit)
26     # print (" Flipping ", bit, " in ", repr(c) + ", giving ", repr(new_c))
27     return s[:pos] + new_c + s[pos + 1:]

```

We can run these functions:

```

1 seed_input = "A quick brown fox"
2 for i in range(10):
3     x = delete_random_character(seed_input)
4     print(repr(x))
5
6 for i in range(10):
7     print(repr(insert_random_character(seed_input)))
8
9 for i in range(10):
10    print(repr(flip_random_character(seed_input)))

```

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

```

1 def mutate(s: str) -> str:
2     """Return s with a random mutation applied"""
3     mutators = [
4         delete_random_character,
5         insert_random_character,
6         flip_random_character
7     ]
8     mutator = random.choice(mutators)
9     # print (mutator)
10    return mutator(s)

```

and call that function:

```

1 for i in range(10):
2     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:

```

1 def is_valid_url(url: str) -> bool:
2     try:
3         result = url_consumer(url)
4         return True
5     except ValueError:
6         return False
7
8 assert is_valid_url("http://www.google.com/search?q=fuzzing")
9 assert not is_valid_url("xyzzy")

```

We can now use our `mutate` function:

```

1 seed_input = "http://www.google.com/search?q=fuzzing"
2 valid_inputs = set()
3 trials = 20
4
5 for i in range(trials):
6     inp = mutate(seed_input)
7     if is_valid_url(inp):
8         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.

```

1 seed_input = "http://www.google.com/search?q=fuzzing"
2 mutations = 50
3 inp = seed_input
4 for i in range(mutations):
5     if i % 5 == 0:
6         print(i, "mutations:", repr(inp))
7     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.

```
1 class MutationFuzzer(Fuzzer):
2     """Base class for mutational fuzzing"""
3
4     def __init__(self, seed: List[str],
5                  min_mutations: int = 2,
6                  max_mutations: int = 10) -> None
7         # ...
8
9     def reset(self) -> None:
10        """Set population to initial seed.
11        To be overloaded in subclasses."""
12        self.population = self.seed
13        self.seed_index = 0
14
15    def create_candidate(self) -> str:
16        """Create a new candidate by mutating a population member"""
17        candidate = random.choice(self.population)
18        trials = random.randint(self.min_mutations, self.
19                                  max_mutations)
20        for i in range(trials):
21            candidate = self.mutate(candidate)
22        return candidate
23
24    def fuzz(self) -> str:
25        if self.seed_index < len(self.seed):
26            # Still seeding
27            self.inp = self.seed[self.seed_index]
28            self.seed_index += 1
29        else:
30            # Mutating
31            self.inp = self.create_candidate()
32        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:

```

1 seed_input = "http://www.google.com/search?q=fuzzing"
2 mutation_fuzzer = MutationFuzzer(seed=[seed_input])
3 print(mutation_fuzzer.fuzz())
4 print(mutation_fuzzer.fuzz())
5 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 inputs⁴:

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.

John Regehr discusses this issue at greater length⁵ and concludes that generational fuzzing tools should operate at all levels, rather than restricting themselves to only some of the levels.

⁴<http://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf>

⁵blog.regehr.org/archives/1039

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 successfully and `FAIL` if the function raises an exception.

```
1 class FunctionRunner(Runner):
2     def __init__(self, function: Callable) -> None:
3         """Initialize. 'function' is a function to be executed"""
4         self.function = function
5
6     def run_function(self, inp: str) -> Any:
7         return self.function(inp)
8
9     def run(self, inp: str) -> Tuple[Any, str]:
10        try:
11            result = self.run_function(inp)
12            outcome = self.PASS
13        except Exception:
14            result = None
15            outcome = self.FAIL
16
17        return result, outcome
```

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

```
1 http_runner = FunctionRunner(url_consumer)
2 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:

```
1 class FunctionCoverageRunner(FunctionRunner):
2     def run_function(self, inp: str) -> Any:
3         with Coverage() as cov:
4             try:
5                 result = super().run_function(inp)
6             except Exception as exc:
7                 self._coverage = cov.coverage()
8                 raise exc
9
10            self._coverage = cov.coverage()
```

```

11     return result
12
13     def coverage(self) -> Set[Location]:
14         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.

```

1 class MutationCoverageFuzzer(MutationFuzzer):
2     """Fuzz with mutated inputs based on coverage"""
3
4     def reset(self) -> None:
5         super().reset()
6         self.coverages_seen: Set[frozenset] = set()
7         # Now empty; we fill this with seed in the first fuzz runs
8         self.population = []
9
10    def run(self, runner: FunctionCoverageRunner) -> Any:
11        """Run function(inp) while tracking coverage.
12            If we reach new coverage,
13            add inp to population and its coverage to
14            population_coverage
15        """
16        result, outcome = super().run(runner)
17        new_coverage = frozenset(runner.coverage())
18        if outcome == Runner.PASS and new_coverage not in self.
19            coverages_seen:
20                # We have new coverage
21                self.population.append(self.inp)
22                self.coverages_seen.add(new_coverage)
23
24    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\x7fge.c*om/{earcq=fuzzing’,
‘http://\www=go\x7fge.c*nm{erchq=fuzzig’, ‘http://www.google.com/ear#h?q=fuzzing’,
‘http://\www=g\x7fgienc*nKmer;chq=nuzzig’, ‘http://www.goog?le.com/?ear#h?q9fuzzing’,
‘http://\www=g\x7fgienC*nKmer;#hq=nuzzig’, ‘http://\www=g\x7fgiec*n(ier;c/hq=nuZrig.’,
‘http://\wwi=w/ag/\x7figenC*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.