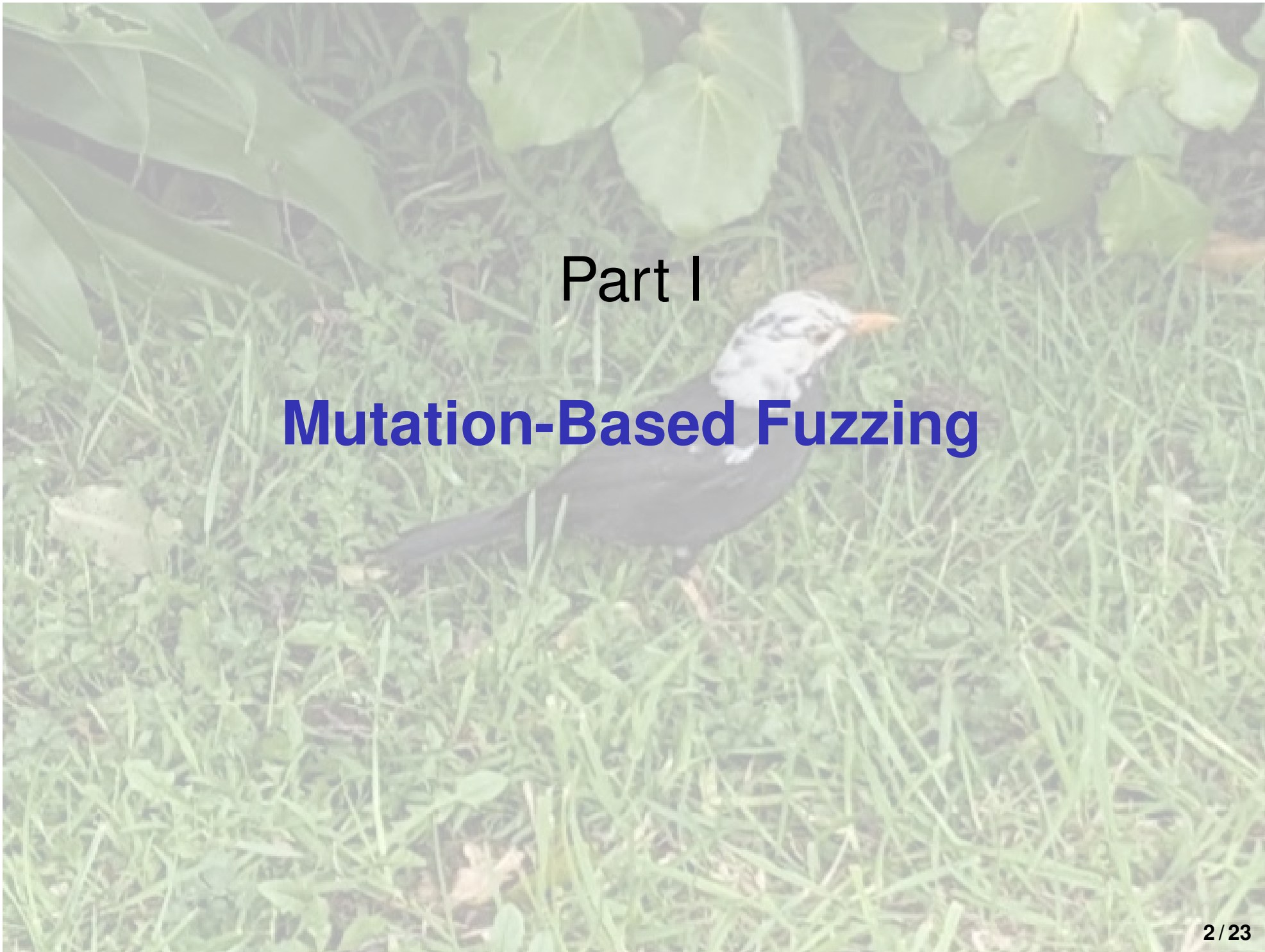
Software Testing, Quality Assurance & Maintenance—Lecture 8

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
```

Hierarchy of inputs: C

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.

Mutation-based Fuzzing

Develop a tool that randomly modifies existing inputs:

- totally randomly, by flipping bytes in the input; or,
- 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 (similar to level 3 above).