Program Analysis

Random Testing and Fuzzing

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2023/2024

Warm-up Quiz

What does this JavaScript code print?

```
function f(a,b) {
  var x;
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}
f(1,2,3);
```

3 **6** **NaN** **Nothing**

Warm-up Quiz

What does this JavaScript code print?

```
function f(a,b) {
  var x;
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}
f(1,2,3);
```

3 **6** **NaN** **Nothing**

Warm-up Quiz

What does this JavaScript code print?

```
function f(a,b) {
  var x;
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}
f(1,2,3);
```

Array-like object that contains all three arguments

3 **6** **NaN** **Nothing**

Warm-up Quiz

What does this JavaScript code print?

```
function f(a,b) {
  var x;
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}
f(1,2,3);
```

Initialized to `undefined`

`undefined` + some number yields `NaN`

3 **6** **NaN** **Nothing**

Automated Testing

- **Manual testing**
  - Important but limited by human time
- **Automated testing**
  - Test execution: Regularly execute regression test suite
  - Test creation: Automatic test generation

Automated Testing

- **Manual testing**
  - Important but limited by human time
- **Automated testing**
  - Test execution: Regularly execute regression test suite
  - Test creation: Automatic test generation

Focus of this lecture

Kinds of Approaches

- **Blackbox**
  - No analysis of program
- **Greybox**
  - Lightweight analysis of program
  - E.g., coverage achieved by inputs
- **Whitebox**
  - More heavyweight analysis of program
  - E.g., conditions that trigger specific paths

Kinds of Approaches

- **Blackbox**
  - No analysis of program
- **Greybox**
  - Lightweight analysis of program
  - E.g., coverage achieved by inputs
- **Whitebox**
  - More heavyweight analysis of program
  - E.g., conditions that trigger specific paths

This lecture

Kinds of Approaches

- **Blackbox**
  - No analysis of program
- **Greybox**
  - Lightweight analysis of program
  - E.g., coverage achieved by inputs

Next lecture

- **Whitebox**
  - More heavyweight analysis of program
  - E.g., conditions that trigger specific paths

Kinds of Approaches

- **Blackbox**
  - No analysis of program
- **Greybox**
  - Lightweight analysis of program
  - E.g., coverage achieved by inputs
- **Whitebox**
  - More heavyweight analysis of program
  - E.g., conditions that trigger specific paths

All of them: Use feedback from test executions

What's "the Program"?

- **Many possible answers**
  - Individual function
  - Class and its methods
  - Entire library
  - Entire stand-alone tool
- **Ideas discussed here work (in principle) on multiple levels**

Outline

- **Introduction**
- **Randoop** ←
  - Based on *Feedback-Directed Random Test Generation*, Pacheco et al., ICSE 2007
- **Greybox fuzzing in AFL**
  - Based on https://lcamtuf.coredump.cx/afl/technical_details.txt

Motivating Examples

Two randomly generated tests:

```
Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```

```
Set s = new HashSet();
s.add("hi");
s.isEmpty();
assertTrue(s.equals(s));
```

Motivating Examples

Two randomly generated tests:

```
Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```

```
Set s = new HashSet();
s.add("hi");
s.isEmpty();
assertTrue(s.equals(s));
```

Only difference

Motivating Examples

Two randomly generated tests:

```
Set s = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```

```
Set s = new HashSet();
s.add("hi");
s.isEmpty();
assertTrue(s.equals(s));
```

Redundant test

Motivating Examples (2)

Three randomly generated tests:

```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

Motivating Examples (2)

Three randomly generated tests:

```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

Violates pre-condition

Motivating Examples (2)

Three randomly generated tests:

```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
assertTrue(d.equals(d));
```

```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

Illegal tests

Feedback-directed Test Generation

Idea: Guide randomized creation of new test inputs by feedback about execution of previous inputs

- Avoid redundant inputs
- Avoid illegal inputs

- Test input here means sequence of method calls
- Software under test: Classes in Java-like language

Approach

- **Build test inputs incrementally**
  - New test inputs extend previous ones
- **As soon as test input is created, execute it**
- **Use execution results to guide generation**
  - away from redundant or illegal method sequences
  - toward sequences that create new object states

Randoop: Input/Output

Randoop: Implementation of feedback-directed random test generation

- Input:
  - Classes under test
  - Time limit
  - Set of contracts
    - Method contracts, e.g., `o.hashCode()` throws no exception
    - Object invariants, e.g., `o.equals(o) == true`
- Output: Test cases with assertions

Example

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

Example

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

Fails when executed

Example

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

No contracts violated up to last method call

Fails when executed

Algorithm

1. Initialize seed components: `i=0; b=false; ...`
2. Do until time limit expires:
   - Create a new sequence
     - Randomly pick a method $T_0.m(T_1, ..., T_k)/T_{ret}$
     - For each T_i, randomly pick a sequence S_i from the components that constructs a value v_i of type T_i
     - Create new sequence $S_{new} = S_1; ...; S_k; T_{ret}\ v_{new} = m(v_1, ..., v_k);$
     - If S_{new} was previously created (lexically), go to
   - Classify the sequence S_{new}
     - May discard, output as test case, or add to components

Classifying a Sequence

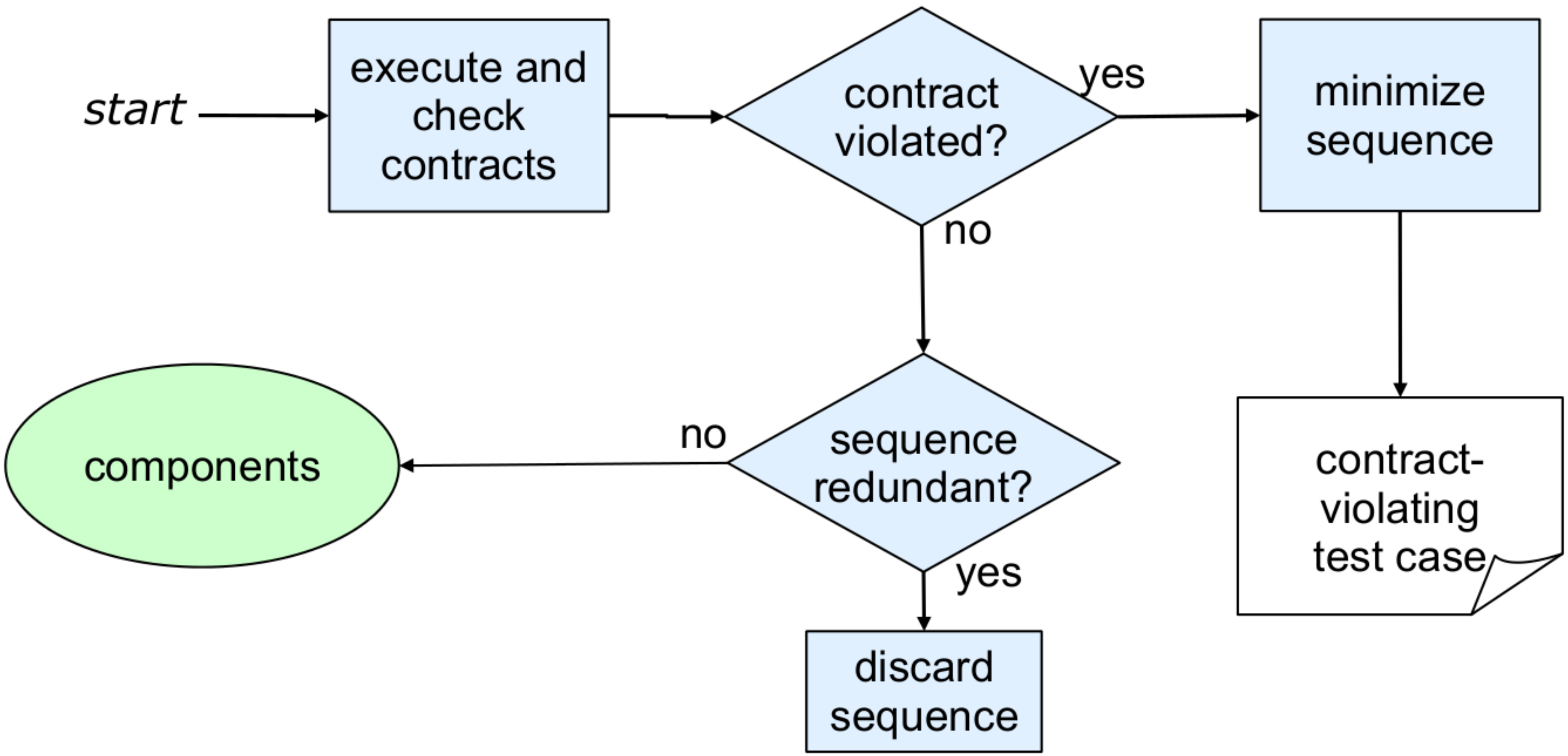


Image source: Slides by Pacheco et al.

Redundant Sequences

- During generation, maintain a set of all objects created
- Sequence is redundant if all objects created during its execution are in the above set (using `equals()` to compare)
- Could also use more sophisticated state equivalence methods
  - E.g., heap canonicalization

Classes under test: java.util.*

1) Pick a method — new HashMap()

→ No values needed

→ New sequence — HashMap m = new HashMap()

2) Classify

→ No contract violated

→ Not redundant

⇒ Add to components

3) Pick a method

→ Sequence

new HashMap()

HashMap m2 = new HashMap()

4) Classify sequence

→ No contract violated

→ Redundant

⇒ Discard sequence

5) Pick method HashMap.values()

→ Need sequence that constructs a value of type HashMap

→ Use sequence from step 2)

⇒ Create sequence

```
HashMap m = new HashMap()
Collection c = m.values()
```

6) Classify sequence

→ No contract violated

→ Not redundant

⇒ Add to components

Test Oracles

- **Testing only useful if there is an oracle**
- **Randoop outputs two kinds of oracles**
  - Oracle for contract-violating test cases:

```
assertTrue(u.equals(u));
```

  - Oracle for normal-behavior test cases:

```
assertEquals(2, l.size());
assertEquals(false, l.isEmpty());
```

Quiz

Which of these tests may be created by Randoop?

Test 1:

```
LinkedList l = new LinkedList();
l.add(23);
```

Test 2:

```
LinkedList l = new LinkedList();
l.get(-5);
```

Test 3:

```
LinkedList l = new LinkedList();
l.add(7);
assertEquals(l.getFirst(), 7);
```

Quiz

Which of these tests may be created by Randoop?

Test 1:

```
LinkedList l = new LinkedList();
l.add(23);
```

(oracle missing)

Test 2:

```
LinkedList l = new LinkedList();
l.get(-5);
```

(crashes)

Test 3:

```
LinkedList l = new LinkedList();
l.add(7);
assertEquals(l.getFirst(), 7);
```

Results

- Applied to data structure implementations and popular library classes
- Achieves 80-100% basic block coverage
- Finds various bugs in JDK collections, classes from the .NET framework, and Apache libraries

Read Pacheco et al.'s paper for details

Outline

- **Introduction**
- **Randoop**
  - Based on *Feedback-Directed Random Test Generation*, Pacheco et al., ICSE 2007
- **Greybox fuzzing in AFL** ←
  - Based on https://lcamtuf.coredump.cx/afl/technical_details.txt

Greybox Fuzzing

- **Guide input generation toward a goal**
  - Guidance based on lightweight program analysis
- **Three main steps**
  - Randomly generate inputs
  - Get feedback from test executions: What code is covered?
  - Mutate inputs that have covered new code

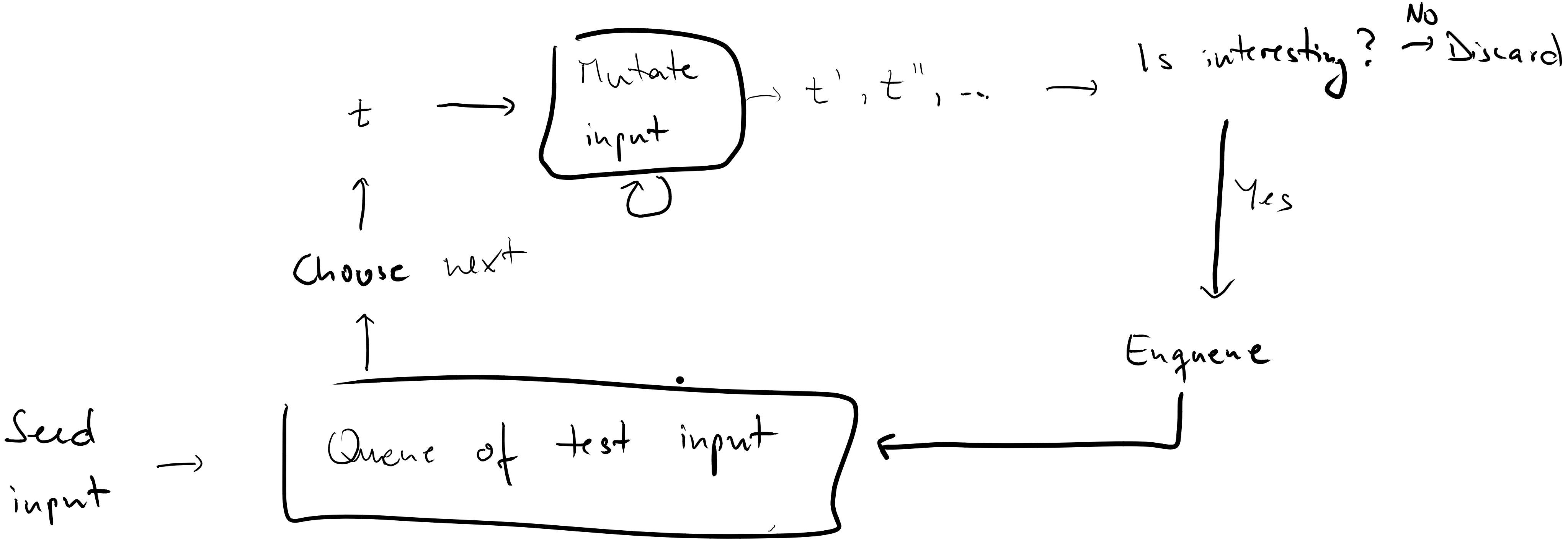
American Fuzzy Lop

American Fuzzy Lop

American Fuzzy Lop

- **Simple yet effective fuzzing tool**
  - Targets C/C++ programs
  - Inputs are, e.g., files read by the program
- **Widely used in industry**
  - In particular, to find security-related bugs
  - E.g., in OpenSSL, PHP, Firefox

Overview of AFL :



Measuring Coverage

- **Different coverage metrics**
  - Line/statement/branch/path coverage
- **Here: Branch coverage**
  - Branches between basic blocks
  - Rationale: Reaching a code location not enough to trigger a bug, but state also matters
  - Compromise between
    - Effort spent on measuring coverage
    - Guidance it provides to the fuzzer

Example

A

if (x > 3) {

B

C

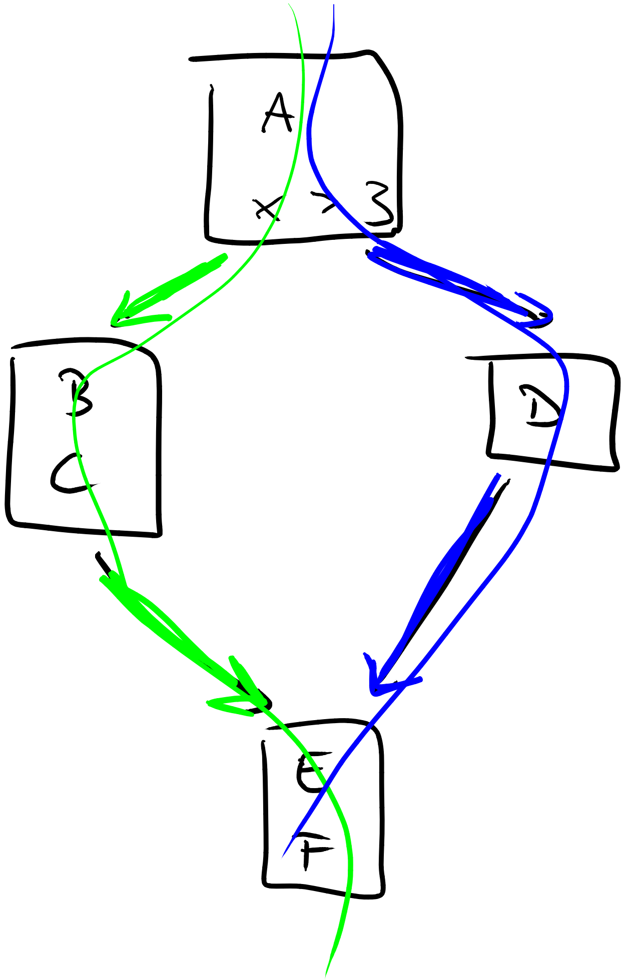
} else {

D

}

E

F



Execution 1

Execution 2

Efficient Implementation

- **Instrumentation added at branching points:**

```
cur_location = /*COMPILE_TIME_RANDOM*/;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Efficient Implementation

- **Instrumentation added at branching points:**

```
cur_location = /*COMPILE_TIME_RANDOM*/;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Advantage:
Works well with separate compilation

Efficient Implementation

- **Instrumentation added at branching points:**

```
cur_location = /*COMPILE_TIME_RANDOM*/;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Globally reachable memory location that stores how often each edge was covered

Efficient Implementation

- **Instrumentation added at branching points:**

```
cur_location = /*COMPILE_TIME_RANDOM*/;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Combine previous and current block into a fixed-size hash

Efficient Implementation

- **Instrumentation added at branching points:**

```
cur_location = /*COMPILE_TIME_RANDOM*/;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Shift to distinguish between “A” followed by “B” from “B” followed by “A”

Detecting New Behaviors

- **Inputs that trigger a new edge in the CFG: Considered as new behavior**
- **Alternative: Consider new paths**
  - More expensive to track
  - Path explosion problem

Example:

Exec. 1: A → B → C → D → E new

Exec. 2: A → B → C → A → E new

Exec. 3: A → B → C → A → B → C → A → B → C → D → E not new

Edge Hit Counts

- **Refinement of the previous definition of “new behaviors”**
- **For each edge, count how often it is taken**
  - Approximate counts based on buckets of increasing size
    - 1, 2, 3, 4-7, 8-15, 16-31, etc.
  - Rationale: Focus on relevant differences in the hit counts

Evolving the Input Queue

- **Maintain queue of inputs**
  - Initially: Seed inputs provided by user
  - Once used, keep input if it covers new edges
  - Add new inputs by mutating existing input
- **In practice: Queue sizes of 1k to 10k**

Mutation Operators

- **Goal: Create new inputs from existing inputs**
- **Random transformations of bytes in an existing input**
  - Bit flips with varying lengths and stepovers
  - Addition and subtraction of small integers
  - Insertion of known interesting integers
    - E.g., 0, 1, INT_MAX
  - Splicing of different inputs

More Tricks for Fast Fuzzing

- **Time and memory limits**
  - Discard input when execution is too expensive
- **Pruning the queue**
  - Periodically select subset of inputs that still cover every edge seen so far
- **Prioritize how many mutants to generate from an input in the queue**
  - E.g., focus on unusual paths or try to reach specific locations

Real-World Impact

- **Open-source tool maintained mostly by Google**
  - Initially created by single developer
  - Various improvements proposed in academia and industry
- **Fuzzers regularly check various security-criticial components**
  - Many thousands of compute hours
  - Hundreds of detected vulnerabilities