Program Analysis

Random Testing and Fuzzing

(Part 3)
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

Send input → Queue of test inputs

Choose next → t → t', t'', ...

Mutate input

is interesting?

No → Discard

Yes → Enqueue
Control Flow Graphs - Basic Blocks

A
if (x > 3) {
    B
    C
} else {
    D
}
E
F

Basic: Sequence of operations/insts.
always executed together
(i.e. no branch in between)
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**

<table>
<thead>
<tr>
<th>Sequence of basic that are executed</th>
<th>Branches covered (i.e., edges in CFG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A → B → C → D → E</td>
<td>AB, BC, CD, DE</td>
</tr>
<tr>
<td>A → B → D → C → E</td>
<td>AB, BD, DC, CE</td>
</tr>
</tbody>
</table>
Efficient Implementation

- Instrumentation added at branching points:

```c
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:

```c
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:

  ```c
  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:

```c
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

Ex. 1: \( A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \) new

Ex. 2: \( A \rightarrow B \rightarrow C \rightarrow A \rightarrow E \) new

Ex. 3: \( A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \) 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-critical components**
  - Many *thousands of compute hours*
  - Hundreds of detected vulnerabilities