Random Testing

Mayur Naik

CIS 573 – Fall 2019
Random Testing (Fuzzing)

- Feed random inputs to a program

- Observe whether it behaves “correctly”
  - Execution satisfies given specification
  - Or just doesn’t crash
    - A simple specification

- Special case of mutation analysis
The Infinite Monkey Theorem

“A monkey hitting keys at random on a typewriter keyboard will produce any given text, such as the complete works of Shakespeare, with probability approaching 1 as time increases.”
Random Testing: Case Studies

• UNIX utilities: Univ. of Wisconsin’s Fuzz study

• Mobile apps: Google’s Monkey tool for Android

• Concurrent programs: Cuzz tool from Microsoft
A Popular Fuzzing Study

• Conducted by Barton Miller @ Univ of Wisconsin

• 1990: Command-line fuzzer, testing reliability of UNIX programs
  – Bombards utilities with random data

• 1995: Expanded to GUI-based programs (X Windows), network protocols, and system library APIs

• Later: Command-line and GUI-based Windows and OS X apps
Fuzzing UNIX Utilities: Aftermath

• **1990:** Caused 25-33% of UNIX utility programs to crash (dump state) or hang (loop indefinitely)

• **1995:** Systems got better... but not by much!

“Even worse is that many of the same bugs that we reported in 1990 are still present in the code releases of 1995.”
A Silver Lining: Security Bugs

• `gets()` function in C has no parameter limiting input length
  ⇒ programmer must make assumptions about structure of input

• Causes reliability issues and security breaches
  – Second most common cause of errors in 1995 study

• Solution: Use `fgets()`, which includes an argument limiting the maximum length of input data
Fuzz Testing for Mobile Apps

class MainActivity extends Activity implements OnClickListener {
    void onCreate(Bundle bundle) {
        Button buttons = new Button[] { play, stop, ... };  
        for (Button b : buttons) b.setOnClickListener(this);
    }
    void onClick(View target) {
        switch (target) {
        case play:
            startService(new Intent(ACTION_PLAY));
            break;
        case stop:
            startService(new Intent(ACTION_STOP));
            break;
        ...
        }
    }
}
Generating Single-Input Events

class MainActivity extends Activity implements OnClickListener {
    void onCreate(Bundle bundle) {
        Button buttons = new Button[] { play, stop, ... };  
        for (Button b : buttons) b.setOnClickListener(this);
    }

    void onClick(View target) {
        switch (target) {
        case play:  
            startActivity(new Intent(ACTION_PLAY));
            break;
        case stop:  
            startActivity(new Intent(ACTION_STOP));
            break;
        ...
        }
    }
}

**TOUCH(x, y)** where x, y are randomly generated:

x in [0..480], y in [0..800]
Black-Box vs. White-Box Testing

TOUCH(x1, y1)  TOUCH(x2, y2)  TOUCH(x3, y3)
Generating Gestures

DOWN\((x_1, y_1)\) MOVE\((x_2, y_2)\) UP\((x_2, y_2)\)
Grammar of Monkey Events

test_case := event *

event := action ( x, y ) | ...

action := DOWN | MOVE | UP

x := 0 | 1 | ... | x_limit

y := 0 | 1 | ... | y_limit
QUIZ: Monkey Events

Give the correct specification of TOUCH and MOTION events in Monkey’s grammar using UP, MOVE, and DOWN statements.

Give the specification of a TOUCH event at pixel (89,215).

Give the specification of a MOTION event from pixel (89,215) to pixel (89,103) to pixel (371,103).
QUIZ: Monkey Events

Give the correct specification of TOUCH and MOTION events in Monkey’s grammar using UP, MOVE, and DOWN statements.

Give the specification of a TOUCH event at pixel (89,215).

DOWN(89,215) UP(89,215)

TOUCH events are a pair of DOWN and UP events at a single place on the screen.

Give the specification of a MOTION event from pixel (89,215) to pixel (89,103) to pixel (371,103).

DOWN(89,215) MOVE(89,103) MOVE(37,103) UP(37,103)

MOTION events consist of a DOWN event somewhere on the screen, a sequence of MOVE events, and an UP event.
Testing Concurrent Programs

Input:

```
p    null
```

Sequential Program:

```
...  
...  
p.close();  
...  
```

Exception
Testing Concurrent Programs

Input:

Thread 1:

```java
...  
p = null;
```

Thread 2:

```java
...  
if (p != null) {
    ...
    p.close();
}
```

Exception
Concurrency Testing in Practice

Thread 1:
```
Sleep();
p = null;
```

Thread 2:
```
Sleep();
if (p != null) {
    Sleep();
    p.close();
}
```

Input:
```
p → new File()
```

Exception
Cuzz: Fuzzing Thread Schedules

• Introduces `Sleep()` calls:
  • Automatically (instead of manually)
  • Systematically before each statement (instead of those chosen by tester)

=> Less tedious, less error-prone

• Gives worst-case probabilistic guarantee on finding bugs
Depth of a Concurrency Bug

• Bug Depth = the number of ordering constraints a schedule has to satisfy to find the bug
Bug Depth: Example 1

• Bug Depth = the number of ordering constraints a schedule has to satisfy to find the bug

```
Thread 1:
...
T t = new T();
...
...
```

```
Thread 2:
...
...
if (t.state == 1)
...
...
```
Bug Depth: Example 2

- Bug Depth = the number of ordering constraints a schedule has to satisfy to find the bug

Thread 1:

```java
...  
p = null;
...
```

Thread 2:

```java
...  
if (p != null) {
   ...
   p.close();
}
```
Depth of a Concurrency Bug

• Bug Depth = the number of ordering constraints a schedule has to satisfy to find the bug

• Observation exploited by Cuzz: bugs typically have small depth
QUIZ: Concurrency Bug Depth

Specify the depth of the concurrency bug in the following example:

Then specify all ordering constraints needed to trigger the bug. Use the notation \((x, y)\) to mean statement \(x\) comes before statement \(y\), and separate multiple constraints by a space.

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: lock(a);</td>
<td>6: lock(b);</td>
</tr>
<tr>
<td>2: lock(b);</td>
<td>7: lock(a);</td>
</tr>
<tr>
<td>3: ( g = g + 1; )</td>
<td>8: ( g = 0; )</td>
</tr>
<tr>
<td>4: unlock(b);</td>
<td>9: unlock(a);</td>
</tr>
<tr>
<td>5: unlock(a);</td>
<td>10: unlock(b);</td>
</tr>
</tbody>
</table>
QUIZ: Concurrency Bug Depth

Specify the depth of the concurrency bug in the following example:

Then specify all ordering constraints needed to trigger the bug. Use the notation \((x,y)\) to mean statement \(x\) comes before statement \(y\), and separate multiple constraints by a space.

\[(1,7) \ (6,2)\]

Thread 1:

1: lock(a);
2: lock(b);
3: \(g = g + 1\);
4: unlock(b);
5: unlock(a);

Thread 2:

6: lock(b);
7: lock(a);
8: \(g = 0\);
9: unlock(a);
10: unlock(b);
Cuzz Algorithm

Input:
\[
\begin{align*}
  &\text{int } n; & \text{// # of threads} \\
  &\text{int } k; & \text{// # of steps - guessed from previous runs} \\
  &\text{int } d; & \text{// target bug depth - randomly chosen}
\end{align*}
\]

State:
\[
\begin{align*}
  &\text{int } \text{pri}[] = \text{new int}[n]; & \text{// thread priorities} \\
  &\text{int } \text{change}[] = \text{new int}[d-1]; & \text{// when to change priorities} \\
  &\text{int } \text{stepCnt}; & \text{// current step count}
\end{align*}
\]

**Initialize()**

\[
\begin{align*}
  &\text{stepCnt} = 0; \\
  &\text{a} = \text{random\_permutation}(1,n); \\
  &\text{for (int } \text{tid} = 0; \text{tid} < n; \text{tid}++) \\
    &\quad \text{pri}[\text{tid}] = \text{a}[\text{tid}] + d; \\
  &\text{for (int } i = 0; i < d-1; i++) \\
    &\quad \text{change}[i] = \text{rand}(1,k);
\end{align*}
\]

**Sleep(tid)**

\[
\begin{align*}
  &\text{stepCnt}++; \\
  &\text{if (stepCnt == change[i] for some } i) \\
    &\quad \text{pri}[\text{tid}] = i; \\
  &\text{while (tid is not highest priority enabled thread)} \\
    &\quad ;
\end{align*}
\]
Probabilistic Guarantee

Given a program with:

- n threads (~tens)
- k steps (~millions)
- bug of depth d (1 or 2)

Cuzz will find the bug with a probability of at least

\[
\frac{1}{n \cdot k^{d-1}}
\]
in each run
Proof of Guarantee (Sketch)

Thread 1
y: p = null;

Thread 2
x: if (p != null)

z: p.close();

Probability(choose correct initial thread priorities) >= 1 / n
Probability(choose correct step to switch thread priorities) >= 1 / k
Probability(triggering bug) >= 1 / (nk)
Proof of Guarantee (Sketch)

Thread 1
...
... y: p = null;
... ...
...
...

Thread 2
2 x: if (p != null) ...
3 ... ...
1 ... ...
... z: p.close(); ...

Probability(choose correct initial thread priorities) >= 1 / n

Probability(choose correct step to switch thread priorities) >= 1 / k

Probability(triggering bug) >= 1 / (nk)
Measured vs. Worst-Case Probability

- Worst-case guarantee is for hardest-to-find bug of given depth
- If bugs can be found in multiple ways, probabilities add up!
- Increasing number of threads helps
  - Leads to more ways of triggering a bug
Cuzz Case Study

Measure bug-finding probability of stress testing vs. Cuzz

- Without Cuzz: 1 Fail in 238,820 runs
  - ratio = 0.000004187
- With Cuzz: 12 Fails in 320 runs
  - ratio = 0.0375

1 day of stress testing = 11 seconds of Cuzz testing!
Cuzz: Key Takeaways

• Bug depth: useful metric for concurrency testing efforts

• Systematic randomization improves concurrency testing

• Whatever stress testing can do, Cuzz can do better
  – Effective in flushing out bugs with existing tests
  – Scales to large number of threads, long-running tests
  – Low adoption barrier
Random Testing: Pros and Cons

Pros:
• Easy to implement
• Provably good coverage given enough tests
• Can work with programs in any format
• Appealing for finding security vulnerabilities

Cons:
• Inefficient test suite
• Might find bugs that are unimportant
• Poor coverage
Coverage of Random Testing

- The lexer is very heavily tested by random inputs
- But testing of later stages is much less efficient
What Have We Learned?

Random testing:

• Is effective for testing security, mobile apps, and concurrency
• Should complement not replace systematic, formal testing
• Must generate test inputs from a reasonable distribution to be effective
• May be less effective for systems with multiple layers (e.g. compilers)