Program Testing and Analysis: Random and Fuzz Testing

Prof. Dr. Michael Pradel
Software Lab, TU Darmstadt
What does the following code print?

```javascript
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 the following code print?

```javascript
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 the following code print?

```javascript
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

- Nothing
- 3
- 6
- NaN
What does the following code print?

```javascript
function f(a,b) {
  var x; // Initialized to undefined
  for (var i = 0; i < arguments.length; i++) {
    x += arguments[i];
  }
  console.log(x);
}

f(1,2,3);
```

3 6 NaN

3 Nothing
Outline

- Feedback-directed random test generation
  Based on *Feedback-Directed Random Test Generation*, Pacheco et al., ICSE 2007

- Adaptive random testing
  Based on *ARTOO: Adaptive Random Testing for Object-oriented Software*, Ciupa et al., ICSE 2008

- Fuzz testing
  Based on *Fuzzing with Code Fragments*, Holler et al., USENIX Security 2012
Adaptive Random Testing

Idea: Testing is more effective when inputs are spread evenly over the input domain

- Generate candidate inputs randomly
- At every step, select input that is furthest away from already tested inputs
Spread Out Evenly?

- Initially proposed for **numeric values**
  - Distance between two values: **Euclidean distance**

- **Example:** \( f(int \ x) \)
  - Suppose to have tested with `Integer.MAX_VALUE` and `Integer.MIN_VALUE`
  - Next test: 0
Spread Out Evenly?

- Initially proposed for **numeric values**
  - Distance between two values: **Euclidean distance**

- **Example:** \( f(int\ x) \)
  - Suppose to have tested with \( \text{Integer.MAX\_VALUE} \) and \( \text{Integer.MIN\_VALUE} \)
  - Next test: 0

**Challenge:**
How to compute distance of objects?
Adaptive random testing

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

\{ \text{space of all possible inputs} \}
Object Distance

- **Measure** how different two objects are
- **Object:** Primitive values, dynamic type, and non-primitive values recursively referred to

\[
dist(p, q) = \text{combination}(
\begin{align*}
\text{elementaryDistance}(p, q), \\
\text{typeDistance}(\text{type}(p), \text{type}(q)), \\
\text{fieldDistance}(&\{\text{dist}(p.a, q.a) \mid \\
& a \in \text{fields}(\text{type}(p) \cap \text{fields}(\text{type}(q)))\})
\end{align*}
\)
Object Distance

- Measure **how different two objects are**
- Object: Primitive values, dynamic type, and non-primitive values recursively referred to

\[
dist(p, q) = \text{combination(}
\begin{align*}
\text{elementaryDistance}(p, q), \\
\text{typeDistance}(\text{type}(p), \text{type}(q)), \\
\text{fieldDistance}\{\text{dist}(p.a, q.a) \mid a \in \text{fields}(\text{type}(p) \cap \text{fields}(\text{type}(q)))\}\}
\end{align*}
\]

Does not require traversing the object
Object Distance

- **Measure** how different two objects are
- **Object**: Primitive values, dynamic type, and non-primitive values recursively referred to

\[
dist(p, q) = \text{combination}(
\quad \text{elementaryDistance}(p, q), \quad
typeDistance(\text{type}(p), \text{type}(q)), \quad
typeDistance(\{\text{dist}(p.a, q.a) \mid a \in \text{fields} (\text{type}(p) \cap \text{fields}(\text{type}(q)))\})
\)

Recursively defined
Elementary Distance

Fixed functions for each possible type:

- For **numbers**: $F(|p - q|)$, where $F$ is a monotonically non-decreasing function with $F(0) = 0$
- For **characters**: 0 if identical, $C$ otherwise
- For **booleans**: 0 if identical, $B$ otherwise
- For **strings**: the Levenshtein distance
- For **references**: 0 if identical, $R$ if different but none is null, $V$ if only one of them is null

$C, B, R, V \in \mathbb{N}$
Examples: Elementary Distance

- int i = 3, j = 9 → dist(i, j) = \( \pi(13 - 9) = 6 \)

- char c = 'a', d = 'a' → dist(c, d) = 0

- String s = "foo", t = "too" → dist(s, t) = 1

- Object o = null, p = new ArrayList() → dist(o, p) = V
Type Distance

Distance between two types

\[ \text{typeDistance}(t, u) = \]
\[ \lambda \times \text{pathLength}(t, u) \]
\[ + \nu \times \sum_{a \in \text{nonShared}(t,u)} \text{weight}_a \]

- \text{pathLength}(t, u) is the minimal distance to a common ancestor in class hierarchy
- \text{nonShared}(t, u) is the set of non-shared fields
- \text{weight}_a is the weight for a specific field

\[ \lambda, \nu \in \mathbb{N} \]
Examples: Type Distance

\[
\text{dist} (B, C) = \lambda \cdot 1 + v \cdot (1 + 1)
\]

\[
\text{dist} (A, B) = \lambda \cdot 0 + v \cdot (1)
\]
Field Distance

Recursively compute distance of all shared fields

\[
\text{fieldDistance}(p, q) = \sum_a \text{weight}_a \ast (\text{dist}(p.a, q.a))
\]

Arithmetic mean: Avoid giving too much weight to objects with many fields
Algorithm for Selecting Inputs

- Global sets \textit{usedObjects} and \textit{candidateObjects}
- Choose object for next test:
  - Initialize $bestDistSum = 0$ and $bestObj = \text{null}$
  - for each $c \in \text{candidateObjects}$:
    - for each $u \in \text{usedObjects}$:
      - $distSum += dist(c,u)$
      - if $distSum > bestDistSum$
        - $bestDistSum = distSum; bestObj = c$
  - Remove $bestObj$ from \textit{candidateObjects}, add to \textit{usedObjects} instead, and run test with $bestObj$
Example

Method under test:
Account.transfer(Account dst, int amount)

Pool of candidates:

- Accounts
  - a1: owner="A" and balance=6782832
  - a2: owner="B" and balance=10
  - a3: owner="O" and balance=99
  - a4: null

- Integers:
  - i1: 100, i2: 287391, i3: 0, i4: -50
First call:
\texttt{a3.transfer(a1, i2)}

Second call:
\texttt{a1.transfer(a4, i4)}
Results

■ Implemented for Eiffel
■ Use randomly generated objects as candidates
■ Use Eiffel’s contracts (pre- and post-conditions, class invariants) as test oracle

Comparison with random testing:
- Find bugs with 5x fewer tests
- But: Takes 1.6x the time of random testing
Outline

- Feedback-directed random test generation
  Based on *Feedback-Directed Random Test Generation*, Pacheco et al., ICSE 2007

- Adaptive random testing
  Based on *ARTOO: Adaptive Random Testing for Object-oriented Software*, Ciupa et al., ICSE 2008

- Fuzz testing
  Based on *Fuzzing with Code Fragments*, Holler et al., USENIX Security 2012
Fuzz Testing

Generate random inputs

- **Generative**: Create new random input, possibly based on constraints and rules
- **Mutative**: Derive new inputs from existing input by randomly modifying it
Grammar-based Language Fuzzing

Idea: **Combine generative and mutative approach to test JavaScript interpreter**

- Create *random JavaScript programs* based on language grammar
- Use and re-compose *fragments of code* from existing corpus of programs
  - Corpus: Programs that have exposed bugs before
Overview of LangFuzz

Phase I
Learning code fragments from sample code and test suite

Phase II
LangFuzz generated (mutated) test cases

Phase III
Feed test case into interpreter, check for crashes and assertions
Learning Code Fragments

- Parse existing programs into ASTs
- Extract code fragments
  - Examples for non-terminals of grammar
Corpus:

expression E

command C

program P

\[ x := !y ; y := !z \]
Mutation of Code

- Randomly pick and parse an existing program

- Randomly pick some fragments and replace with fragments from phase 1 of same type
if !x > 4 then y := !z else skip

if !x > 4 then y := !z else x := !y

command from corpus
Generation of Code

Breadth-first application of grammar rules

- Set current expansion $e_{cur}$ to start symbol $P$
- Loop $k$ iterations:
  - Choose a random non-terminal $n$ in $e_{cur}$
  - Pick one of the rules, $r$, that can be applied to $n$
  - Replace occurrence of $n$ in $e_{cur}$ by $r(n)$

After $k$ iterations: Replace remaining non-terminals with fragments
\( h = 3 \)

\[
\begin{align*}
  e_{\text{var}} &= \mathcal{P} \\
  e_{\text{var}} &= C \\
  e_{\text{var}} &= \text{while } B \text{ do } C \\
  e_{\text{var}} &= \text{while } B \text{ do } C; C
\end{align*}
\]

\[
\begin{align*}
  P ::= C | \ldots \\
  C ::= \ldots | \text{while } B \text{ do } C | \ldots \\
  C ::= \ldots | C; C | \ldots \\
\end{align*}
\]

Replace with fragments learned from corpus.
Which of the following SIMP programs could have been generated by LangFuzz?

\[ \text{if } B \text{ then } C; \ C \]

\[ \text{if } !x > 3 \text{ then skip; } y := 1 \]

\[ \text{if } !x > 3 \text{ then while; while} \]
Quiz

Which of the following SIMP programs could have been generated by LangFuzz?

\[
\text{if } B \text{ then } C; \ C
\]

(has unexpanded non-terminals)

\[
\text{if } !x > 3 \text{ then skip; } y := 1
\]

(syntactically incorrect)

\[
\text{if } !x > 3 \text{ then while; } \text{while}
\]
Results

- Used to test Mozilla’s and Chrome’s JavaScript engines
- Found various bugs
  Mostly crashes of engine due to memory issues
- Rewarded with $50,000 bug bounties
- First author now works at Mozilla
Random and fuzz testing

- Fully automated and unbiased
- Non-naive approaches can be very effective
- Trade-off: Cost of generating inputs vs. effectiveness in exposing bugs
  - Quickly generated, less effective tests may be better than slowly generated, more effective tests