Badger: Complexity Analysis with Fuzzing and Symbolic Execution

Yannic Noller
Rody Kersten
Corina S. Pasareanu
Complexity Analysis

discover vulnerabilities related to worst-case time/space complexity, e.g., Denial-of-Service

```
0   public void sort (int[] a) {
1     int N = a.length;
2     for (int i = 1; i < N; i++) {
3       int j = i - 1;
4       int x = a[i];
5       while ((j >= 0) && (a[j] > x)) {
6         a[j + 1] = a[j];
7         j--;
8       }
9       a[j + 1] = x;
10   }
```

Insertion Sort

find worst-case input:
automated + fast + concrete

- worst-case complexity: \(O(n^2)\)
- e.g. \(a=[8, 7, 6]\) (\(n=3\))
Our Contributions

• combine **fuzzing** and **symbolic execution** to find algorithmic complexity vulnerabilities

• Badger, a framework for analysis of Java applications

• analysis parameterized by a cost metric

• handling of user-defined cost
Badger

Increased coverage or increased cost

**Problem**

KelinciWCA (based on AFL)

fuzzer and symbolic execution run in parallel

**Solution**

exchange interesting inputs

symbolic execution

based on Symbolic PathFinder (SPF)

**Example**

**Evaluation**

**Related**

**Summary**
KelinciWCA

- based on AFL, extends Kelinci [Kersten2017]
- mutation-based greybox fuzzing
- cost-guided fuzzer: coverage + cost
- cost metrics: timing / memory / user-defined
- maintain current highscore
SymExe with SPF

import inputs

fuzzer

export inputs

interesting input

SymExe

concolic execution

includes

worst-case analysis

Trie Extension / Input Assessment

most promising node

Exploration

trie-guided symbolic execution

bounded symbolic execution

model generation

input generation

Input Generation

model generation

input generation
Example

public void sort (int[] a) {
    int N = a.length;
    for (int i = 1; i < N; i++) {
        int j = i - 1;
        int x = a[i];
        while ((j >= 0) && (a[j] > x)) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = x;
    }
}

Insertion Sort

Trie extension with initial input. The most promising node get selected.

initial input
a=[37, 42, 48]

- Trie
  - Trie extension
    - Trie extension with initial input. The most promising node get selected.
Example

```java
public void sort (int[] a) {
    int N = a.length;
    for (int i = 1; i < N; i++) {
        int j = i - 1;
        int x = a[i];
        while ((j >= 0) && (a[j] > x)) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = x;
    }
}
```

Insertion Sort

Exploration and input generation.

\[ pc = \text{sym}_0 \leq \text{sym}_1 \land \text{sym}_1 > \text{sym}_2 \]

new input
a=[0, 1, 0]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>Example</th>
<th>Evaluation</th>
<th>Related</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:yannic.noller@hu-berlin.de">yannic.noller@hu-berlin.de</a></td>
<td>International Symposium on Software Testing and Analysis (ISSTA) 2018</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Example**

```java
public void sort (int[] a) {
    int N = a.length;
    for (int i = 1; i < N; i++) {
        int j = i - 1;
        int x = a[i];
        while ((j >= 0) && (a[j] > x)) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = x;
    }
}
```

**Insertion Sort**

**Assessment** of new input and **extension** of the trie. New **most promising** node gets selected.

new input `a=[0, 1, 0]`

```
id=0
ROOT
score=8.5
```

```
id=1
line=5
choice=0
score=8.5
```

```
id=2
line=5
choice=0
score=7.0
```

```
id=3
line=5
choice=1
score=10
```

```
id=4
line=5
choice=0
score=10
```
Research Questions

RQ1: Since Badger combines fuzzing and symbolic execution, is it better than each part on their own in terms of:
(a) Quality of worst-case, and
(b) Speed?

RQ2: Is KelinciWCA better than Kelinci in terms of:
(a) Quality of worst-case, and
(b) Speed?

RQ3: Can Badger reveal worst-case vulnerabilities?
## Experiments

<table>
<thead>
<tr>
<th>ID</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insertion Sort</td>
</tr>
<tr>
<td>2</td>
<td>Quicksort</td>
</tr>
<tr>
<td>3a</td>
<td>Regular Expression (fixed input)</td>
</tr>
<tr>
<td>3b</td>
<td>Regular Expression (fixed regex)</td>
</tr>
<tr>
<td>4</td>
<td>Hash Table</td>
</tr>
<tr>
<td>5</td>
<td>Compression</td>
</tr>
<tr>
<td>6</td>
<td>Image Processor</td>
</tr>
<tr>
<td>7</td>
<td>Smart Contract</td>
</tr>
</tbody>
</table>

we report the average values (our full data set is available online)

we report the average values

each experiment for 5 hours and 5 times
Insertion Sort (N=64)

- Badger after 20min: 9305
- KelinciWCA 9305 after 2.85 hours

Initial input score: 509
no significant difference between Badger and KelinciWCA

Quicksort (N=64)

<table>
<thead>
<tr>
<th>time (minutes)</th>
<th>1</th>
<th>21</th>
<th>41</th>
<th>61</th>
<th>81</th>
<th>101</th>
<th>121</th>
<th>141</th>
<th>161</th>
<th>181</th>
<th>201</th>
<th>221</th>
<th>241</th>
<th>261</th>
<th>281</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>costs (# jumps)</td>
<td>0</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>500</td>
<td>600</td>
<td>700</td>
<td>800</td>
<td>900</td>
<td>1000</td>
<td>1100</td>
<td>1200</td>
<td>1300</td>
<td>1400</td>
<td>1500</td>
</tr>
</tbody>
</table>

- Kelinci
- KelinciWCA
- SymExe
- Badger

Initial input score: 2829

Summary

3719 1.31x
3683 1.30x
3161
2970
Image Processor (2x2 JPEG)

- Initial input score: 8712

- Time (minutes): 1, 21, 41, 61, 81, 101, 121, 141, 161, 181, 201, 221, 241, 261, 281, 300

- Costs (# jumps)
  - Kelinci: 349,438 (40.11x)
  - KelinciWCA: 291,384
  - SymExe: 193,730 (22.24x)
  - Badger: 188,719

Legend:
- Blue: Kelinci
- Green: KelinciWCA
- Orange: SymExe
- Red: Badger

Related:
- Problem
- Solution
- Example
- Evaluation
- Summary
Existing Solutions

- **Fuzzing**
  e.g. SlowFuzz [Petsios2017]

- **Symbolic Execution**
  e.g. WISE [Burnim2009] , SPF-WCA [Luckow2017]

- **Fuzzing + Symbolic Execution**
  e.g. Driller [Stephens2016]
Badger: Complexity Analysis with Fuzzing and Symbolic Execution

**Complexity Analysis**

discover vulnerabilities related to worst-case time/space complexity, e.g., Denial-of-Service

```java
public void sort(int[] a) {
    int N = a.length;
    for (int i = 1; i < N; i++) {
        int j = i - 1;
        int x = a[i];
        while (j >= 0 && a[j] > x) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = x;
    }
}
```

- find worst-case input: automated + fast + concrete
  - worst-case complexity: \( O(n^2) \)
  - e.g. \( a=\{8, 7, 6\} \) (\( n=3 \))

**SymEx with SPF**

`SymEx` with `SPF`

**KelinciWCA**

- based on `Kelinci` [Karstens2017]
- mutation-based greybox fuzzing
- cost-guided fuzzier: coverage + cost
- cost metrics: timing / memory / user-defined
- maintain current highscore

- `git clone https://github.com/isstac/badger.git`

yannic.noller@hu-berlin.de

International Symposium on Software Testing and Analysis (ISSTA) 2018
References


