Evaluating and Improving Fault Localization
## Debugging is expensive

Your program has a bug. What do you do?

- Reproduce it
- Locate it
- Fix it

Focus of this talk
Fault localization as a black box

```
c = foo;
u = bar();
while (c < u)
  c = c.baz();
return c;
```

<table>
<thead>
<tr>
<th>Line ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3) c = foo;</td>
</tr>
<tr>
<td>(1) u = bar();</td>
</tr>
<tr>
<td>(4) while (c &lt; u)</td>
</tr>
<tr>
<td>(2) c = c.baz();</td>
</tr>
<tr>
<td>(5) return c;</td>
</tr>
</tbody>
</table>
Agenda

- **Spectrum-based** and **mutant-based** fault localization
- **Evaluating** fault localization techniques
- **Fault provenance**: are artificial faults good proxies for real faults?
  - ➢ No!
  - ➢ Why not?
  - ➢ **What matters** on real faults, then?
  - ➢ Doing better
Let’s design a FL technique!

```java
if (unflushedValues > 0) {
    if (index >= 0 && !this.allowDuplicateXValues) {
        XYDataItem existing = (XYDataItem) this.data.get(index);
        try {
            overwritten = (XYDataItem) existing.clone();
        } catch (CloneNotSupportedException e) {
            throw new SeriesException("Couldn't clone XYDataItem!");
        }
        existing.setY(y);
    }
    ...
}
```

More ●s ⇒ more suspicious
More ●●s ⇒ less suspicious
Let’s design a FL technique!

For each statement

<table>
<thead>
<tr>
<th>Line#</th>
<th>Susp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

weighting factors

Line#

| 7     |
| 6     |
| 2     |
| ...   |
There are many variants on spectrum-based FL:

**Ochiai**\(^1\)
\[
S(s) = \frac{\text{failed}(s)}{\sqrt{\text{totalfailed} \cdot (\text{failed}(s) + \text{passed}(s))}}
\]

**Tarantula**\(^2\)
\[
S(s) = \frac{\frac{\text{failed}(s)}{\text{totalfailed}}}{\frac{\text{failed}(s)}{\text{totalfailed}} + \frac{\text{passed}(s)}{\text{totalpassed}}}
\]

**D**\(^*\)\(^3\)
\[
S(s) = \frac{\text{failed}(s)^*}{\text{passed}(s) + (\text{totalfailed} - \text{failed}(s))}
\]

Another approach to FL: “mutation-based”

```python
def f(arg):
    if arg not in cache:
        return cache[arg]
    ...
    cache[arg] = (start+stop)/2
    cache.sync()
    return (start+stop+1)/2
```

```python
def f(arg):
    if arg in cache:
        return cache[arg]
    ...
    cache[arg] = (start-stop)/2
    cache.sync()
    return (start+stop+1)/2
```

```python
def f(arg):
    if arg in cache:
        return cache[arg]
    ...
    cache[arg] = (start+stop)/2
    cache.sync()
    return (start+stop+1)/2
```

```python
def f(arg):
    if arg in cache:
        return cache[arg]
    ...
    cache[arg] = start+stop/2
    cache.sync()
    return (start+stop+1)/2
```

More $\blacktriangle$ ⇒ more suspicious

More $\blacktriangle$ ⇒ less suspicious

```python
def f(arg):
    if arg in cache:
        return cache[arg]
    ...
    cache[arg] = (start+stop)/2
    cache.sync()
    return (start+stop+1)/2
```

```python
def f(arg):
    if arg in cache:
        return cache[arg]
    ...
    cache[arg] = (start+stop)/2
    cache.sync()
    return (start+stop+1)/2
```

```python
def f(arg):
    if arg in cache:
        return cache[arg]
    ...
    cache[arg] = (start+stop)/2
    cache.sync()
    return (start+stop+1)/2
```
Another approach to FL: “mutation-based”

For each mutant

<table>
<thead>
<tr>
<th>Mut#</th>
<th>Susp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.6</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line#</th>
<th>Susp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

weighting factors

λ

collect

sort

Line#

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
There are few variants on mutation-based FL:

**Metallaxis**\(^1\)  \[ S(s) = \max_{m \in \text{mut}(s)} \frac{\text{failed}(m)}{\sqrt{\text{total failed} \cdot (\text{failed}(m) + \text{passed}(m))}} \]

**MUSE** \(^2\)  \[ S(s) \propto \text{avg}_{m \in \text{mut}(s)} \left[ \text{failed}(m) - \frac{\text{total failed}}{\text{total passed}} \text{passed}(m) \right] \]

---

\(^1\) M. Papadakis and Y. Le Traon. Metallaxis-FL: Mutation-based fault localization.
\(^2\) S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization.
How do you tell whether a FL technique is good?

Program + Tests + Defect knowledge

Find defect in ranking

(3) \( u = \text{foo}; \)
(1) \( c = \text{bar}(); \)
(4) while \( (c < u) \)
(2) \( c = c.\text{baz}(); \)
...

Score (smaller = better)

4/90 avg 0.04

Blue technique is the best FL technique
How do you get defect information for evaluation?

- **Artificial faults** (mutants)
  - Easy to make lots of faults
  - Easy to reason about
  - Not necessarily realistic

- **Real faults** (from issue trackers)
  - Hard to collect; fewer faults
  - Diverse and complicated
  - Reflect real-world use cases

---

Are artificial faults good substitutes for real faults?

A FL technique that does well on artificial faults may do badly on real ones! We:

- generated many artificial faults by mutating fixed statements
- repeated previous comparisons
  - on artificial faults
  - on real faults

Do the same techniques win on both? **No!**

<table>
<thead>
<tr>
<th>Previous (Winner &gt; loser)</th>
<th>Artificial Replicated?</th>
<th>Ours (Winner &gt; loser)</th>
<th>Real Replicated?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ochiai &gt; Tarantula</td>
<td>yes</td>
<td>yes</td>
<td>(insig.)</td>
</tr>
<tr>
<td>Burinel &gt; Ochiai</td>
<td>no</td>
<td>no</td>
<td>(insig.)</td>
</tr>
<tr>
<td>Burinel &gt; Tarantula</td>
<td>yes</td>
<td>yes</td>
<td>(insig.)</td>
</tr>
<tr>
<td>Op2 &gt; Ochiai</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Op2 &gt; Tarantula</td>
<td>yes</td>
<td>yes</td>
<td>(insig.)</td>
</tr>
<tr>
<td>DStar &gt; Ochiai</td>
<td>yes</td>
<td>yes</td>
<td>(insig.)</td>
</tr>
<tr>
<td>DStar &gt; Tarantula</td>
<td>yes</td>
<td>yes</td>
<td>(insig.)</td>
</tr>
<tr>
<td>Ochiai &gt; Jaccard</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Jaccard &gt; Tarantula</td>
<td>yes</td>
<td>yes</td>
<td>(insig.)</td>
</tr>
<tr>
<td>Burinel &gt; Jaccard</td>
<td>no</td>
<td>no</td>
<td>(insig.)</td>
</tr>
<tr>
<td>Op2 &gt; Jaccard</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Metallaxis &gt; Ochiai</td>
<td>(insig.)</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>MUSE &gt; Op2</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>MUSE &gt; Tarantula</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>MUSE &gt; Jaccard</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

SBFL-SBFL MBFL-SBFL
Are artificial faults good substitutes for real faults?

(No!)

Artificial faults vs. Real faults

Artificial faults

Real faults

better
Why the difference?

- Real faults often involve unmutable lines (e.g. `break`, `return`)

- MBFL does very well on “reversible” artificial faults

\[
\text{sum} = \text{sum} + x \quad \text{create fault} \quad \text{sum} = \text{sum} - x \quad \text{mutate} \quad \text{sum} = \text{sum} + x
\]
Common structure

For each mutant

\[ \lambda \]

weighting factors

<table>
<thead>
<tr>
<th>Line#</th>
<th>Susp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line#</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>…</td>
</tr>
</tbody>
</table>
Common structure

For each mutant

<table>
<thead>
<tr>
<th>Mut#</th>
<th>Susp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0.6</td>
</tr>
<tr>
<td>3</td>
<td>0.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

λ

weighting factors

Line#

<table>
<thead>
<tr>
<th>Line#</th>
<th>Susp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

collect

sort

<table>
<thead>
<tr>
<th>Mut#</th>
<th>Line#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Common structure

For each element

\[ \lambda \]

For each element

<table>
<thead>
<tr>
<th>Elem#</th>
<th>Susp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Collect

<table>
<thead>
<tr>
<th>Line#</th>
<th>Susp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Sort

<table>
<thead>
<tr>
<th>Line#</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Identity for SBFL

Weighting factors
Common structure

Technique Space

- SBFL
- MBFL: what counts as a failing test “detecting” a mutant?
  - AnError(1) → AnError(2)
  - ...
  - AnError → OtherError
  - AnError → pass

Important

Unimportant

weighting factors

λ

collect
New techniques

- SBFL and MBFL both have outliers… but in different cases!
- Average them together!
- Other (smaller) improvements:
  - Make MBFL incorporate mutant coverage information
  - Increase resolution of SBFL by using mutants
Summary

def f(arg):
    if arg not in cache:
        return cache[arg]
    ...
    cache[arg] = (start+stop)/2
    cache.sync()
    return (start+stop+1)/2

if (unflushed
try {
    overwri
} catch (Cl
    throw n
    existing,
} ...

Artificial Replicated? | Real Replicated?
---|---
yes | (inig.)
no | no
yes | (inig.)
no | no
yes | (inig.)
no | no
yes | (inig.)
no | no
yes | (inig.)
no | no
no | no
no | no
no | no

For each element:

1. Select elements based on suspiciousness.
2. Collect the identified elements.
3. Sort the collected elements.

Evaluation

What matters?

Summary

Motivation | Black-box model | Approaches | Evaluation | Artificial vs. real faults | Failure modes | Design space | New techniques | Summary
--- | --- | --- | --- | --- | --- | --- | --- | ---
Spectrum | Mutant | ...Evaluation | Replication | What matters?
Future work

- Are artificial faults still bad proxies for real faults with other families of FL techniques?

- Could generated test suites make artificial faults better proxies?

- Do some mutation operators produce better artificial faults than others?
Motivation

Black-box model

Summary

Approaches

Spectrum

Mutant

Artificial vs. real faults

Replication

New techniques

Design space

Failure modes

Evaluation

What matters?

...Evaluation
Alternative metric: top-$n$

- “Average percent through the program until first faulty statement” might not be the best metric.
- Alternative: “probability a faulty statement is in the $n$ most suspicious.”
- $n=5$ for debugging, $n=200$ for program repair tools \[1\]

<table>
<thead>
<tr>
<th>Technique</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-200</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCBFL-hybrid-avg</td>
<td>36%</td>
<td>45%</td>
<td>85%</td>
</tr>
<tr>
<td>MRSBFL-hybrid-avg</td>
<td>31%</td>
<td>41%</td>
<td>86%</td>
</tr>
<tr>
<td>DStar</td>
<td>30%</td>
<td>39%</td>
<td>82%</td>
</tr>
<tr>
<td>Ochiai</td>
<td>30%</td>
<td>39%</td>
<td>82%</td>
</tr>
<tr>
<td>Jaccard</td>
<td>29%</td>
<td>39%</td>
<td>81%</td>
</tr>
<tr>
<td>Metallaxis</td>
<td>29%</td>
<td>39%</td>
<td>77%</td>
</tr>
<tr>
<td>Barinel</td>
<td>27%</td>
<td>38%</td>
<td>80%</td>
</tr>
<tr>
<td>Tarantula</td>
<td>27%</td>
<td>37%</td>
<td>80%</td>
</tr>
<tr>
<td>Op2</td>
<td>27%</td>
<td>37%</td>
<td>80%</td>
</tr>
<tr>
<td>MUSE</td>
<td>19%</td>
<td>23%</td>
<td>45%</td>
</tr>
</tbody>
</table>