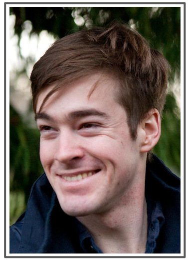
# Evaluating and Improving Fault Localization



Spencer Pearson

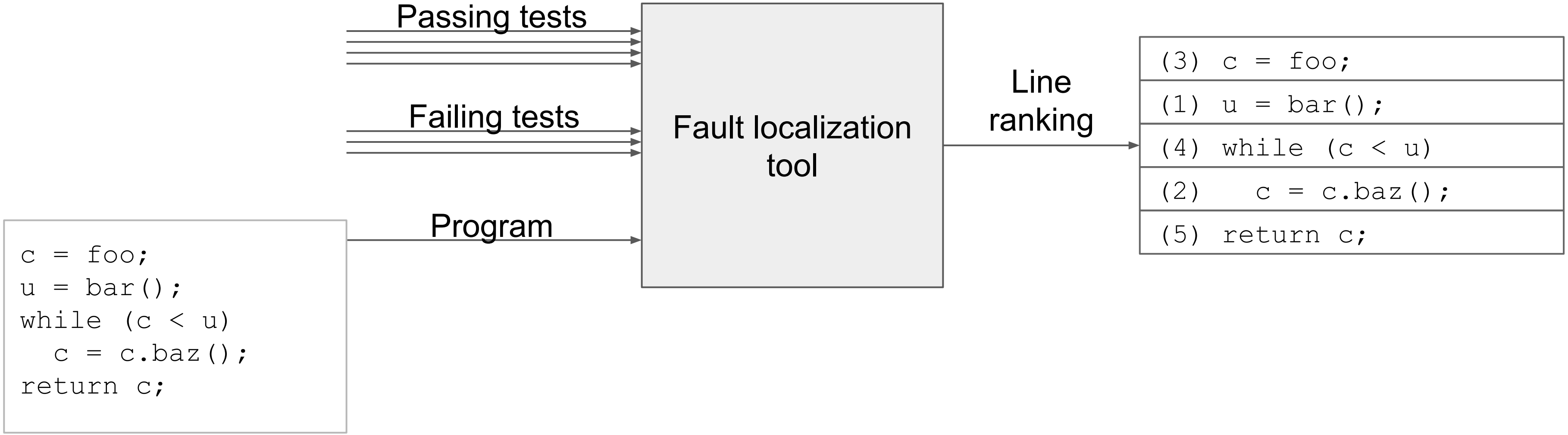
Michael Ernst

# Debugging is expensive

Your program has a bug. What do you do?

- Reproduce it
- Locate it ⇦ Focus of this talk
- Fix it

# Fault localization as a black box



# Agenda

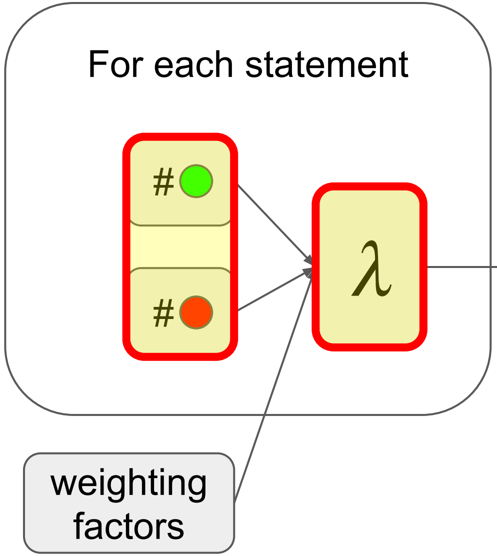
- **Spectrum-based** and **mutant-based** fault localization
- **Evaluating** fault localization techniques
- **Fault provenance:** are artificial faults good proxies for real faults?
  - ➢ **No!**
  - ➢ **Why not?**
  - ➢ **What matters** on real faults, then?
  - ➢ **Doing better**

# Let’s design a FL technique!

```
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!




| Line# | Susp. |
|---|---|
| 1 | 0.2 |
| 2 | 0.5 |
| 3 | 0.0 |
| ... | ... |

sort

| Line# |
|---|
| 7 |
| 6 |
| 2 |
| ... |

# There are many variants on spectrum-based FL:

**Ochiai[1]**

$$S(s) = \frac{failed(s)}{\sqrt{totalfailed \cdot (failed(s) + passed(s))}}$$

**Tarantula[2]**

$$S(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

**D*[3]**

$$S(s) = \frac{failed(s)^*}{passed(s) + (totalfailed - failed(s))}$$

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization.
[2] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization.
[3] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar method for effective software fault localization.

# Another approach to FL: “mutation-based”

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

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

```
def f(arg):
  if arg in cache:
    return cache[arg]
  ...
  cache[arg] = (start+stop)/2
  cache.sync()
  return (start+stop-0)/2
```

```
def f(arg):
  if None in cach
    return cache
  ...
  cache[arg] = (s
  cache.sync()
  return (start+stop+1)/2
```

```
  return (start+stop+1)/2
```

```
:
  n cache:
  cache[arg]

  g] = (start+stop)/2
  nc()
  return (start/stop+1)/2
```

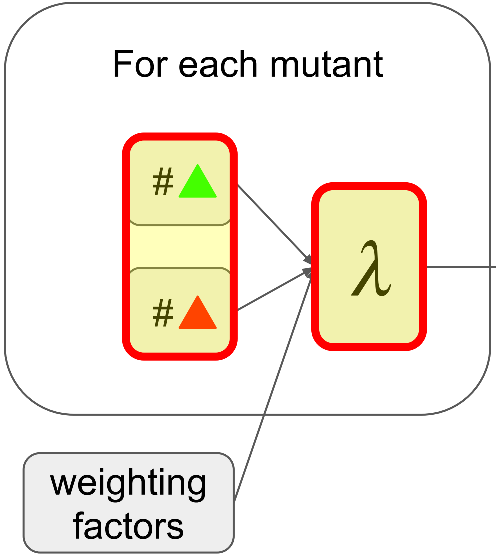
More ⇒ more suspicious
More ⇒ less suspicious

```
def f(arg):
  if arg in None:
    return cache[arg]
  ...
```

```
def f(arg):
  if arg in cache:
    return cache[arg]
  ...
```

```
def f(arg):
  if arg in cache:
    return cache[arg]
  ...
```

# Another approach to FL: “mutation-based”




| Mut# | Susp. |
|---|---|
| 1 | 0.1 |
| 2 | 0.6 |
| 3 | 0.1 |
| ... | ... |

collect

| Line# | Susp. |
|---|---|
| 1 | 0.2 |
| 2 | 0.5 |
| 3 | 0.0 |
| ... | ... |

sort

| Line# |
|---|
| 7 |
| 6 |
| 2 |
| ... |

# There are few variants on mutation-based FL:

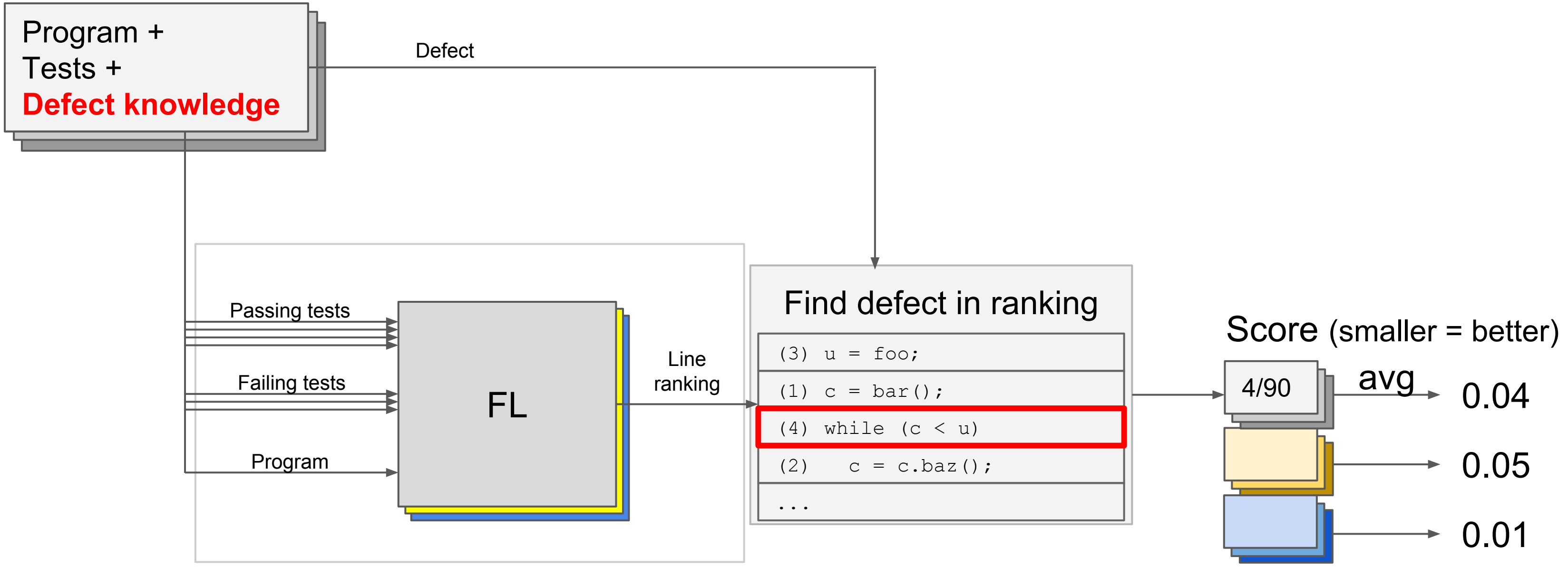
**Metallaxis[1]** $$S(s) = \max_{m \in mut(s)} \frac{failed(m)}{\sqrt{totalfailed \cdot (failed(m) + passed(m))}}$$

---

**MUSE [2]** $$S(s) \propto \underbrace{\operatorname*{avg}_{m \in mut(s)}}_{\text{collect}} \underbrace{\left[ failed(m) - \frac{totalfailed}{totalpassed} passed(m) \right]}_{\lambda}$$

[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?




Blue technique is the best FL technique

# How do you get defect information for evaluation?

Program +
Tests +
**Defect knowledge**

```
int x;
int sum;
int iters;
sum = xs[0];
...
```

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

Used by previous research

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

Provided by the recent project *Defects4J [1]*

[1] Just et al. "Defects4J: A database of existing faults to enable controlled testing studies for Java programs." ISSTA 2014 Proceedings. ACM, 2014.

# 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?

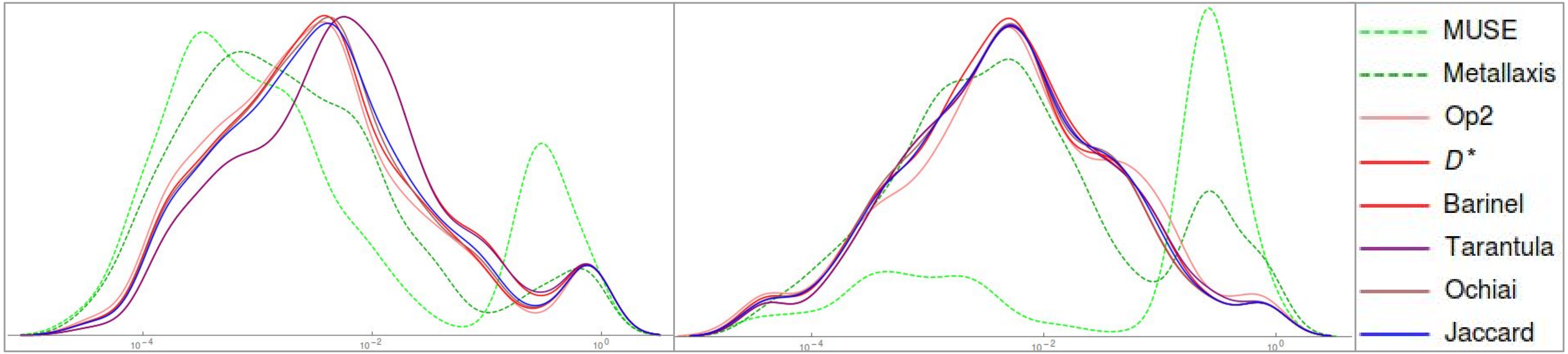
**No!**

| **Previous** (Winner > loser) | **Ours** **Artificial** Replicated? | **Real** Replicated? | |
|---|---|---|---|
| Ochiai > Tarantula | yes | *(insig.)* | SBFL-SBFL |
| Barinel > Ochiai | no | *(insig.)* | |
| Barinel > Tarantula | yes | *(insig.)* | |
| Op2 > Ochiai | yes | no | |
| Op2 > Tarantula | yes | *(insig.)* | |
| DStar > Ochiai | yes | *(insig.)* | |
| DStar > Tarantula | yes | *(insig.)* | |
| Ochiai > Jaccard | yes | yes | |
| Jaccard > Tarantula | yes | *(insig.)* | |
| Barinel > Jaccard | no | *(insig.)* | |
| Op2 > Jaccard | yes | no | |
| Metallaxis > Ochiai | *(insig.)* | no | MBFL-SBFL |
| MUSE > Op2 | no | **no** | |
| MUSE > Tarantula | no | **no** | |
| MUSE > Jaccard | no | **no** | |

# Are artificial faults good substitutes for real faults?

(No!)

Artificial faults Real faults




better

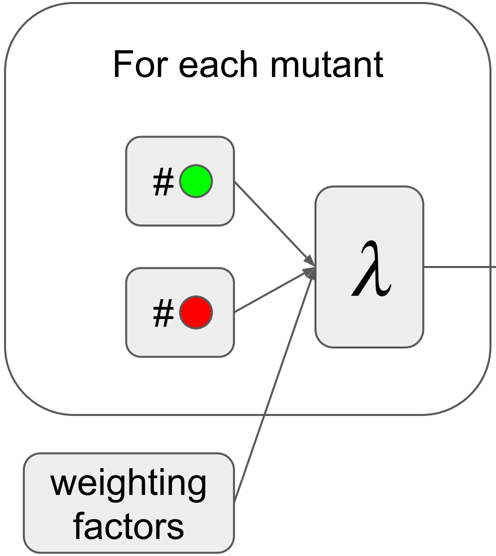
# Why the difference?

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

- MBFL does very well on “reversible” artificial faults

`sum = sum + x` —create fault→ `sum = sum - x` —mutate→ `sum = sum + x`

# Common structure

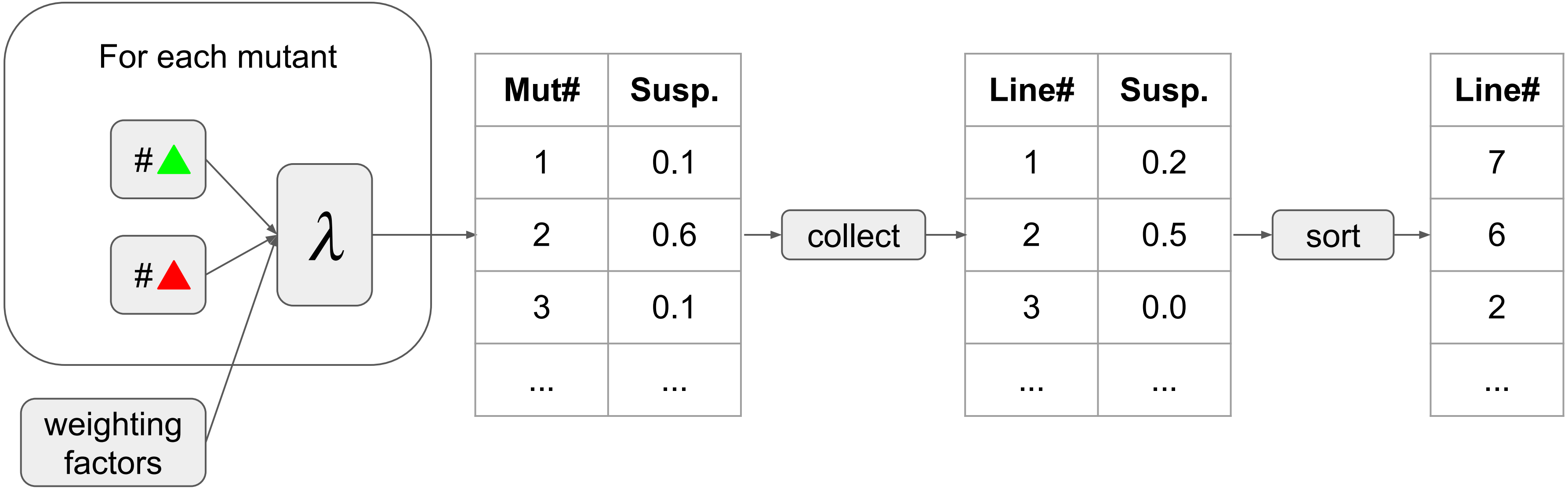



| Line# | Susp. |
|---|---|
| 1 | 0.2 |
| 2 | 0.5 |
| 3 | 0.0 |
| ... | ... |

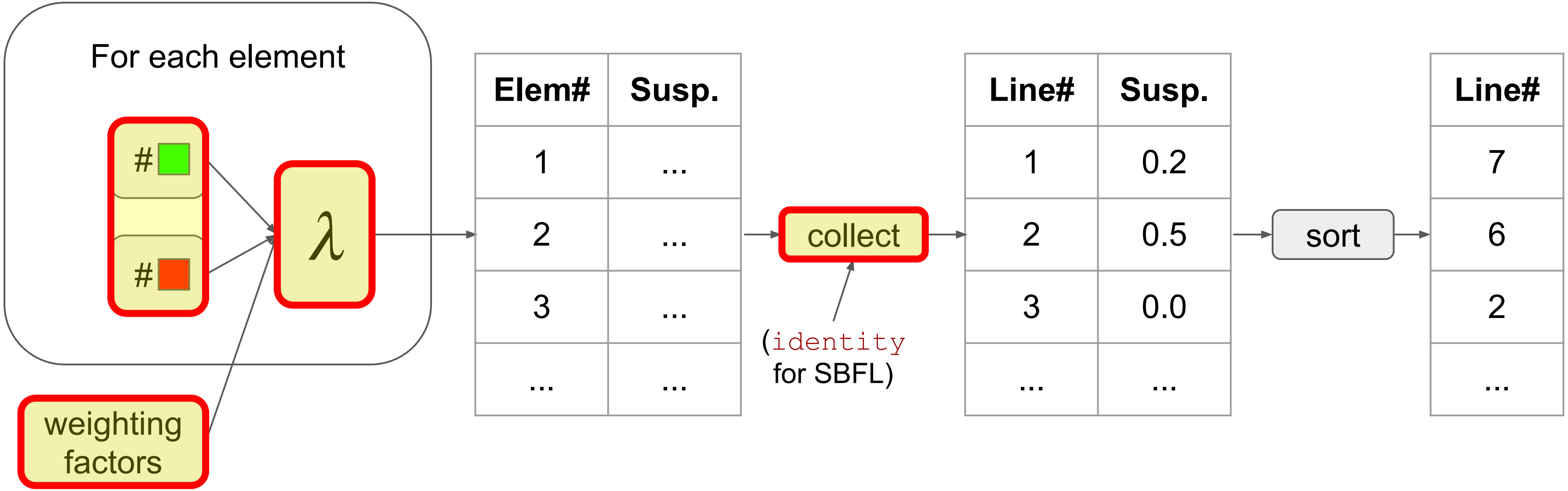
sort

| Line# |
|---|
| 7 |
| 6 |
| 2 |
| ... |

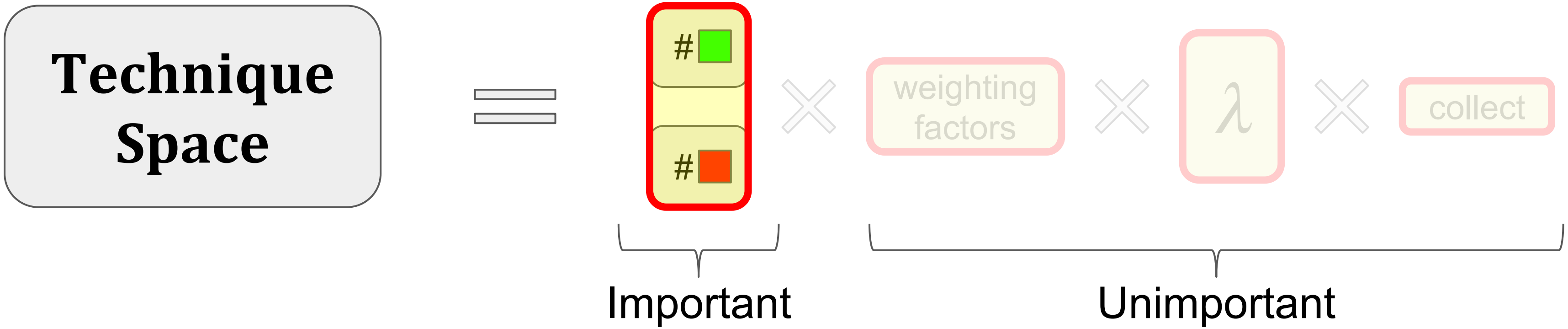
# Common structure



# Common structure



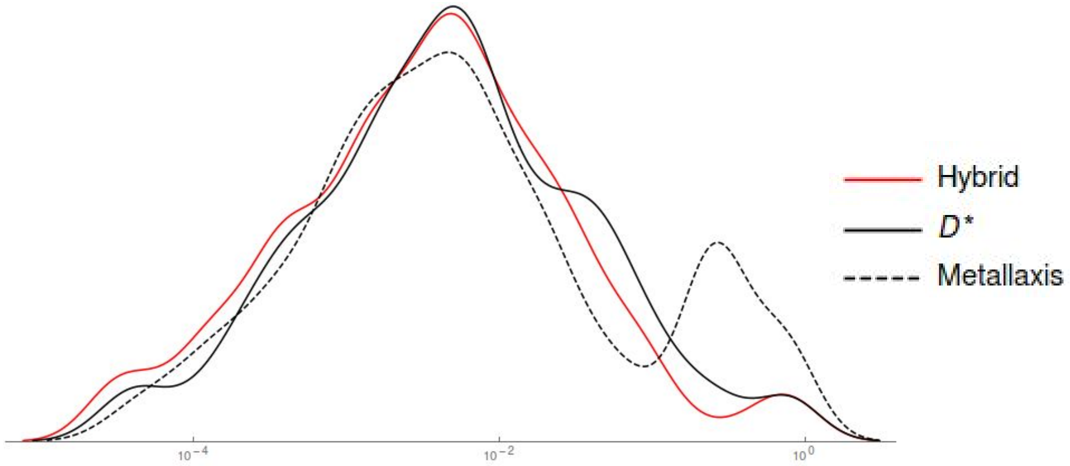
# Common structure



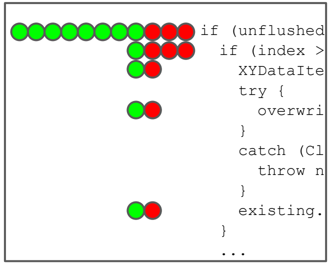

- SBFL
- MBFL: what counts as a failing test "detecting" a mutant?
  - AnError(1)→AnError(2) ✓
  - …
  - AnError→OtherError
  - AnError→pass

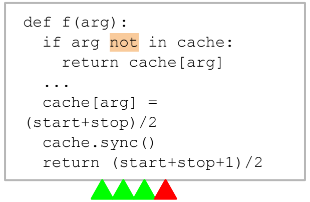
# New techniques

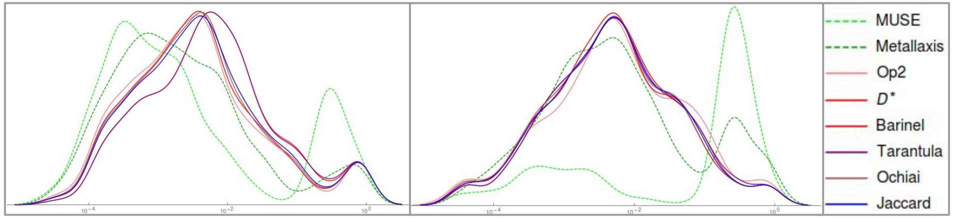
- SBFL and MBFL both have outliers… but in different cases!
- Average them together!
- Other (smaller) improvements:
  - Make MBFL incorporate mutant coverage information
  - Increase resolution of SBFL by using mutants



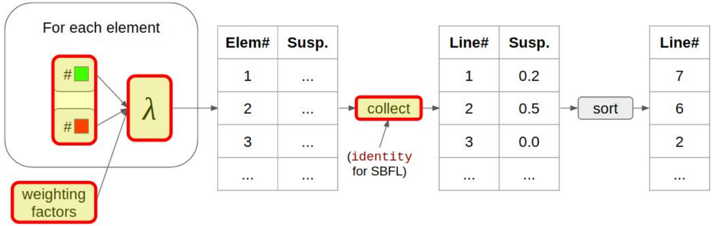
# Summary

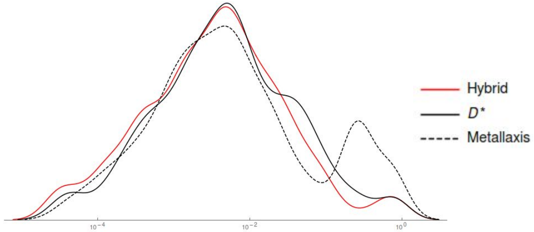



| **Artificial** Replicated? | **Real** Replicated? |
|---|---|
| yes | *(insig.)* |
| no | *(insig.)* |
| yes | *(insig.)* |
| yes | no |
| yes | *(insig.)* |
| yes | *(insig.)* |
| yes | *(insig.)* |
| yes | yes |
| yes | *(insig.)* |
| no | *(insig.)* |
| yes | no |
| *(insig.)* | no |
| no | **no** |
| no | **no** |
| no | **no** |



# 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?

# 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]

| Technique | Top-5 | Top-10 | Top-200 |
|---|---|---|---|
| MCBFL-hybrid-avg | 36% | 45% | 85% |
| MRSBFL-hybrid-avg | 31% | 41% | 86% |
| DStar | 30% | 39% | 82% |
| Ochiai | 30% | 39% | 82% |
| Jaccard | 29% | 39% | 81% |
| Metallaxis | 29% | 39% | 77% |
| Barinel | 27% | 38% | 80% |
| Tarantula | 27% | 37% | 80% |
| Op2 | 27% | 37% | 80% |
| MUSE | 19% | 23% | 45% |

[1] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems.