Statistical fault localization

UW CSE P 504

Today

- Recap: invariants and metamorphic testing
- Automated debugging
  - **Statistical fault localization**
  - Automated patch generation
- Defect prediction

Recap: invariants and metamorphic testing

Kick-starting the discussion

1. What is a program invariant? What guarantees does Daikon provide for its discovered invariants? How is it related to a specification?
2. What is a partial test oracle, a follow-up test input, and a metamorphic relation?
3. How are invariants and metamorphic relations similar and how are they different? (Context: using them as partial test oracles in software testing.)

Post open questions/confusions to the forum.

Recap: Pre/post-conditions and invariants

```
double avgAbs(double[] nums) {
 int n = nums.length;
 double sum = 0;

 int i = 0;
 while (i != n) {
   if(nums[i]>0) {
    sum = sum + nums[i];
   else {
    sum = sum - nums[i];
   }
   i = i + 1;
 }

 return sum / n;
}
```

Entry point

Exit point

Recap: data diversity and metamorphic testing

Context:

- Input i_1 yields output o_1 (“initial input”)
- Expected output for a given input is unknown

Simplest case: related inputs with identical outcomes

- Example: `abs(x) = abs(-x)` (“follow-up input”)
- Generalizing: $p(i_1) = p(R_i(i_1))$
  - The SUT (system under test) p is `abs`
  - The input relation R_i is negation

Recap: data diversity and metamorphic testing

Context:

- Input i_1 yields output o_1 ("initial input")
- Expected output for a given input is unknown

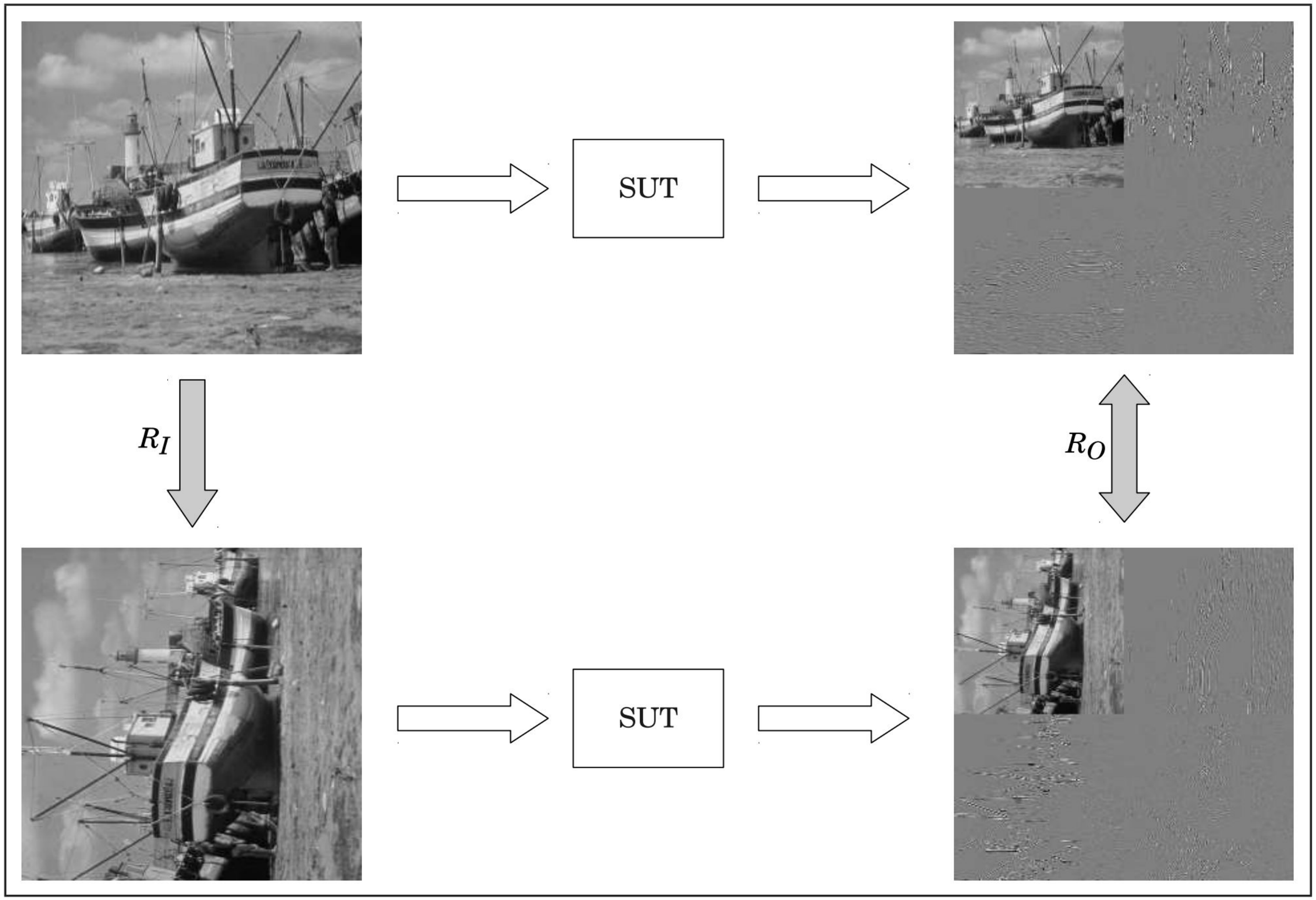
More expressive: related inputs and related outputs

- R_i: $i_1 \Rightarrow i_2$ ("follow-up input")
- R_o: $o_1 \Rightarrow o_2$ ("necessary condition")
- Generalizing: $R_o(\mathrm{p}(i_1), \mathrm{p}(R_i(i_1))) = \text{true}$
- Generalizing: $R(i_1, i_2, o_1, o_2) = \text{true}$
  - Example: R_{abs}(a, b, c, d) = (a = -b) ⇒ (c = d)

Typical metamorphic test case:

```
i1 = random selection...
o1 = p(i1)
i2 = Ri(i1)
o2 = p(i2)
assert Ro(o1, o2)
```

Recap: data diversity and metamorphic testing



How can you localize a defect?

-

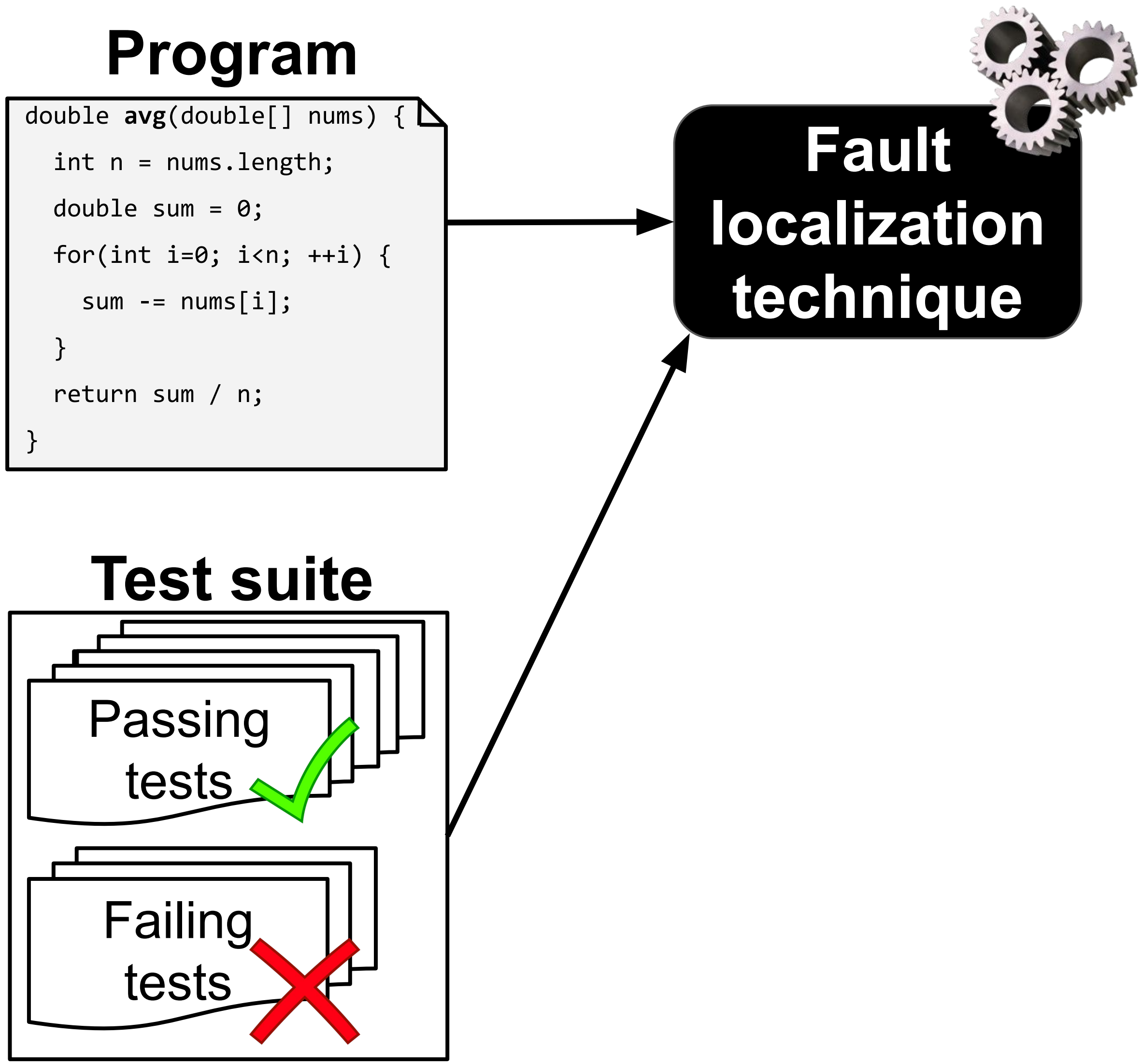
How can you localize a defect?

- Static analysis: linting, bug finding, verification
- Logging:
  - Assert statement (success then failure brackets the defect)
  - Stack trace
  - Logs
  - Bug reports
  - Performance regression
  - Coverage: Statistical fault localization: ranks source code lines
- Compare multiple stack traces/logs/bug reports
- Similarity to previous defects
- Minimized input (e.g., binary search, delta debugging)
- Minimized program
  - Version control history
  - Unit testing
- Differential testing (programs, values; e.g., metamorphic)

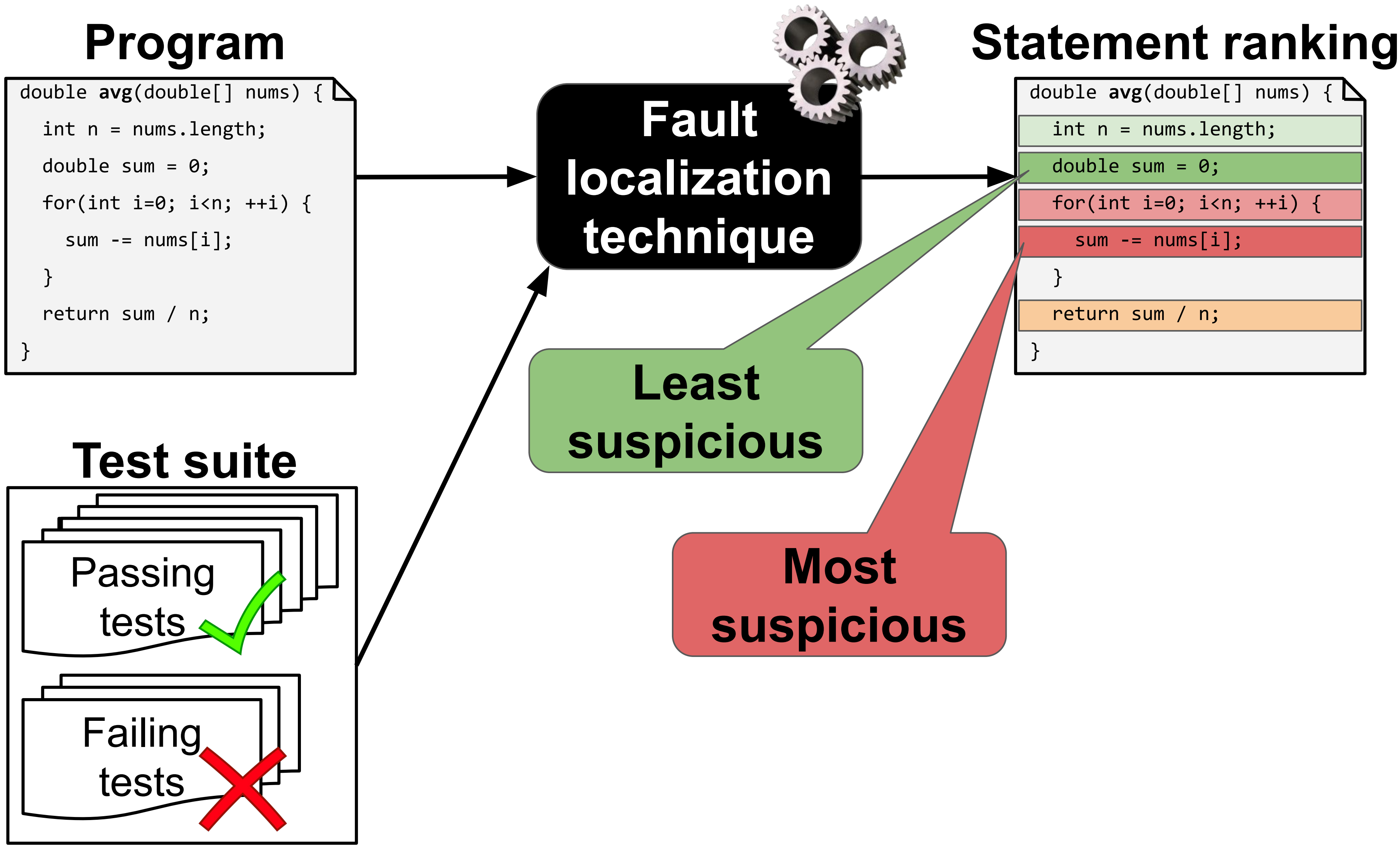
Statistical fault localization

"Fault" is here a synonym for "defect"
(but "fault" also has other meanings)

What is statistical fault localization?



What is statistical fault localization?



Statistical fault localization: count success & failure

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```

Statistical fault localization: count success & failure

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```

- Run all tests
  - t1 passes

Statistical fault localization: count success & failure

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```

- Run all tests
  - t1 passes
  - t2 passes

Statistical fault localization: count success & failure

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```

- Run all tests
  - t1 passes
  - t2 passes
  - t3 passes

Statistical fault localization: count success & failure

Program

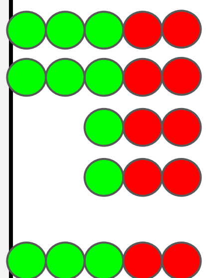
```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```

- Run all tests
  - t1 passes
  - t2 passes
  - t3 passes
  - t4 fails

Statistical fault localization: count success & failure

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```



- Run all tests
  - t1 passes
  - t2 passes
  - t3 passes
  - t4 fails
  - t5 fails

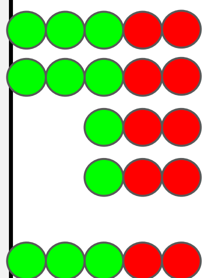
Which lines seem most suspicious?

More ● = more suspicious

Spectrum-based fault localization

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```



Spectrum-based FL (SBFL)

- **Compute** suspiciousness **per statement**
- Given statement *s*, many ways to combine:
  - passed(*s*)
  - failed(*s*)
  - totalpassed
  - totalfailed

Statement **covered** by **failing test**

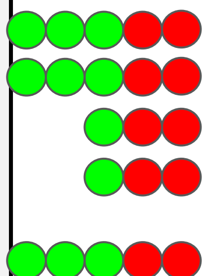
Statement **covered** by **passing test**

More → statement is more suspicious

Spectrum-based fault localization

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```



Spectrum-based FL (SBFL)

- **Compute** suspiciousness **per statement**
- Given statement *s*, many ways to combine:
  - passed(*s*)
  - failed(*s*)
  - totalpassed
  - totalfailed
- Example:

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

Statement **covered** by **failing test**

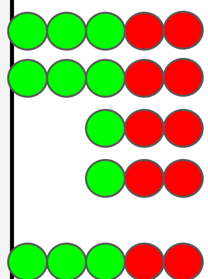
Statement **covered** by **passing test**

More → statement is more suspicious

Spectrum-based fault localization

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```

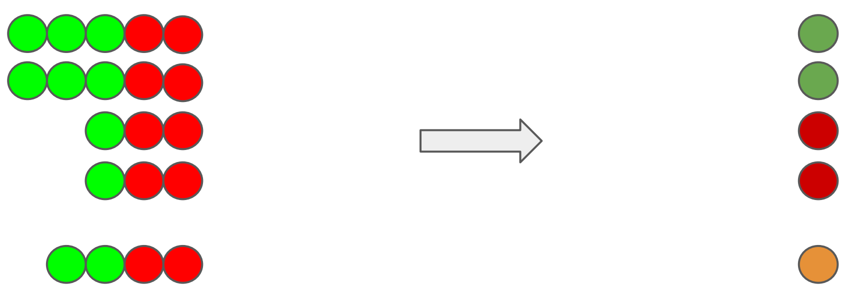


Spectrum-based FL (SBFL)

- **Compute** suspiciousness **per statement**
- Example:

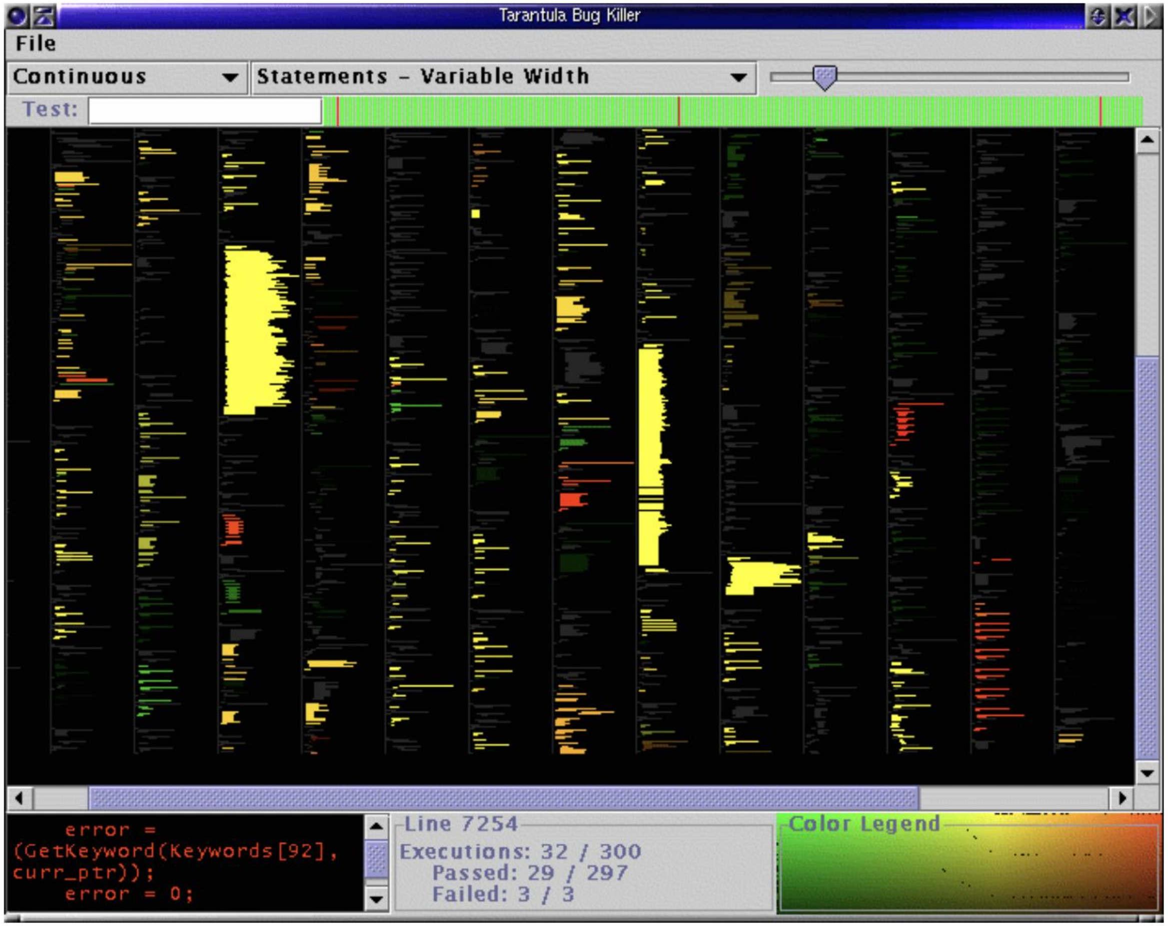
$$S(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

Visualization: the key idea behind Tarantula.



Jones et al., *Visualization of test information to assist fault localization*, ICSE'02

Spectrum-based fault localization

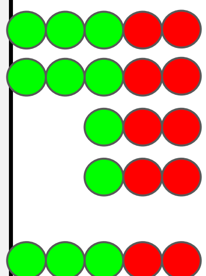


Jones et al., *Visualization of test information to assist fault localization*, ICSE'02

Spectrum-based fault localization

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```



Spectrum-based FL (SBFL)

- **Compute** suspiciousness **per statement**
- Example:

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

Mutation-based fault localization

Program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum -= nums[i];
  }
  return sum / n;
}
```

Mutants

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum += nums[i];
  }
  return sum / n;
}
```

Mutation-based FL (MBFL)

- **Compute** suspiciousness **per mutant**
- **Aggregate** results **per statement**
- Example:

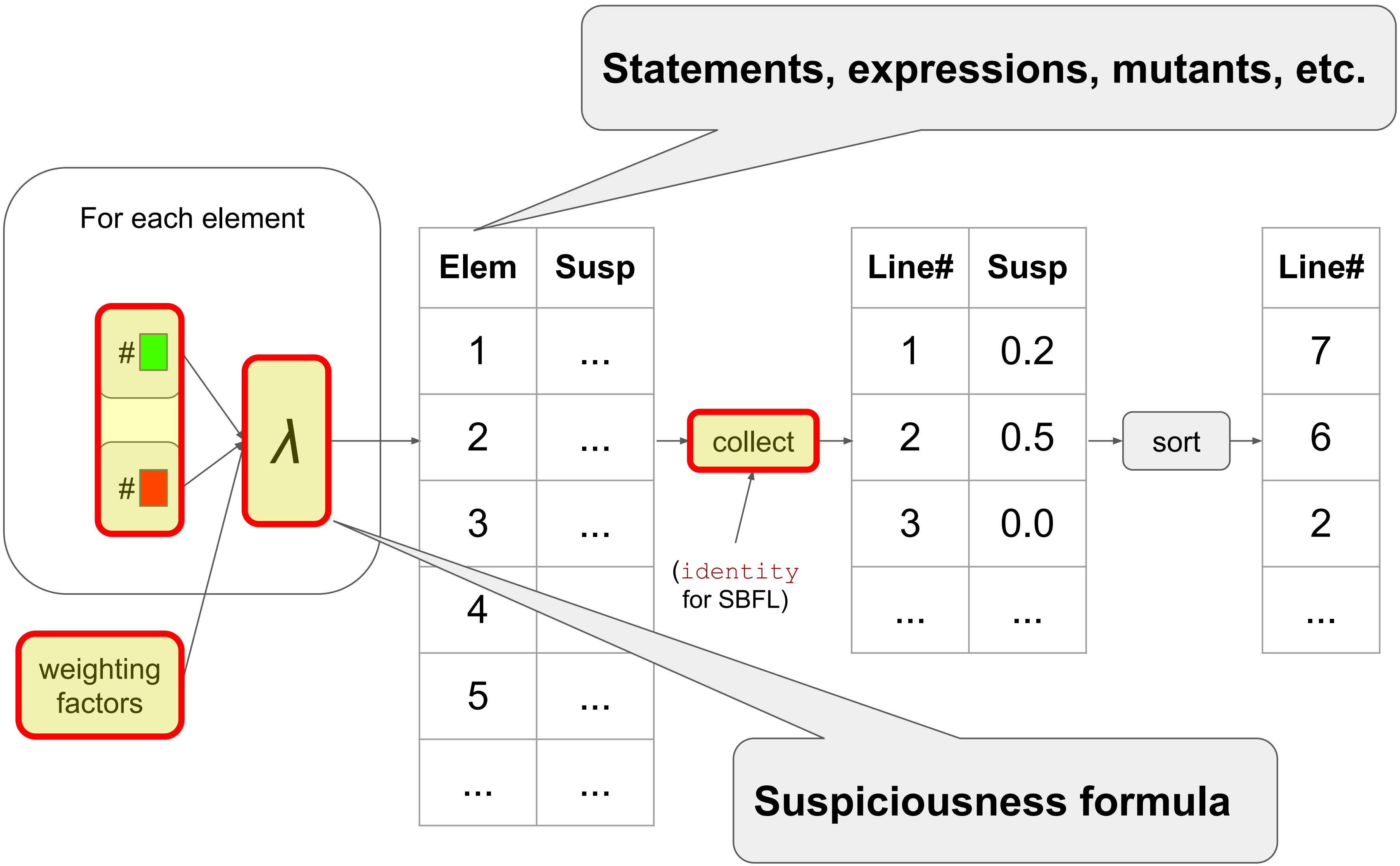
$$S(s) = \max_{m \in mut(s)} \frac{failed(m)}{\sqrt{totalfailed \cdot (failed(m) + passed(m))}}$$

Mutant **affects failing test** outcome

Mutant **breaks passing test**

More → mutant is more suspicious!

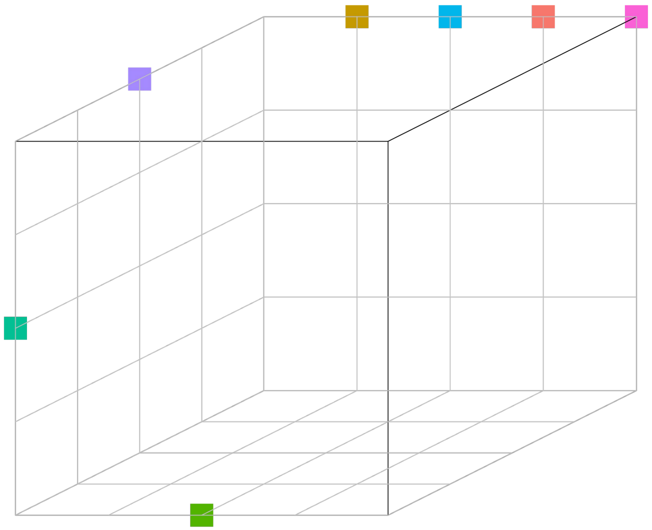
Common structure of SBFL and MBFL



What design decisions matter?

Defined and explored a design space for SBFL and MBFL

- 4 design factors (e.g., formula)

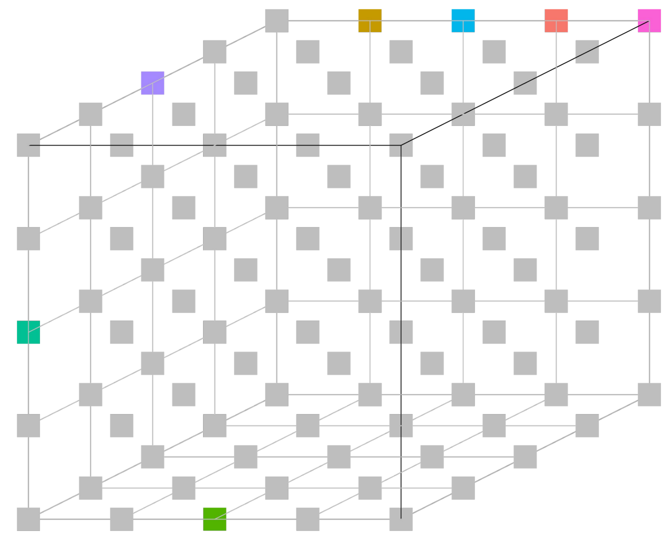


Pearson et al., *Evaluating and Improving Fault Localization*, ICSE'17

What design decisions matter?

Defined and explored a design space for SBFL and MBFL

- 4 design factors (e.g., formula)
- 156 FL techniques

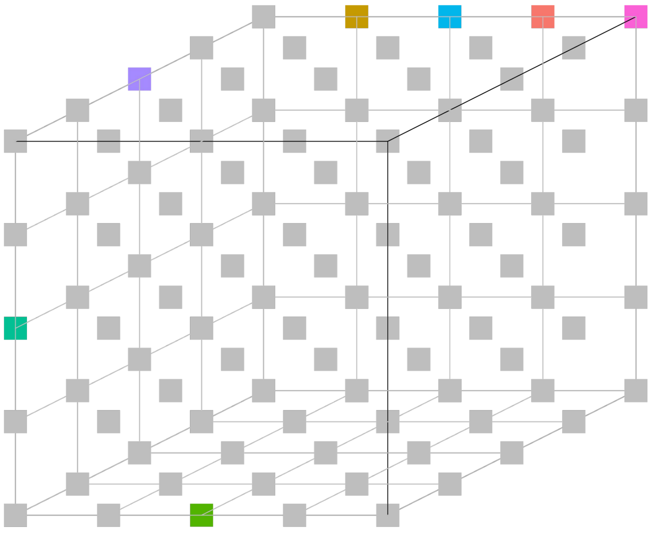


Pearson et al., *Evaluating and Improving Fault Localization*, ICSE’17

What design decisions matter?

Defined and explored a design space for SBFL and MBFL

- 4 design factors (e.g., formula)
- 156 FL techniques



Results

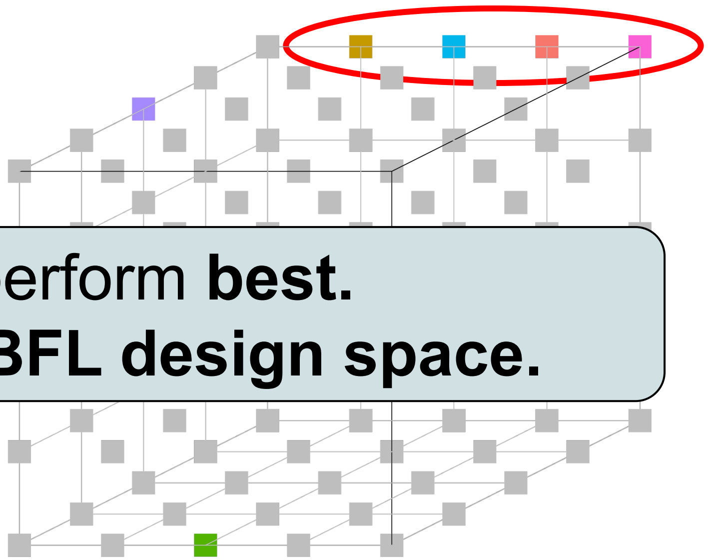
- Most design decisions **don't matter** (in particular for SBFL)
- Definition of test-mutant interaction matters for MBFL
- Barinel, D*, Ochiai, and Tarantula are indistinguishable

Pearson et al., *Evaluating and Improving Fault Localization*, ICSE'17

What design decisions matter?

Defined and explored a design space for SBFL and MBFL

- 4 design factors (e.g., formula)
- 156 FL techniques



Existing **SBFL techniques** perform **best.**
No breakthroughs in the **MBFL/SBFL design space.**

- Most design decisions **don’t matter** (in particular for SBFL)
- Definition of test-mutant interaction matters for MBFL
- Barinel, D*, Ochiai, and Tarantula are indistinguishable

Pearson et al., *Evaluating and Improving Fault Localization*, ICSE’17

Effectiveness of SBFL and MBFL

- Top-10 useful for practitioners[1].
- Top-200 useful for automated patch generation[2].

Technique	**Top-5**	**Top-10**	**Top-200**
Hybrid	36%	45%	85%
DStar (*best SBFL*)	30%	39%	82%
Metallaxis (*best MBFL*)	29%	39%	77%

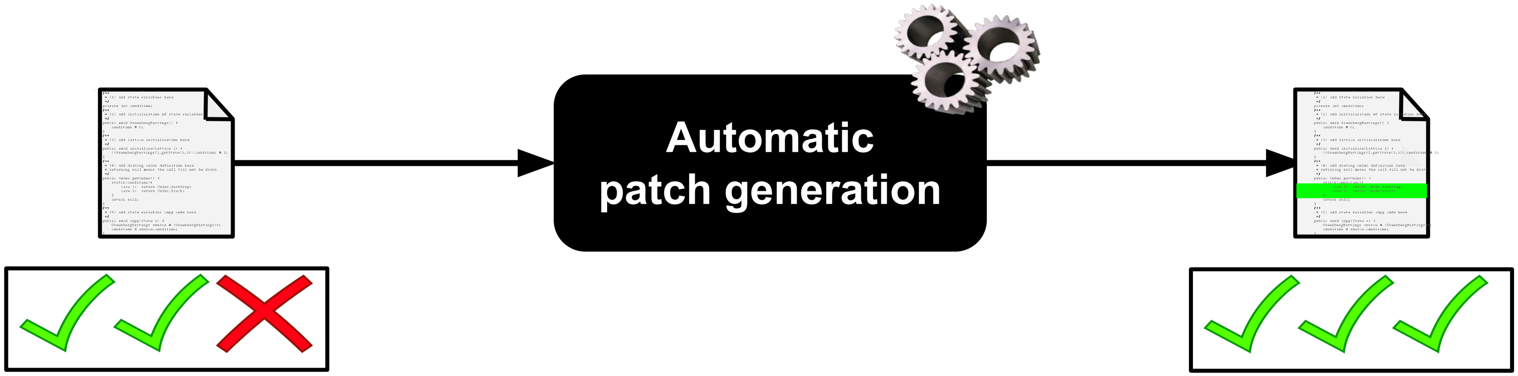
What assumptions underpin these results? Are they realistic?

[1]Kochhar et al., *Practitioners' Expectations on Automated Fault Localization*, ISSTA'16
[2]Long and Rinard, *An analysis of the search spaces for generate and validate patch generation systems*, ICSE'16

Automated patch generation

Automatic patch generation (program repair)

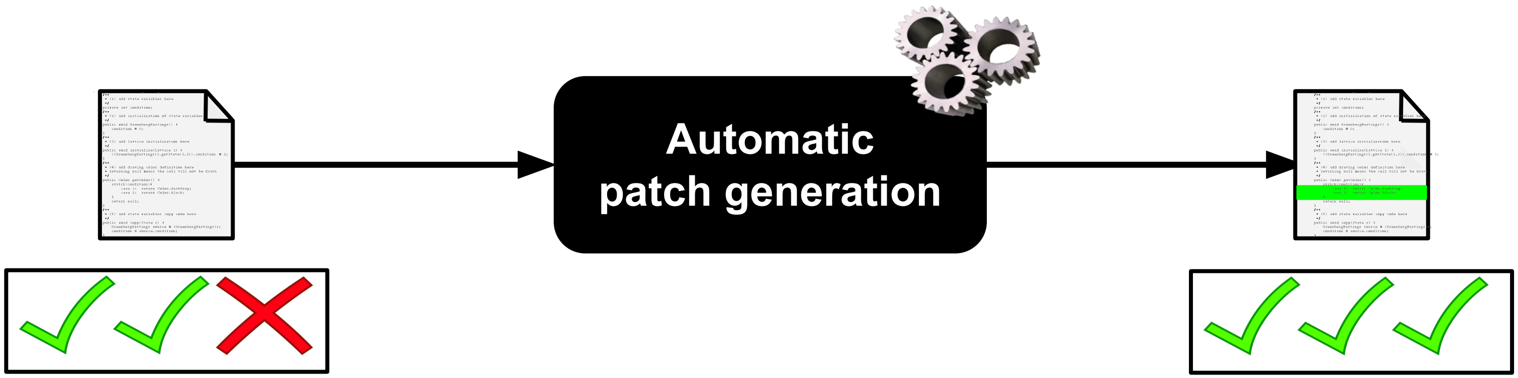
Generate-and-validate Approaches



What are the **main components** of a (generate-and-validate) patch generation approach?

Automatic patch generation (program repair)

Generate-and-validate Approaches



Main components:

- **Fault localization**
- Mutation + fitness evaluation
- Patch validation

Defect prediction

Defect prediction: the addressed problem

Problem

- QA is limited...

Defect prediction: the addressed problem

Problem

- QA is limited...by time and money.

Defect prediction: the addressed problem

Problem

- QA is limited...by time and money.
- How should we allocate limited QA resources?

Defect prediction: the addressed problem

Problem

- QA is limited...by time and money.
- How should we allocate limited QA resources?
  - Focus on components that are most error-prone.
  - Focus on components that are most likely to fail in the field.

How do we know what components are critical or error-prone?

Defect prediction: a bird's-eye view

Model

- Learn a model from historic data (same project vs. different project)

Defect prediction: a bird's-eye view

Model

- Learn a model from historic data (same project vs. different project)

Predictions

- Classification: is a file/method buggy
- Ranking: how many bugs does a file/method contain

Granularity

- Most research has focused on file-level granularity

Defect prediction: a bird's-eye view

Model

- Learn a model from historic data (same project vs. different project)

Predictions

- Classification: is a file/method buggy
- Ranking: how many bugs does a file/method contain

Granularity

- Most research has focused on file-level granularity

Which type of prediction and what granularity are most useful?

Defect prediction: a bird’s-eye view

Model

- Learn a model from historic data (same project vs. different project)

Predictions

- Classification: is a file/method buggy
- Ranking: how many bugs does a file/method contain

Granularity

- Most research has focused on file-level granularity

What types of metrics matter?

Defect prediction: metrics

Change metrics

- Source-code changes
- Code churn
- Previous bugs

Code metrics

- Complexity metrics (e.g., size, McCabe, dependencies)
- Design metrics (e.g., inheritance hierarchy)

Organizational metrics

- Team structure
- Contribution structure
- Communication

What metrics are most important?

Defect prediction: some results

Predictor	Precision	Recall
Pre-Release Bugs	73.80%	62.90%
Test Coverage	83.80%	54.40%
Dependencies	74.40%	69.90%
Code Complexity	79.30%	66.00%
Code Churn	78.60%	79.90%
Org. Structure	86.20%	84.00%

From: N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality. ICSE 2008.

Teaser: static analysis

How does your compiler optimize code?

- Constant folding, common subexpression elimination (avoid computations)
- Liveness analysis (free up registers)
- …

A *dataflow analysis* estimates the value of each expression

Designing a static analysis

Main challenges:

- Choose an *abstract domain*; example: even, odd
  - Must be a lattice: each pair of elements has a unique lub
  - Needs a top (unknown) and a bottom element
- Define a *transfer function* for each language construct

$\langle$ x is odd; y is odd $\rangle$

```
y = x++;
```

$\langle$ x is even; y is odd $\rangle$

Iterate to a fixed point, over the control flow graph

In-class exercise: fault localization