Today

- P590 vs. the real world
- Delta debugging
- Statistical fault localization
- In-class 6
P590 vs. the real world

How to bridge the gap between theory and practice?

- **Use cases** -- what are use cases for mutation-based testing, constraint-based testing, and formal reasoning?
- **Path to adoption** -- how to (incrementally) implement the concepts covered in P590?
- **Open challenges** -- what challenges that you are facing every day should researchers focus on?
Software testing vs. software debugging

Testing: is there a bug?

```java
@Test
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    int i = 0;
    while (i<n) {
        sum = sum + nums[i];
        i = i + 1;
    }
    double avg = sum * n;
    return avg;
}

testAvg failed: 2.0 != 18.0
```

Debugging: where is the bug?

how to fix the bug?
Focus today is software debugging

Testing: is there a bug?
@Test
public void testAvg() {
    double[] nums = new double[]{1.0, 2.0, 3.0};
    double actual = Math.avg(nums);
    double expected = 2.0;
    assertEquals(expected, actual, EPS);
}

testAvg failed: 2.0 != 18.0

Debugging: where is the bug?
how to fix the bug?
Delta debugging
This is a crashing test case

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Priority</th>
<th>Bug Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 3.1</td>
<td>P1</td>
<td>blocker</td>
</tr>
<tr>
<td>Windows 95</td>
<td>P2</td>
<td>critical</td>
</tr>
<tr>
<td>Windows 98</td>
<td>P3</td>
<td>major</td>
</tr>
<tr>
<td>Windows ME</td>
<td>P4</td>
<td>normal</td>
</tr>
<tr>
<td>Windows NT</td>
<td>P5</td>
<td>minor</td>
</tr>
</tbody>
</table>

● **Crashed Mozilla**
● **How would you debug the problem?**
This is a crashing test case

<table>
<thead>
<tr>
<th>Select Name: op sys</th>
<th>Select Name: priority</th>
<th>Select Name: bug severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>--</td>
<td>blocker</td>
</tr>
<tr>
<td>Windows 3.1</td>
<td>P1</td>
<td>critical</td>
</tr>
<tr>
<td>Windows 95</td>
<td>P2</td>
<td>major</td>
</tr>
<tr>
<td>Windows 98</td>
<td>P3</td>
<td>normal</td>
</tr>
<tr>
<td>Windows ME</td>
<td>P4</td>
<td>minor</td>
</tr>
<tr>
<td>Windows 2000</td>
<td>P5</td>
<td>trivial</td>
</tr>
<tr>
<td>Windows NT</td>
<td></td>
<td>enhancement</td>
</tr>
<tr>
<td>Mac System 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mac System 7.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mac System 7.6.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mac System 8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mac System 8.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mac System 8.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mac System 9.x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MacOS X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MacOS X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSDI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FreeBSD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NetBSD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenBSD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BeOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP-UX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRIX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neutrino</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenVMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solaris</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solaris</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SunOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SunOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Crashed Mozilla
- How would you debug the problem?
- A minimal test case is: `<SELECT>`
- Can we automate the process of minimizing test cases?
- Let's design an algorithm
Delta debugging: live example

Debugging a non-termination issue

- Sorting routine.
- Terminates and correctly sorts one large input file.
- Does not terminate on another large input file.
Statistical fault localization
What is statistical fault localization?
What is statistical 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;
}

Test suite

Passing tests

Failing tests

Fault localization technique
What is statistical fault localization?

Program

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

Fault localization technique

Statement ranking

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

Test suite

- Passing tests
- Failing tests
Fault localization: how it works

Program

```java
double avg(double[] nums) {
    int n = nums.length;
    double sum = 0;
    for(int i=0; i<n; ++i) {
        sum += nums[i];
    }
    return sum * n;
}
```
Fault localization: how it works

Program

```java
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

```sql
```

```markdown
```
Fault localization: how it works

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
Fault localization: how it works

**Program**

```java
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
Fault localization: how it works

Program

default 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
Fault localization: how it works

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;
}
Fault localization: how it works

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;
}

Which line seems most suspicious?

- Run all tests
  - t1 passes
  - t2 passes
  - t3 passes
  - t4 fails
  - t5 fails
  - t6 fails
Spectrum-based fault localization

Program

```java
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:**

\[
S(s) = \frac{\text{failed}(s) / \text{totalfailed}}{\text{failed}(s) / \text{totalfailed} + \text{passed}(s) / \text{totalpassed}}
\]

- **Statement covered** by failing test
- **Statement covered** by passing test

More statement is more suspicious!

Jones et al., *Visualization of test information to assist fault localization*, ICSE’02
Mutation-based fault localization

Program

```java
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

```java
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:

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

- Mutant affects failing test outcome
- Mutant breaks passing test

More \(\uparrow\) mutant is more suspicious!

Papadakis and Traon, Metallaxis-FL: mutation-based fault localization, STVR’15
Common structure of SBFL and MBFL

For each element

\[
\lambda
\]

weighting factors

<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>4</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

collect

(identity for SBFL)

<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>
What design decisions matter?

Defined and explored a design space for SBFL and MBFL

- 4 design factors (e.g., formula)
What design decisions matter?

Defined and explored a design space for SBFL and MBFL

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

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

- Top-10 useful for practitioners\(^1\).
- Top-200 useful for automated program repair\(^2\).

<table>
<thead>
<tr>
<th>Technique</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hybrid</td>
<td>36%</td>
<td>45%</td>
<td>85%</td>
</tr>
<tr>
<td>DStar (best SBFL)</td>
<td>30%</td>
<td>39%</td>
<td>82%</td>
</tr>
<tr>
<td>Metallaxis (best MBFL)</td>
<td>29%</td>
<td>39%</td>
<td>77%</td>
</tr>
</tbody>
</table>

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
In-class exercise 6