Reducing the Cost of Program Debugging with Effective Software Fault Localization

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong
Speaker Biographical Sketch

- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas

- Guest Researcher
  Computer Security Division
  National Institute of Standards and Technology (NIST)

- Vice President, IEEE Reliability Society

- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)

- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  - *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*

- Founder & Steering Committee co-Chair for the SERE conference
  (*IEEE International Conference on Software Security and Reliability*)
  (http://paris.utdallas.edu/sere13)
Outline

• Motivation and Background

• Execution Dice-based Fault Localization

• Suspiciousness Ranking-based Fault Localization
  – Program Spectra-based Fault Localization
  – Code Coverage-based Fault Localization
  – Statistical Analysis-based Fault Localization
  – Neural Network-based Fault Localization
  – Similarity Coefficient-based Fault Localization

• Empirical Evaluation

• Theoretical Comparison: Equivalence

• Mutation-based Automatic Bug Fixing

• Conclusions
Motivation

- Testing and debugging activities constitute one of the most expensive aspects of software development
  - Often more than 50% of the cost [Hailpern & Santhanam, 2003]

- Manual debugging is…
  - Tedious
  - Time Consuming
  - Error prone
  - Prohibitively expensive

Need ways to debug… automatically

**Debugging Today**

- Program debugging consists of three fundamental activities
  - Learning that the program has a fault – *fault detection*
  - Finding the location of the fault – *fault localization*
  - Actually removing the fault – *fault fixing*

- A lot of progress has been made in the area of test case generation and thus we can assume that we will have a collection of test cases (i.e., a test set) that can reveal that the program has faults.
  - So the programmer can avoid the first task (*fault detection*).

- Recently *fault localization* has received a lot of focus
  - It is one of the most expensive debugging activities [Vessey, 1985]

- **Fault Fixing** has also been an important research area
  - Have to be very careful not to introduce new faults in the process

---

Software Fault Localization
Objectives

- Develop a robust and reliable fault localization technique to identify faults from *dynamic behaviors* of programs

- Reduce the cost of program debugging by providing *a more accurate set of candidate fault positions*

- Provide software engineers with *effective tool support*
Perfect Bug Detection

• A bug in a statement will be detected by a programmer if the statement is examined
  – A correct statement will not be mistakenly identified as a faulty statement
  – If the assumption does not hold, a programmer may need to examine more code than necessary in order to find a faulty statement
Traditional Approach
Commonly Used Techniques

- Insert *print* statements
- Add *assertions* or set *breakpoints*
- Examine core dump or stack trace

Rely on programmers’ intuition and domain expert knowledge
Outline

• Motivation and Background
• Execution Dice-based Fault Localization
  • Suspiciousness Ranking-based Fault Localization
    – Program Spectra-based Fault Localization
    – Code Coverage-based Fault Localization
    – Statistical Analysis-based Fault Localization
    – Neural Network-based Fault Localization
    – Similarity Coefficient-based Fault Localization
• Empirical Evaluation
• Theoretical Comparison: Equivalence
• Mutation-based Automatic Bug Fixing
• Conclusions
Execution Dice-based Fault Localization
Execution Slice & Dice

• Faults reside in the execution slice of a test that fails on execution
  – An execution slice is the set of a program’s code
    (blocks, statements, decisions, c-uses, or p-uses) executed by a test
  – An execution slice can be constructed very easily if we know the coverage of
    the test (instead of reporting the coverage percentage, it reports which parts of
    the program are covered).
  – Too much code in the slice

• Narrowing search domain by execution dices
  – An execution dice is obtained by subtracting
    successful execution slices from failed execution slices

Dice = Execution slices of failed tests – Execution slices of successful tests
Example (1)

A Sample Program

```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right       : area = b*c / 2;
equilateral  : area = a*a * sqrt(3)/4;
otherwise    : s = (a+b+c)/2;
                 area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```
**Example (2)**

**Initial Test Set**

<table>
<thead>
<tr>
<th>Test case</th>
<th>Input</th>
<th>Output</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td>$T_1$</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$T_2$</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>$T_3$</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>$T_4$</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>$T_5$</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
**Example (3)**

**Failure Detected**

<table>
<thead>
<tr>
<th>Test case</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a b c</td>
<td>class area</td>
</tr>
<tr>
<td>( T_1 )</td>
<td>2 2 2</td>
<td>equilateral 1.73</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>4 4 3</td>
<td>isosceles 5.56</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>5 4 3</td>
<td>right 6.00</td>
</tr>
<tr>
<td>( T_4 )</td>
<td>6 5 4</td>
<td>scalene 9.92</td>
</tr>
<tr>
<td>( T_5 )</td>
<td>3 3 3</td>
<td>equilateral 3.90</td>
</tr>
<tr>
<td>( T_6 )</td>
<td>4 3 3</td>
<td><strong>scalene</strong> 4.47</td>
</tr>
</tbody>
</table>

*Failure!*
Example (4)

Where is the Bug?

```
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right : area = b*c / 2;
    equilateral : area = a*a * sqrt(3)/4;
    otherwise : s = (a+b+c)/2;
                area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```
Example (5)

Execution Slice w.r.t. the Failed Test $T_6 = (4 \ 3 \ 3)$

```plaintext
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right : area = b*c / 2;
equilateral : area = a*a * sqrt(3)/4;
otherwise : s = (a+b+c)/2;
            area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

Too much code needs
To be examined!
**Example (6): Which Test Should be Used?**

<table>
<thead>
<tr>
<th>Test case</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>$T_1$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$T_2$</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>$T_3$</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>$T_4$</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>$T_5$</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$T_6$</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

*Failure!*
### Example (7)

#### A Successful Test $T_2$ and a Failed Test $T_6$

<table>
<thead>
<tr>
<th>Test case</th>
<th>Input</th>
<th>Output Class</th>
<th>Output Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$a = 2$</td>
<td>$b = 2$</td>
<td>$c = 2$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$a = 4$</td>
<td>$b = 4$</td>
<td>$c = 3$</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$a = 5$</td>
<td>$b = 4$</td>
<td>$c = 3$</td>
</tr>
<tr>
<td>$T_4$</td>
<td>$a = 6$</td>
<td>$b = 5$</td>
<td>$c = 4$</td>
</tr>
<tr>
<td>$T_5$</td>
<td>$a = 3$</td>
<td>$b = 3$</td>
<td>$c = 3$</td>
</tr>
<tr>
<td>$T_6$</td>
<td>$a = 4$</td>
<td>$b = 3$</td>
<td>$c = 3$</td>
</tr>
</tbody>
</table>

*Success*

*Failure! (should be isosceles)*
Example (8)

Execution Slice w.r.t. the Successful Test $T_2 = (4 \ 4 \ 3)$

```plaintext
read (a, b, c);
class = scalene;
if a == b || b == a
    class = isosceles;
if a*a == b*b + c*c
    class = right;
if a == b && b == c
    class = equilateral;
case class of
    right : area = b*c / 2;
equilateral : area = a*a * sqrt(3)/4;
otherwise : s = (a+b+c)/2;
            area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```
Example (9)

Execution Dice = Slice (4 3 3) - Slice (4 4 3)

```java
read (a, b, c);
class = scalene;
if a = b || b = a
    class = isosceles;
if a*a = b*b + c*c
    class = right;
if a = b && b = c
    class = equilateral;
case class of
    right : area = b*c / 2;
equilateral : area = a*a * sqrt(3)/4;
otherwise : s = (a+b+c)/2;
    area = sqrt(s*(s-a)*(s-b)*(s-c));
end;
write(class, area);
```

**Bug!**
Should be $b=c$
One Failed and One Successful Test

Possible locations of faults

- Code in the execution dice (top priority)
- A bug is in the failed execution slice (the red path) but not in the successful execution slice (the blue path)
- Code in the failed execution slice but not in the dice
- A bug is in the failed execution slice (the red path) and in the successful execution slice (the blue path)

The dicing-based technique can be effective in locating some program bugs


† Authors are listed in alphabetical order
‡ Number of citations: 155 (according to the Google Scholar)
A test case in green runs the program successfully

A test case in red fails the program
## Locating Bugs using Execution Dice (2)

<table>
<thead>
<tr>
<th>File</th>
<th>Tool</th>
<th>Options</th>
<th>Summary</th>
<th>TestCases</th>
<th>Update</th>
<th>GoBack</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **by-type**
- **by-file**
- **by-function**

### Block Slicing Summary by File Over Selected Testcases

<table>
<thead>
<tr>
<th>File</th>
<th>Coverage</th>
<th>Files</th>
<th>Passed Tests</th>
<th>Failed Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td>69 of 178</td>
<td>38.8%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>misc.c</td>
<td>18 of 101</td>
<td>17.8%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qsort.c</td>
<td>3 of 33</td>
<td>9.1%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comp1.c</td>
<td>0 of 102</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comp2.c</td>
<td>0 of 112</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>skip.c</td>
<td>0 of 29</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sort.c</td>
<td>0 of 59</td>
<td>0%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Total**

<table>
<thead>
<tr>
<th>Coverage: 90 of 614</th>
<th>Passed Tests: 1 of 4</th>
<th>Failed Tests: 1 of 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**χSlice**
Locating Bugs using Execution Dice (3)

Code in blue is executed by the failed test **AND** the successful one.

Code in red is executed by the failed test **BUT NOT** the successful one.

Code in white is **NOT** executed by the failed test.
Multiple Failed and Successful Tests (1)

- The more that successful tests execute a piece of code, the less likely for it to contain any fault.

- The more that failed tests with respect to a given fault execute a piece of code, the more likely for it to contain this fault.

- A piece of code containing a specific fault is
  - inversely proportional to the number of successful tests that execute it
  - proportional to the number of failed tests (with respect to this fault) that execute it.

Multiple Failed and Successful Tests (2)

• Need to consider *precision* and *recall*
  – intersection of failed tests – union of successful tests
  – union of failed tests – union of successful tests
  – intersection of failed tests – intersection of successful tests
  – union of failed tests – intersection of successful tests
More Advanced Heuristics

- A bad dice does not contain the bug
  - Augmentation of a bad execution dice using inter-block data dependency

- A good dice with too much code
  - Refining a good execution dice using additional successful tests

Augmentation of A Bad Execution Dice $D^{(1)}$ (1)

- Bug is not in the execution dice
- Much code that is executed by both the failed test (the red path) and the successful test (the blue path)
- How to prioritize the code that still needs to be examined

Bug!
Augmentation of A Bad Execution Dice $D^{(1)}$ (2)

- If the bug is not in $D^{(1)}$, we need to examine additional code from the rest of the failed execution slice (i.e., $E_F - D^{(1)}$ denoted by $\Phi$)
  - For a block $\beta$, the notation $\beta \in \Phi$ implies $\beta$ is in the failed execution slice $E_F$ but not in $D^{(1)}$.

- More prioritization based on *inter-block data dependency*

- Define a “*direct data dependency*” relation $\Delta$ between a block $\beta$ and an execution dice $D^{(1)}$ such that $\beta \Delta D^{(1)}$

  if and only if $\beta$ defines a variable $x$ that is used in $D^{(1)}$ or $\beta$ uses a variable $y$ defined in $D^{(1)}$. 
Augmentation of A Bad Execution Dice $D^{(1)}$ (3)

- Code in red is in the execution dice
- Code in blue has some data dependency with code in red ➔ Higher Priority
- Code in green has no data dependency with code in red ➔ Lower Priority
**Augmentation of A Bad Execution Dice \( D^{(1)} \) (4)**

- Construct \( \mathcal{A}^{(1)} \), the augmented code segment from the first iteration, such that \( \mathcal{A}^{(1)} = \{ \beta \mid \beta \in \Phi \land (\beta \Delta D^{(1)}) \} \).

- set \( k = 1 \)
- Examine code in \( \mathcal{A}^{(k)} \) to see whether it contains the bug (\( \leftarrow \))
- If YES,
  - then
    - STOP because we have located the bug
  - else
    - set \( k = k + 1 \)
- Construct \( \mathcal{A}^{(k)} \), the augmented code segment from the \( k \)th iteration, such that \( \mathcal{A}^{(k)} = \mathcal{A}^{(k-1)} \cup \{ \beta \mid b \in \Phi \land (\beta \Delta \mathcal{A}^{(k-1)}) \} \).
- If \( \mathcal{A}^{(k)} = \mathcal{A}^{(k-1)} \) (i.e., no new code can be included from the \( (k-1) \)th iteration to the \( k \)th iteration)
  - then
    - STOP
      At this point we have \( \mathcal{A}^{(*)} \), the final augmented code segment, which equals \( \mathcal{A}^{(k)} \) (and \( \mathcal{A}^{(k-1)} \) as well)
  - else
    - Go back to step (\( \leftarrow \))
Augmentation of A Bad Execution Dice \( D^{(1)} \) (5)

```c
#include <stdio.h>

int main() {
    float a, b, c, d, x, y;

    S_0: scanf("%f %f", &a, &b);
    S_1: if (a <= 0)
        S_2:     c = 2*a + 1;
        else
        S_3:     c = 3*a;
    S_4: if (b <= 0)
        S_5:     d = b*b - 4*a*c
        else
        S_6:     d = 5*b;
    S_7: x = b + d;
    S_8: y = c + d;
    S_9: printf("x = %f & y = %f \n", x, y);
}

(a) the execution slice with respect to a failed test \( t_1 \)
    (a=3; b=5)
(b) the execution slice with respect to a successful test \( t_2 \)
    (a= -3; b=5)
#include <stdio.h>

int main() {
    float a, b, c, d, x, y;

    S0   scanf ("%f %f", &a, &b);
    S1   if (a <= 0)
    S2       c = 2*a + 1;
    S3       else
    S4       c = 3*a;
    S5       else
    S6       d = 5*b;
    S7       x = b + d;
    S8       y = c + d;
    S9       printf ("x = %f & y = %f
" , x, y);
}

dice obtained by subtracting the execution slice in (b) from the execution slice in (a)

#include <stdio.h>

int main() {
    float a, b, c, d, x, y;

    S0   scanf ("%f %f", &a, &b);
    S1   if (a <= 0)
    S2       c = 2*a + 1;
    S3       else
    S4       c = 3*a;
    S5       else
    S6       d = 5*b;
    S7       x = b + d;
    S8       y = c + d;
    S9       printf ("x = %f & y = %f\n", x, y);
}

Code that has direct data dependency with S3 (i.e., code in the dice)

Bug! Should be 2*c
Refining of A Good Execution Dice $D^{(1)}$

- Construct the execution slices (denoted by $\Theta_1, \Theta_2, \ldots, \Theta_k$) with respect to successful tests $t_1, t_2, \ldots, t_k$
- $D^{(1)} = E_F - \Theta_1$
- $D^{(2)} = D^{(1)} - \Theta_2 = E_F - \Theta_1 - \Theta_2$
- We have $D^{(1)} \supseteq D^{(2)} \supseteq D^{(3)}$, etc.

Since we want to examine the more suspicious code before the less suspicious code, code in $D^{(2)}$ should be examined before code in $D^{(1)}$ but not in $D^{(2)}$.

Part (a) Code in $D^{(1)}$ is highlighted in red.
Part (b) Code in $D^{(2)}$ is highlighted in red.
Part (c) Code in $D^{(3)}$ is highlighted in red.
An Incremental Approach

- Assume
  - debugging as soon as a failure is detected (i.e., only one failed test)
  - \( n \) (say 3) successful tests
- Assume the bug is in the code which is executed by the failed test but not the successful test(s)
  - first examining the code in \( D(3) \) followed by code in \( D(2) \) but not in \( D(3) \), then code in \( D(1) \) but not in \( D(2) \)
- If this assumption does not hold (i.e., the bug is not in \( D(1) \)), then we need to inspect additional code in the failed execution slice but not in \( D(1) \)
  - then starting with code in \( A(1) \) but not in \( D(1) \), followed by \( A(2) \) but not in \( A(1) \), …

- Prioritize code in a failed execution slice based on its likelihood of containing the bug. The prioritization is done by first using the refining method and then the augmentation method.
  - Examining code in \( D(3) \), \( D(2) \) but not in \( D(3) \), \( D(1) \) but not in \( D(2) \), \( A(1) \) but not in \( D(1) \), \( A(2) \) but not in \( A(1) \), \( A(3) \) but not in \( A(2) \), … etc.
- In the worst case, we have to examine all the code in the failed execution slice.
Outline

• Motivation and Background
• Execution Dice-based Fault Localization
  • Suspiciousness Ranking-based Fault Localization
    – Program Spectra-based Fault Localization
    – Code Coverage-based Fault Localization
    – Statistical Analysis-based Fault Localization
    – Neural Network-based Fault Localization
    – Similarity Coefficient-based Fault Localization
• Empirical Evaluation
• Theoretical Comparison: Equivalence
• Mutation-based Automatic Bug Fixing
• Conclusions
Suspiciousness Ranking-based Fault Localization
Overview

• Compute the suspiciousness (likelihood of containing bug) of each statement

• Rank all the executable statements in descending order of their suspiciousness

• Examine the statements one-by-one from the top of the ranking until the first faulty statement is located

• Statements with higher suspiciousness should be examined before statements with lower suspiciousness as the former are more likely to contain bugs than the latter
Techniques for Computing Suspiciousness

- Code coverage-based and calibration
- Crosstab: statistical analysis-based
- BP (Back Propagation) & RBF (Radial Basis Function) neural network
- Similarity coefficient-based
- Tarantula: heuristic-based
- SOBER: statistical analysis-based
- Liblit: statistical analysis-based

Take advantage of code coverage (namely, execution slice) and execution result of each test (success or failure) for debugging.
Motivation and Background

Execution Dice-based Fault Localization

Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization

Empirical Evaluation

Theoretical Comparison: Equivalence

Mutation-based Automatic Bug Fixing

Conclusions
Spectra-based Fault Localization
### Spectra-Based Fault Localization Techniques

#### Possible Program Spectra

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BHS</td>
<td>Branch Hit Spectra</td>
</tr>
<tr>
<td>BCS</td>
<td>Branch Count Spectra</td>
</tr>
<tr>
<td>CPS</td>
<td>Complete Path Spectra</td>
</tr>
<tr>
<td>PHS</td>
<td>Path Hit Spectra</td>
</tr>
<tr>
<td>PCS</td>
<td>Path Count Spectra</td>
</tr>
<tr>
<td>DHS</td>
<td>Data-Dependence Hit Spectra</td>
</tr>
<tr>
<td>DCS</td>
<td>Data-Dependence Count Spectra</td>
</tr>
<tr>
<td>OPS</td>
<td>Output Spectra</td>
</tr>
<tr>
<td>ETS</td>
<td>Execution Trace Spectra</td>
</tr>
<tr>
<td>DVS</td>
<td>Data Value Spectra</td>
</tr>
<tr>
<td>ESHS</td>
<td>Executable Statement Hit Spectra</td>
</tr>
</tbody>
</table>
A Sample Program for Program Spectra

- Given an integer $n$ and a real number $x$, the program calculates $x^n$

```c
1   double power (double x, int n)
2   {
3       int i;
4       int rv = 1;
5       for (i=0; i<abs(n); i++)
6           {
7               rv = rv \times x;
8           }
9       if (n<0)
10           {
11               if (x!=0)
12                   rv = 1/rv;
13               else
14                   {
15                   printf("Error input.\n");
16                   return 0;
17               }
18           }
19       return rv;
20   }
```
**Branch Hit Spectra**

- BHS records the *conditional branches* that are covered by the test execution.
- Suppose there are \( m \) conditional branches: \( b_1, b_2, \ldots, b_m \).
- The spectrum with respect to \( b_i \) (\( i = 1, 2, \ldots, m \)) indicates whether \( b_i \) is covered by the test execution.
- There are 6 branches in the sample program: (5,7), (5,9), (9,19), (9,11), (11,12), and (11,15).
- When test case \( (x = 2, n = 3) \) is executed, the branch hit spectrum is \((Y, Y, Y, N, N, N)\).
**Branch Count Spectra**

- BCS records the *number of times that each conditional branch* is executed.
- Suppose there are $m$ conditional branches: $b_1, b_2, \ldots, b_m$. The spectrum with respect to $b_i$ ($i = 1, 2, \ldots, m$), denoted by $s_i$, indicates that $b_i$ is executed $s_i$ times by the test execution.
- When test case (2,3) is executed, the *branch count spectrum* is $(3, 1, 1, 0, 0, 0)$.
  - (5,7) is executed 3 times
  - (9,19) is executed one time
  - (11,15) is not executed
**Complete Path Spectra**

- CPS records the *complete paths* that are traversed by the test execution
- When test case (2,3) is executed, the CPS is \((3,4,(5,7)^3,9,19)\)

Statement 5 and 7 are executed 3 times
**Path Hit Spectra**

- PHS records the *intra-procedural, loop-free paths* that are covered by the test execution.

- The sample program has six possible paths:
  - 3,4,5,9,19
  - 3,4,5,7,9,19
  - 3,4,5,9,11,12,19
  - 3,4,5,7,9,11,12,19
  - 3,4,5,9,11,15,16
  - 3,4,5,7,9,11,15,16

- With respect to the execution of test case (2,3), the *path hit spectrum* can be represented by:
  - (Y,N,N,N,N,N)

  *(3,4,5,9,19) is covered  (3,4,5,7,9,11,15,16) is not covered*
**Path Count Spectra**

- PCS records the *number of times that each intra-procedural, loop-free path is covered by the test execution*.

- The sample program has six possible loop-free paths:
  - 3,4,5,9,19
  - 3,4,5,7,9,19
  - 3,4,5,9,11,12,19
  - 3,4,5,7,9,11,12,19
  - 3,4,5,9,11,15,16
  - 3,4,5,7,9,11,15,16

- When test case (2,3) is executed, the *path count spectrum* can be represented by:
  - (1,0,0,0,0,0)

- When the function is executed more than one time, the elements in PCS may be larger than 1.
**Data-Dependence Hit Spectra**

- DHS records the *definition-use pairs* that are covered by the execution.
- With respect to the sample program, let’s focus on the following definition-use pairs:
  - (rv, 4, 7)
  - (rv, 4, 19)
  - (rv, 7, 7)
  - (rv, 7, 12)
  - (rv, 7, 19)
  - (rv, 12, 19)
- When test case (2,3) is executed, the spectrum can be represented by
  - (Y,N,Y,N,Y,N) which implies (rv,4,7), (rv,7,7) and (rv,7,19) are covered by this execution.
Data-Dependence Count Spectra

- DCS records the **number of times that each definition-use pair is executed**

- With respect to the sample program, let’s focus on the following definition-use pairs
  - (rv, 4, 7)
  - (rv, 4, 19)
  - (rv, 7, 7)
  - (rv, 7, 12)
  - (rv, 7, 19)
  - (rv, 12, 19)

- When test case (2,3) is executed, the **data-dependence count spectrum** can be represented by (1,0,2,0,1,0)

  (rv, 7, 7) is executed 2 times
Output Spectra

- OPS records the *outputs produced* by the test executions.

- With respect to the sample program, when test case (2,3) is executed, the *output spectrum* can be represented by a value 8, which is the output of the function.
**Execution Trace Spectra**

- ETS records the *sequence of each program statement* traversed by the test execution.

- With respect to the sample program, when case (2,3) is executed, the execution trace spectrum can be represented by:

  \[(\text{int } i, \text{ double } rv = 1, (\text{for}(i=0; i<\text{abs}(n); i++), rv = rv * x )^3, \text{if}(n<0), \text{return } rv)\]

- Difference between ETS and CPS (Complete Path Spectrum):
  - ETS records the actual instructions, whereas CPS does not.

These statements are executed 3 times.
Data Value Spectra

- DVS records the *values of variables*

- With respect to the sample program, we focus on the value of variable \( rv \)
  - When test case (2,3) is executed, the sequence of the values of \( rv \) is (1,2,4,8) which is one of the DVS representations
Executable Statement Hit Spectra

- ESHS records the \textit{executable statements that are covered} by the test execution.
  - excluding comments, blank lines, (some) variable declarations, function declarations, etc.

- Suppose there are $m$ executable statements: $s_1, s_2, \ldots, s_m$

- The spectrum with respect to $s_i$ ($i = 1, 2, \ldots, m$), indicates whether $s_i$ is covered by the test execution.

- There are 9 executable statements at lines 4, 5, 7, 9, 11, 12, 15, 16 and 19

- When test case (2,3) is executed, the \textit{executable statement hit spectrum} is
  \((\bigcirc, Y, Y, Y, \bigcirc, N, N, N, Y)\).

\[\text{Statement 4 is executed} \quad \text{Statement 11 is not executed}\]
**Outline**

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions
Code Coverage-based Fault Localization & Calibration

Suppose for a large test suite, say 1000 test cases, a majority of them, say 995, are successful test cases and only a small number of failed test cases (five in this example) will cause an execution failure.

The challenge is how to use these five failed tests and the 995 successful tests to conduct an effective debugging.

- How can each additional test case that executes the program successfully help locate program bugs?
- What about each additional test case that makes the program execution fail?
Code Coverage-based & Calibration (2)

- Should all the successful test executions provide the same contribution to locate software bugs?

- Intuitively, the answer should be “no”

- If a piece of code has already been executed successfully 994 times, then the contribution of the 995th successful execution is likely to be less than, for example, the contribution of the second successful execution when the code is only executed successfully once

- We propose that with respect to a piece of code, the contribution introduced by the first successful test that executes it in computing its likelihood of containing a bug is larger than or equal to that of the second successful test that executes it, which is larger than or equal to that of the third successful test that executes it, etc.

- The same also applies to the failed tests.
If the statement $S$ is executed by at least one failed test, then the total contribution from all the successful tests that execute $S$ should be less than the total contribution from all the failed tests that execute $S$ (namely, $\sum_{i=1}^{N_S} c_{S,i} < \sum_{k=1}^{N_F} c_{F,k}$).

All the tests in the same failed group have the same contribution towards fault localization, but tests from different groups have different contributions.
**Code Coverage-based Calibration (4)**

- For illustrative purposes, we set $G_F = G_S = 3$, $n_{F,1} = n_{S,1} = 2$, and $n_{F,2} = n_{S,2} = 4$
  - The first failed (or successful) group has at most two tests, the second group has at most four from the remaining, and the third has everything else, if any.

- We also assume each test case in the first, second, and third failed groups gives a contribution of 1, 0.1 and 0.01, respectively ($w_{F,1} = 1$, $w_{F,2} = 0.1$, and $w_{F,3} = 0.01$).

- Similarly, we set $w_{S,1} = 1$, $w_{S,2} = 0.1$, and $w_{S,3}$ to be a small value defined as $\alpha \times \chi_{F/S}$ where $\alpha$ is a scaling factor.

\[
\left[ (1.0) \times n_{F,1} + (0.1) \times n_{F,2} + (0.01) \times n_{F,3} \right] - \left[ (1.0) \times n_{S,1} + (0.1) \times n_{S,2} + \alpha \times \chi_{F/S} \times n_{S,3} \right]
\]

where

\[
n_{F,1} = \begin{cases} 0, & \text{for } N_F = 0 \\ 1, & \text{for } N_F = 1 \\ 2, & \text{for } N_F \geq 2 \end{cases}
\]

\[
n_{F,2} = \begin{cases} 0, & \text{for } N_F \leq 2 \\ N_F - 2, & \text{for } 3 \leq N_F \leq 6 \\ 4, & \text{for } N_F > 6 \end{cases}
\]

\[
n_{F,3} = \begin{cases} 0, & \text{for } N_F \leq 6 \\ N_F - 6, & \text{for } N_F > 6 \end{cases}
\]

\[
n_{S,1} = \begin{cases} 0, & \text{for } N_S = 0, 1 \\ 1, & \text{for } N_S = 2 \text{ and } N_F \geq 1 \end{cases}
\]

\[
n_{S,2} = \begin{cases} 0, & \text{for } N_S \leq n_{F,1} \\ N_S - n_{F,1}, & \text{for } n_{F,1} < N_S < n_{F,1} + n_{S,2} \\ n_{S,2}, & \text{for } N_S \geq n_{F,1} + n_{S,2} \end{cases}
\]

\[
n_{F_S} = \begin{cases} 0, & \text{for } N_S \leq n_{F,1} + n_{S,2} \\ N_S - n_{F,1} - n_{S,2}, & \text{for } n_{F,1} + n_{S,2} < N_S \leq n_{F,1} + n_{S,2} + n_{S,3} \\ n_{S,3}, & \text{for } N_S > n_{F,1} + n_{S,2} + n_{S,3} \end{cases}
\]
### Code Coverage-based & Calibration (5)

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
<th>$S_7$</th>
<th>$S_8$</th>
<th>$S_9$</th>
<th>$S_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_2$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_3$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_4$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_5$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_6$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_7$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_8$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_9$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_{11}$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_{12}$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_{13}$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_{14}$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_{15}$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_{16}$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_{17}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_{18}$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_{19}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$t_{20}$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Number of failed tests that execute each statement:**

- $r_{failed}$

**Number of successful tests that execute each statement:**

- $r_{successful}$

**Suspiciousness of each statement:**

- $s_{i} = \frac{r_{failed} - r_{successful}}{r_{total}}$

<table>
<thead>
<tr>
<th>Failed Tests</th>
<th>Successful Tests</th>
<th>Suspiciousness</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td>0.993</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0.998</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0.980</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>0.980</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.970</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>0.970</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0.963</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>0.980</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>0.963</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>0.993</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0.993</td>
</tr>
</tbody>
</table>

**More Details**
Two fundamental principles

- $c_{S,1} \geq c_{S,2} \geq c_{S,3} \geq \ldots \geq c_{S,N_S}$ and $c_{F,1} \geq c_{F,2} \geq c_{F,3} \geq \ldots \geq c_{F,N_F}$

- $\sum_{i=1}^{N_S} c_{S,i} < \sum_{k=1}^{N_F} c_{F,k}$
Outline

• Motivation and Background
• Execution Dice-based Fault Localization
• Suspiciousness Ranking-based Fault Localization
  – Program Spectra-based Fault Localization
  – Code Coverage-based Fault Localization
  – Statistical Analysis-based Fault Localization
  – Neural Network-based Fault Localization
  – Similarity Coefficient-based Fault Localization
• Empirical Evaluation
• Theoretical Comparison: Equivalence
• Mutation-based Automatic Bug Fixing
• Conclusions
Crosstab-based Fault Localization

The crosstab (cross-classification table) analysis is used to study the relationship between two or more categorical variables.

A crosstab is constructed for each statement as follows

<table>
<thead>
<tr>
<th></th>
<th>(\omega) is covered</th>
<th>(\omega) is not covered</th>
<th>(\sum)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>(N_{CS}(\omega))</td>
<td>(N_{US}(\omega))</td>
<td>(N_{S})</td>
</tr>
<tr>
<td>(N_F)</td>
<td>(N_{CF}(\omega))</td>
<td>(N_{UF}(\omega))</td>
<td>(N_{F})</td>
</tr>
<tr>
<td>(N_C(\omega))</td>
<td>(N_{C}(\omega))</td>
<td>(N_{U}(\omega))</td>
<td>(N)</td>
</tr>
</tbody>
</table>

- \(N\): total number of test cases
- \(N_F\): total number of failed test cases
- \(N_S\): total number of successful test cases
- \(N_C(\omega)\): number of test cases covering \(\omega\)
- \(N_{CF}(\omega)\): number of failed test cases covering \(\omega\)
- \(N_{CS}(\omega)\): number of successful test cases covering \(\omega\)
- \(N_U(\omega)\): number of test cases not covering \(\omega\)
- \(N_{UF}(\omega)\): number of failed test cases not covering \(\omega\)
- \(N_{US}(\omega)\): number of successful test cases not covering \(\omega\)
Dependency Relationship (1)

For each crosstab, we conduct a hypothesis test to check the dependency relationship. The null hypothesis is

\[ H_0: \text{Program execution result is independent of the coverage of statement } \omega \]

A chi-square test can be used to determine whether this hypothesis should be rejected. The Chi-square statistic is given by

\[
\chi^2(\omega) = \frac{(N_{CF}(\omega) - E_{CF}(\omega))^2}{E_{CF}(\omega)} + \frac{(N_{CS}(\omega) - E_{CS}(\omega))^2}{E_{CS}(\omega)} + \frac{(N_{UF}(\omega) - E_{UF}(\omega))^2}{E_{UF}(\omega)} + \frac{(N_{US}(\omega) - E_{US}(\omega))^2}{E_{US}(\omega)}
\]  

(1)

where \( E_{CF}(\omega) = \frac{N_c(\omega) \times N_F}{N} \), \( E_{CS}(\omega) = \frac{N_c(\omega) \times N_S}{N} \), \( E_{US}(\omega) = \frac{N_U(\omega) \times N_S}{N} \), and \( E_{UF}(\omega) = \frac{N_U(\omega) \times N_F}{N} \).

Under the null hypothesis, the statistic \( \chi^2(\omega) \) has approximately a Chi-square distribution.
Dependency Relationship (2)

• Given a level of significance $\sigma$ (for example, 0.05), we can find the corresponding Chi-square critical value $\chi^2_\sigma$, from the Chi-square distribution table.

  – If $\chi^2(\omega) > \chi^2_\sigma$, we reject the null hypothesis, i.e., the execution result is dependent on the coverage of $\omega$.

  – Otherwise, we accept the null hypothesis, i.e., the execution result and the coverage of $\omega$ are “independent.”
Degree of Association (1)

• The “dependency” relationship indicates a high association among the variables, whereas the “independency” relationship implies a low association.

• Instead of the so-called “dependency”/“independency” relationship, we are more interested in the degree of association between the execution result and the coverage of each statement.

• This degree can be measured based on the standard Chi-square statistic. However, such a measure increases with increasing sample size. As a result, the measure by itself may not give the “true” degree of association.

• One way to fix this problem is to use the contingency coefficient computed as follows

\[ M(\omega) = \frac{\chi^2(\omega)/N}{\sqrt{(row-1)(col-1)}} \]  

where row and col are the number of categorical variables in all rows and columns, respectively, of the crosstab.
The contingency coefficient $M(\omega)$ lies between 0 and 1.

- When $\chi^2(\omega) = 0$, it has the lower limit 0 for complete independence.
- In the case of complete association, the coefficient can reach the upper limit 1 when $row = col$

In our case, $row = col = 2$ and $N$ is fixed. From Equation (2), $M(\omega)$ increases with increasing $\chi^2(\omega)$.

Under this condition, the Chi-square statistic $\chi^2(\omega)$ for statement $\omega$ gives a good indication of the degree of the association between the execution result and the coverage of $\omega$.

- $N$ is fixed because every faulty version is executed with respect to all the test cases.
What kind of Execution Result is More Associated (1)

- Need to decide whether it is the failed or the successful execution result that is more associated with the coverage of the statement.

- For each statement \( \omega \), we compute \( P_F(\omega) \) and \( P_S(\omega) \) as \( \frac{N_{CT}(\omega)}{N_p} \) and \( \frac{N_{CS}(\omega)}{N_s} \) which are the percentages of all failed and successful tests that execute \( \omega \).

- If \( P_F(\omega) \) is larger than \( P_S(\omega) \), then the association between the failed execution and the coverage of \( \omega \) is higher than that between the successful execution and the coverage of \( \omega \).
**What kind of Execution Result is More Associated**

(2)

- We define $\varphi(\omega)$ as
  
  $$\varphi(\omega) = \frac{P_F(\omega)}{P_S(\omega)} = \frac{N_{CF}(\omega)/N_F}{N_{CS}(\omega)/N_S}$$

(3)

- If $\varphi(\omega) = 1$, we have $\chi^2(\omega) = 0$, which implies the execution result is completely independent of the coverage of $\omega$. In this case, we say the coverage of $\omega$ makes the same contribution to both the failed and the successful execution result.

- If $\varphi(\omega) > 1$, the coverage of $\omega$ is more associated with the failed execution.

- If $\varphi(\omega) < 1$, the coverage of $\omega$ is more associated with the successful execution.
**Five Classes of Statements**

- Depending on the values of $\chi^2(\omega)$ and $\varphi(\omega)$, statements of the program being debugged can be classified into five classes:
  - Statements with $\varphi > 1$ and $\chi^2 > \chi^2_{\sigma}$ have a high degree of association between their coverage and the failed execution result.
  - Statements with $\varphi > 1$ and $\chi^2 \leq \chi^2_{\sigma}$ have a low degree of association between their coverage and the failed execution result.
  - Statements with $\varphi < 1$ and $\chi^2 > \chi^2_{\sigma}$ have a high degree of association between their coverage and the successful execution result.
  - Statements with $\varphi < 1$ and $\chi^2 \leq \chi^2_{\sigma}$ have a low degree of association between their coverage and the successful execution result.
  - Statements with $\varphi = 1$ (under this situation $0 = \chi^2 < \chi^2_{\sigma}$) whose coverage is independent of the execution result.

Statements in the first class are most likely (i.e., have the highest suspiciousness) to contain program bugs followed by those in the second, the fifth, and the fourth classes, respectively. Statements in the third class are least likely (i.e., have the least suspiciousness) to contain bugs.
Suspiciousness of Each Statement

- The larger the coefficient $M(\omega)$, the higher the association between the execution result and the coverage of $\omega$.
  - For statements in the first and the second classes, those with a larger $M$ are more suspicious.
  - For statements in the third and the fourth classes, those with a smaller $M$ are more suspicious.

- The suspiciousness of a statement $\omega$ can be defined by a statistic $\zeta$ as

$$\zeta(\omega) = \begin{cases} 
M(\omega) & \text{if } \varphi(\omega) > 1 \\
0 & \text{if } \varphi(\omega) = 1 \\
- M(\omega) & \text{if } \varphi(\omega) < 1 
\end{cases}$$  \hspace{1cm} (4)

- Each $\zeta$ lies between -1 and 1. The larger the $\zeta$ value, the more suspicious the statement $\omega$. 
**Crosstab Example (1)**

- The following table gives the **statement coverage** and **execution results**. Of the 36 test cases, there are nine failed tests (e.g., \( t_1 \)) and 27 successful tests (e.g., \( t_2 \))
  - An entry 1 implies the statement is **covered** by the corresponding test and an entry 0 means it is not.
  - An entry 1 implies a **failed** execution and an entry 0 means a successful execution.

<table>
<thead>
<tr>
<th></th>
<th>( s_1 )</th>
<th>( s_2 )</th>
<th>( s_3 )</th>
<th>( s_4 )</th>
<th>( s_5 )</th>
<th>( s_6 )</th>
<th>( s_7 )</th>
<th>( s_8 )</th>
<th>( s_9 )</th>
<th>( s_{10} )</th>
<th>( r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( t_6 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( t_7 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_8 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_9 )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_{10} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_{11} )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_{12} )</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_{13} )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( t_{14} )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_{15} )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_{16} )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( t_{17} )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( t_{18} )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

| \( f_{19} \) | 1         | 1         | 1         | 1         | 0         | 0         | 1         | 0         | 1         | 1         | 0     |
| \( f_{20} \) | 0         | 0         | 1         | 1         | 1         | 1         | 0         | 1         | 1         | 0         | 1     |
| \( f_{21} \) | 1         | 1         | 0         | 0         | 0         | 1         | 1         | 0         | 1         | 0         | 1     |
| \( f_{22} \) | 0         | 0         | 0         | 1         | 1         | 0         | 1         | 1         | 0         | 0         | 0     |
| \( f_{23} \) | 0         | 1         | 1         | 0         | 0         | 0         | 1         | 0         | 0         | 1         | 1     |
| \( f_{24} \) | 1         | 1         | 0         | 1         | 0         | 0         | 0         | 1         | 0         | 0         | 1     |
| \( f_{25} \) | 1         | 0         | 0         | 1         | 1         | 1         | 0         | 1         | 0         | 0         | 1     |
| \( f_{26} \) | 1         | 1         | 1         | 0         | 1         | 0         | 0         | 1         | 1         | 0         | 1     |
| \( f_{27} \) | 1         | 0         | 0         | 0         | 1         | 1         | 0         | 1         | 1         | 0         | 1     |
| \( f_{28} \) | 0         | 1         | 1         | 1         | 1         | 0         | 0         | 1         | 0         | 0         | 1     |
| \( f_{29} \) | 0         | 0         | 1         | 1         | 1         | 1         | 0         | 1         | 0         | 0         | 1     |
| \( f_{30} \) | 1         | 0         | 0         | 0         | 1         | 1         | 0         | 1         | 1         | 0         | 1     |
| \( f_{31} \) | 1         | 1         | 0         | 1         | 0         | 0         | 1         | 0         | 1         | 0         | 0     |
| \( f_{32} \) | 1         | 0         | 0         | 1         | 1         | 1         | 0         | 0         | 1         | 0         | 1     |
| \( f_{33} \) | 1         | 1         | 0         | 0         | 0         | 1         | 0         | 1         | 1         | 0         | 1     |
| \( f_{34} \) | 1         | 0         | 1         | 1         | 1         | 0         | 0         | 1         | 1         | 0         | 1     |
| \( f_{35} \) | 0         | 0         | 0         | 1         | 1         | 0         | 1         | 1         | 0         | 0         | 1     |
| \( f_{36} \) | 0         | 0         | 1         | 0         | 1         | 0         | 0         | 1         | 0         | 0         | 1     |
**Crosstab Example (2)**

- We can construct the crosstab for $s_1$ as shown in the following

<table>
<thead>
<tr>
<th></th>
<th>$s_1$ is covered</th>
<th>$s_1$ is not covered</th>
<th>$\sum$</th>
</tr>
</thead>
<tbody>
<tr>
<td>successful executions</td>
<td>16</td>
<td>11</td>
<td>27</td>
</tr>
<tr>
<td>failed executions</td>
<td>9</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>$\sum$</td>
<td>25</td>
<td>11</td>
<td>36</td>
</tr>
</tbody>
</table>

- We have
  
  $$E_{CF}(s_1) = \frac{N_C(s_1) \times N_E}{N} = \frac{25 \times 9}{36} = 6.25,$$
  
  $$E_{CS}(s_1) = \frac{N_C(s_1) \times N_S}{N} = \frac{25 \times 27}{36} = 18.75,$$
  
  $$E_{UF}(s_1) = \frac{N_U(s_1) \times N_E}{N} = \frac{11 \times 9}{36} = 2.75,$$
  
  $$E_{US}(s_1) = \frac{N_U(s_1) \times N_S}{N} = \frac{11 \times 27}{36} = 8.25.$$

- From Equation (1)
  
  $$\chi^2(s_i) = \frac{(N_{CF}(s_i) - E_{CF}(s_i))^2}{E_{CF}(s_i)} + \frac{(N_{CS}(s_i) - E_{CS}(s_i))^2}{E_{CS}(s_i)} + \frac{(N_{UF}(s_i) - E_{UF}(s_i))^2}{E_{UF}(s_i)} + \frac{(N_{US}(s_i) - E_{US}(s_i))^2}{E_{US}(s_i)}$$
  
  $$= \frac{(9 - 6.25)^2}{6.25} + \frac{(16 - 18.75)^2}{18.75} + \frac{(0 - 2.75)^2}{2.75} + \frac{(11 - 8.25)^2}{8.25} = 5.2800$$
If we choose the level of significance as 0.05, the Chi-square critical value is 3.841. Since $\chi^2(s_1) = 5.2800$ is larger than 3.841, the null hypothesis for $s_1$ should be rejected.

Similarly, we can compute $\chi^2$ for other statements. For example, we have $\chi^2(s_2) = 4.4954$, $\chi^2(s_3) = 0.1481$, and $\chi^2(s_4) = 1.3333$.

Next, we use Equation (2) to compute the contingency coefficient $M$ for each statement. We have $M(s_1) = 0.1467$, $M(s_2) = 0.1249$, $M(s_3) = 0.0041$, and $M(s_4) = 0.0370$.

Compute $\phi$ and $\zeta$ using Equations (3) and (4).

Based on the suspiciousness, statement $s_8$ should be examined first for locating program bugs followed by $s_1$, $s_5$, $s_{10}$, $s_9$, $s_6$, $s_3$, $s_7$, $s_4$, and $s_2$. 
Outline

• Motivation and Background
• Execution Dice-based Fault Localization
• Suspiciousness Ranking-based Fault Localization
  – Program Spectra-based Fault Localization
  – Code Coverage-based Fault Localization
  – Statistical Analysis-based Fault Localization
  – Neural Network-based Fault Localization
  – Similarity Coefficient-based Fault Localization
• Empirical Evaluation
• Theoretical Comparison: Equivalence
• Mutation-based Automatic Bug Fixing
• Conclusions
RBF Neural Network-based Fault Localization


A typical RBF neural network has a three-layer feed-forward structure

- **Input layer**: Serve as an input distributor to the hidden layer by passing inputs to the hidden layer without changing their values.

- **Hidden layer**: All neurons in this layer simultaneously receive the \( n \)-dimensional real-valued input vector \( x \).

- Each neuron uses a **Radial Basis Function (RBF)** as the activation function
- An RBF is a strictly positive radically symmetric function, where the center \( \mu \) has the unique maximum and the value drops off rapidly to zero away from the center
- When the distance between \( x \) and \( \mu \) (denoted as \( ||x - \mu|| \)) is smaller than the receptive field width \( \sigma \), the function has an appreciable value.
- A commonly used RBF is the **Gaussian basis function**

\[
R_j(x) = \exp\left(-\frac{||x - \mu_j||^2}{2\sigma_j^2}\right)
\]

where \( \mu_j \) and \( \sigma_j \) are the **mean** (namely, the **center**) and the **standard deviation** (namely, the **width**) of the receptive field of the \( j^{th} \) hidden layer neuron, and \( R_j(x) \) is the corresponding activation function.
RBF Neural Network (2)

- Usually the distance $\|x - \mu\|$ is the Euclidean distance between $x$ and $\mu$ ($\|x - \mu\|_E$).
- However, ($\|x - \mu\|_E$) is inappropriate in the fault localization context.
- We use a weighted bit-comparison based distance ($\|x - \mu\|_{WBC}$).

Let $x$ be $c_{t_i}$ (the coverage vector of $i^{th}$ test case $t_i$)

$$\|c_{t_i} - \mu_j\|_{WBC} = \sqrt{1 - \cos \theta_{c_{t_i}, \mu_j}}$$

where $\cos \theta_{c_{t_i}, \mu_j} = \frac{c_{t_i} \cdot \mu_j}{\|c_{t_i}\|_E \|\mu_j\|_E} = \frac{\sum_{k=1}^{m} (c_{t_i})_k (\mu_j)_k}{\sqrt{\sum_{k=1}^{m} [(c_{t_i})_k]^2} \times \sqrt{\sum_{k=1}^{m} [(\mu_j)_k]^2}}$,

where $(c_{t_i})_k$ and $(\mu_j)_k$ are the $k^{th}$ element of $c_{t_i}$ and $\mu_j$, respectively.

This distance is more desirable because it effectively takes into account the number of bits that are both 1 in two coverage vectors (i.e., those statements covered by both vectors).
**RBF Neural Network (3)**

- **Output layer:** \( y = [y_1, y_2, \ldots, y_k] \) with \( y_i \) as the output of the \( i^{th} \) neuron given by

\[
y_i = \sum_{j=1}^{h} w_{ji} R_j(x) \quad \text{for } i = 1, 2, \ldots, k
\]

where \( h \) is the number of neurons in the hidden layer and \( w_{ji} \) is the weight associated with the link connecting the \( j^{th} \) hidden layer neuron and the \( i^{th} \) output layer neuron.
**RBF Neural Network**

An RBF network implements a mapping from the $m$ dimensional real-valued input space to the $k$ dimensional real-valued output space. In between, there is a layer of hidden-layer space.

- The transformation from the input space to the hidden-layer space is **nonlinear**, whereas the transformation from the hidden-layer space to the output space is **linear**.
- The parameters that need to be trained are the *centers* (i.e., $m_1, m_2, \ldots, m_h$) and *widths* (i.e., $s_1, s_2, \ldots, s_h$) of the receptive fields of hidden layer neurons, and the *output layer weights*. 
RBF Neural Network (5)

- We construct an RBF neural network with
  - $m$ input layer neurons (each of which corresponds to one element in a given coverage vector of a test case)
  - one output layer neuron (corresponding to the execution result of test $t_i$)
  - one hidden layer between the input and output layers
RBF Neural Network (6)

• Once an RBF network is trained, it provides a good mapping between the input (the coverage vector of a test case) and the output (the corresponding execution result).

• It can then be used to identify suspicious code of a given program in terms of its likelihood of containing bugs.

• To do so, we use a set of virtual test cases $v_1, v_2, \ldots, v_m$ whose coverage vectors are where

$$
\begin{bmatrix}
    c_{v_1} \\
    c_{v_2} \\
    M \\
    c_{v_m}
\end{bmatrix} =
\begin{bmatrix}
    1 & 0 & L & 0 \\
    0 & 1 & L & 0 \\
    M & M & O & M \\
    0 & 0 & L & 1
\end{bmatrix}
$$

Note that execution of test $v_j$ covers only one statement $s_j$.
If the execution of $v_j$ fails, the probability that the bugs are contained in $s_j$ is high.

This suggests that during the fault localization, we should first examine the statements whose corresponding virtual test case fails.

However, the execution results of these virtual tests can rarely be collected in the real world because it is very difficult, if not impossible, to construct such tests.

When the coverage vector $c_{v_j}$ of a virtual test case $v_j$ is input to the trained neural network, its output $\hat{r}_{v_j}$ is the conditional expectation of whether the execution of $v_j$ fails given $c_{v_j}$.

This implies the larger the value of $\hat{r}_{v_j}$, the more likely that the execution of $v_j$ fails.

Together, we have the larger the value of $\hat{r}_{v_j}$ the more likely it is that $s_j$ contains the bug.

We can treat $\hat{r}_{v_j}$ as the suspiciousness of $s_j$ in terms of its likelihood of containing the bug.
Summary of RBF-based Fault Localization (1)

Train an RBF neural network using the coverage vectors and program execution results.
Summary of RBF-based Fault Localization (2)

Compute the suspiciousness of each statement in $P$ using virtual test cases.
Three Novel Aspects

- Introduce a method for representing test cases, statement coverage, execution results within a modified RBF neural network formalism
  - Training with example test cases and execution results
  - Testing with virtual test cases

- Develop a novel algorithm to simultaneously estimate the number of hidden neurons and their receptive field centers

- Instead of using the traditional Euclidean distance which has been proved to be inappropriate in the fault localization context, a weighted bit-comparison based distance is defined to measure the distance between the statement coverage vectors of two test cases.
  - Estimate the number of hidden neurons and their receptive field centers
  - Compute the output of each hidden neuron
**RBF Example (1)**

- Suppose we have a program with ten statements. Seven test cases have been executed on the program. Table 1 gives the coverage vector and the execution result of each test.

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
<th>$s_6$</th>
<th>$s_7$</th>
<th>$s_8$</th>
<th>$s_9$</th>
<th>$s_{10}$</th>
<th>$r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$t_2$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$t_3$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$t_4$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$t_5$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$t_6$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_7$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$s_1$ is executed by $t_1$  
$s_6$ is not executed by $t_2$  
$t_1$ is a successful test  
$t_6$ is a failed test
**RBF Example (2)**

- An RBF neural network is constructed and trained
  - 10 neurons in the input layer
  - 7 neurons in the hidden layer
  - The field width $\sigma$ is 0.395
  - 1 neuron in the output layer
  - The output layer weights are $\mathbf{w} = [w_1, w_2, w_3, w_4, w_5, w_6, w_7]^T$
    
    $=[-1.326, -0.665, 0.391, -0.378, -0.308, 1.531, 1.381]^T$
**RBF Example (3)**

- Use the **coverage vectors of the virtual test cases** as the inputs to the trained network.

- The output with respect to each statement is the suspiciousness of the corresponding statement.

```
\[
\begin{bmatrix}
  c_{v_1} \\
  c_{v_2} \\
  c_{v_3} \\
  c_{v_4} \\
  c_{v_5} \\
  c_{v_6} \\
  c_{v_7} \\
  c_{v_8} \\
  c_{v_9} \\
  c_{v_{10}}
\end{bmatrix} =
\begin{bmatrix}
  1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

Part (a): Input coverage vectors

```

Part (b): Outputs produced by the trained network which are the suspiciousness of the statements

<table>
<thead>
<tr>
<th>(\hat{r}_{v_1})</th>
<th>(\hat{r}_{v_2})</th>
<th>(\hat{r}_{v_3})</th>
<th>(\hat{r}_{v_4})</th>
<th>(\hat{r}_{v_5})</th>
<th>(\hat{r}_{v_6})</th>
<th>(\hat{r}_{v_7})</th>
<th>(\hat{r}_{v_8})</th>
<th>(\hat{r}_{v_9})</th>
<th>(\hat{r}<em>{v</em>{10}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0384</td>
<td>0.0481</td>
<td>0.1246</td>
<td>0.0768</td>
<td>0.0173</td>
<td>0.0179</td>
<td>0.0157</td>
<td>0.2900</td>
<td>0.0066</td>
<td>0.0782</td>
</tr>
</tbody>
</table>

Highest/ Most suspicious: \(\hat{r}_{v_8}\)

Lowest/ Least suspicious: \(\hat{r}_{v_9}\)
**BP versus RBF**

- Although BP (back propagation) networks are the widely used networks for supervised learning, RBF networks (whose output layer weights are trained in a *supervised* way) are even better in our case because RBF can *learn much faster* than BP networks and do not suffer from pathologies like *local minima* as BP networks do.
**Outline**

- Motivation and Background
- Execution Dice-based Fault Localization
- Suspiciousness Ranking-based Fault Localization
  - Program Spectra-based Fault Localization
  - Code Coverage-based Fault Localization
  - Statistical Analysis-based Fault Localization
  - Neural Network-based Fault Localization
  - Similarity Coefficient-based Fault Localization
- Empirical Evaluation
- Theoretical Comparison: Equivalence
- Mutation-based Automatic Bug Fixing
- Conclusions
DStar – A Similarity Coefficient-based Fault Localization
The Construction of $D^*$ (1)

- The suspiciousness assigned to a statement should be

- **Intuition 1**: directly proportional to the number of failed test cases that cover it
  \[ \text{suspiciousness}(s) \propto N_{CF} \]

- **Intuition 2**: inversely proportional to the number of successful test cases that cover it
  \[ \text{suspiciousness}(s) \propto 1/N_{CS} \]

- **Intuition 3**: inversely proportional to the number of failed test cases that do not cover it
  \[ \text{suspiciousness}(s) \propto 1/N_{UF} \]

- Conveniently enough such a coefficient already exists
  \[ \text{Kulczynski} \] [Kulczynski, 1928]:
  \[ N_{CF}/(N_{CS}+N_{UF}) \]
The Construction of $D^*$ (with $* = 2$)(2)

- However, we also have a fourth intuition …

- **Intuition 4**: Intuition 1 is the most sound of the other intuitions and should therefore carry a higher weight.

- *Kulczynski* does not lead to the realization of the fourth intuition.

- Under the circumstances we might try to do something like this:

  $$\text{suspiciousness}(s) = \frac{2 \times N_{CF}}{N_{UF} + N_{CS}}$$

  or maybe even

  $$\text{suspiciousness}(s) = \frac{100 \times N_{CF}}{N_{UF} + N_{CS}}$$

- But this is not going to help us (as we shall later see)

- So instead we make use of a different coefficient ($D^*$)

  $$\text{suspiciousness}(s) = \frac{N_{CF} \times N_{CF}}{N_{UF} + N_{CS}}$$
\textbf{D* Example : with $*$ = 2 (1)}

- Suppose we are writing a program that computes the sum or average of two numbers.
  - But with respect to the sum computation (statement 5), instead of adding the two numbers, we accidentally subtract them

<table>
<thead>
<tr>
<th>Stmt. #.</th>
<th>Program ( $P$ )</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>read (a);</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>2</td>
<td>read (b);</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>3</td>
<td>read (choice);</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>4</td>
<td>if (choice == “sum”)</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>5</td>
<td>result = a - b; //Correct: a + b;</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>6</td>
<td>else if (choice == “average”)</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>7</td>
<td>result = (a + b) / 2;</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>8</td>
<td>print (result);</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Execution Result (0 = Successful / 1 = Failed) 1 1 0 0 0 0
D* Example: with * = 2 (2)

- Next we collect the statistics we need for D* ($N_{CF}$, $N_{UF}$ and $N_{CS}$)

<table>
<thead>
<tr>
<th>Stmt. #</th>
<th>$N_{CF}$</th>
<th>$N_{UF}$</th>
<th>$N_{CS}$</th>
<th>Suspiciousness based on D* $N_{CF} \times N_{CF}/ (N_{UF} + N_{CS})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Statement ranking: 5, 1, 2, 3, 4, 8, 6, 7

Most suspicious

Tied together

Tied together
Other Fault Localization Techniques
Tarantula, Ochiai, SOBER, & Liblit05

- **Tarantula**

  \[
  \text{suspiciousness}(e) = \frac{\text{failed}(e)}{\text{totalfailed}} \times \frac{\text{passed}(e)}{\text{totalpassed}} + \frac{\text{failed}(e)}{\text{totalfailed}}
  \]

  - \(\text{passed}(e)\) is the number of passed test cases that execute statement \(e\) one or more times
  - \(\text{failed}(e)\) is the number of failed test cases that execute statement \(e\) one or more times
  - \(\text{totalpassed}\) is the total number of test cases that pass in the test suite
  - \(\text{totalfailed}\) is the total number of test cases that fail in the test suite

- **Ochiai**

  \[
  \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}
  \]

- **SOBER**

- **Liblit05**
Outline

• Motivation and Background
• Execution Dice-based Fault Localization
• Suspiciousness Ranking-based Fault Localization
  – Program Spectra-based Fault Localization
  – Code Coverage-based Fault Localization
  – Statistical Analysis-based Fault Localization
  – Neural Network-based Fault Localization
  – Similarity Coefficient-based Fault Localization
• Empirical Evaluation
• Theoretical Comparison: Equivalence
• Mutation-based Automatic Bug Fixing
• Conclusions
Empirical Evaluation
Is a Technique Good at Locating Faults?

“Good” is more of a relative term. We can show a fault localization technique is good by showing that it is more effective than other competing techniques.

We do this via rigorous case studies:
- Using a comprehensive set of subject programs
- Comparing the effectiveness between different fault localization techniques
- Evaluating across multiple criteria

Since it is not possible to theoretically prove that one fault localization technique is always more effective than another, such empirical evaluation is typically the norm.
- We will return to this issue later on
### Subject Programs

- Four sets of subject programs – the *Siemens* suite, the *Unix* suite, *gzip* and *Ant* – were used (19 different programs in all – C & Java)
  - Two additional programs (*grep* and *make*) are also used which makes a total of 21 programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of Code</th>
<th>Number of faulty versions used†</th>
<th>Number of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>565</td>
<td>5</td>
<td>4130</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>510</td>
<td>10</td>
<td>4115</td>
</tr>
<tr>
<td>schedule</td>
<td>412</td>
<td>9</td>
<td>2650</td>
</tr>
<tr>
<td>schedule2</td>
<td>307</td>
<td>9</td>
<td>2710</td>
</tr>
<tr>
<td>replace</td>
<td>563</td>
<td>32</td>
<td>5542</td>
</tr>
<tr>
<td>tcas</td>
<td>173</td>
<td>41</td>
<td>1608</td>
</tr>
<tr>
<td>tot_info</td>
<td>406</td>
<td>23</td>
<td>1052</td>
</tr>
<tr>
<td>cal</td>
<td>202</td>
<td>20</td>
<td>162</td>
</tr>
<tr>
<td>checkeq</td>
<td>102</td>
<td>20</td>
<td>166</td>
</tr>
<tr>
<td>col</td>
<td>308</td>
<td>30</td>
<td>156</td>
</tr>
<tr>
<td>comm</td>
<td>167</td>
<td>12</td>
<td>186</td>
</tr>
<tr>
<td>crypt</td>
<td>134</td>
<td>14</td>
<td>156</td>
</tr>
<tr>
<td>look</td>
<td>170</td>
<td>14</td>
<td>193</td>
</tr>
<tr>
<td>sort</td>
<td>913</td>
<td>21</td>
<td>997</td>
</tr>
<tr>
<td>spline</td>
<td>338</td>
<td>13</td>
<td>700</td>
</tr>
<tr>
<td>tr</td>
<td>137</td>
<td>11</td>
<td>870</td>
</tr>
<tr>
<td>uniq</td>
<td>143</td>
<td>17</td>
<td>431</td>
</tr>
<tr>
<td>gzip</td>
<td>6573</td>
<td>28</td>
<td>211</td>
</tr>
<tr>
<td>Ant</td>
<td>75333</td>
<td>23</td>
<td>871</td>
</tr>
</tbody>
</table>

† Some versions were created using mutation-based fault injection
Techniques $D^*$ is Compared to

- First compared $D^*$ to the Kulczynski coefficient

- Also compared it with 11 other well-known coefficients forming a baker’s dozen [Choi et al. 2010, Willett 2003]
  (1) Simple-Matching
  (2) BraunBanquet
  (3) Dennis
  (4) Mountford
  (5) Fossum
  (6) Pearson ($\chi^2$)
  (7) Gower
  (8) Michael
  (9) Pierce
  (10) Baroni-Urbani/Buser
  (11) Tarwid

- Further comparisons with other techniques were also performed
  - To be discussed later
Three Evaluation Metrics/Criteria

• Number of statements examined
  – The number of statements that need to be examined by D* to locate faults versus other techniques
  – An absolute measure

• The EXAM score: the percentage of code examined
  – The percentage of code that needs to be examined by using D* to locate faults versus other techniques
  – A relative (graphical) measure

• The Wilcoxon Signed-Rank Test
  – Evaluate the alternative hypothesis that other techniques will require the examination of more statements than D*
    □ D* is more effective than other techniques
    □ Null hypothesis being that the other techniques require the examination of a number of statements that is less than or equal to that required by D*
  – A statistical measure
**Ties in the Ranking: Best/Worst**

- The suspiciousness assigned to a statement by D* (and other techniques) may not be unique, i.e., two or more statements can be tied for the same position in the ranking.

  From our example: Statement ranking: \(5, 1, 2, 3, 4, 8, 6, 7\)

  Tied together

- Assuming a faulty statement and some correct statements are tied
  - In the *best* case we examine the faulty statement *first*
  - In the *worst* case we examine it *last*

- For each of the previously discussed evaluation criteria, we will have the *best case* and the *worst case* effectiveness.
  - Presenting only the *average* would have resulted in a loss of information
**Results – Total Number of Statements Examined**

<table>
<thead>
<tr>
<th>Fault Localization Technique</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Siemens</td>
<td>Unix</td>
</tr>
<tr>
<td>D*</td>
<td>1754</td>
<td>1805</td>
</tr>
<tr>
<td>Kulcynzki</td>
<td>2327</td>
<td>2358</td>
</tr>
<tr>
<td>Simple-Matching</td>
<td>6335</td>
<td>5545</td>
</tr>
<tr>
<td>BraunBanquet</td>
<td>2438</td>
<td>2767</td>
</tr>
<tr>
<td>Dennis</td>
<td>2206</td>
<td>2934</td>
</tr>
<tr>
<td>Mountford</td>
<td>1974</td>
<td>2183</td>
</tr>
<tr>
<td>Fossum</td>
<td>2230</td>
<td>2468</td>
</tr>
<tr>
<td>Pearson</td>
<td>3279</td>
<td>3581</td>
</tr>
<tr>
<td>Gower</td>
<td>6586</td>
<td>8630</td>
</tr>
<tr>
<td>Michael</td>
<td>1993</td>
<td>3713</td>
</tr>
<tr>
<td>Pierce</td>
<td>8072</td>
<td>11782</td>
</tr>
<tr>
<td>Baroni-Urbani/Buser</td>
<td>3547</td>
<td>3189</td>
</tr>
<tr>
<td>Tarwid</td>
<td>2453</td>
<td>3399</td>
</tr>
</tbody>
</table>

- D* is very consistent in its performance
- Often the worst case of D* is better than the best case of the other techniques (Note that * = 2)
Results – EXAM Score (Siemens suite)

D* is clearly the most effective
Results – EXAM Score (Unix suite)

D* is clearly the most effective
Results – EXAM Score (Gzip)

D* is clearly the most effective
Results – EXAM Score (Ant)

D* is clearly the most effective
Generally the confidence with which we can claim that \( D^* \) is more effective than the other techniques is very high (easily over 99%).

But there are a few exceptions.

Why? Perhaps this has something to do with the way our hypothesis was constructed.

**Results – Wilcoxon Signed-Rank Test (1)**

<table>
<thead>
<tr>
<th>Fault Localization Technique</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Siemens</td>
<td>Unix</td>
</tr>
<tr>
<td>Kulczynski</td>
<td>99.99%</td>
<td>99.99%</td>
</tr>
<tr>
<td>Simple-Matching</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>BraunBanquet</td>
<td>99.99%</td>
<td>100%</td>
</tr>
<tr>
<td>Dennis</td>
<td>99.99%</td>
<td>100%</td>
</tr>
<tr>
<td>Mountford</td>
<td>99.99%</td>
<td>99.99%</td>
</tr>
<tr>
<td>Fossum</td>
<td>100%</td>
<td>99.99%</td>
</tr>
<tr>
<td>Pearson</td>
<td>100%</td>
<td>99.99%</td>
</tr>
<tr>
<td>Gower</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Pierce</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Baroni-Urbani/Buser</td>
<td>99.99%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Results – Wilcoxon Signed-Rank Test (2)

• Let us modify our alternative hypothesis to consider equalities.
  – We now evaluate to see if $D^*$ is more effective than, or at least as effective as, the other techniques.
  – Which is to say $D^*$ requires the examination of a number of statements that is less than or equal to that required by the other techniques.

<table>
<thead>
<tr>
<th>Fault Localization Technique</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gzip</td>
<td>Ant</td>
</tr>
<tr>
<td>Kulcynski</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Simple-Matching</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>BraunBanquet</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Dennis</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Mountford</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Fossum</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Pearson</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Baroni-Urbani/Buser</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Confidence levels have gone up significantly. All entries but one are greater than 95%. D* is clearly the most effective
More Discussion on $D^*$

- $D^*$ with a higher value for the *
- Compare $D^*$ with other fault localization techniques
Effectiveness of D*  

- The effectiveness of D* for the *make* program increases until it levels off as the value of * increases.

- A similar observation also applies to other programs.

![Graph showing the effectiveness of D*]
### Effectiveness of Other Fault Localization Techniques

- The best- and worst-case effectiveness of 18 fault localization techniques (excluding D*) on 21 different programs.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unix</td>
<td>1655</td>
<td>5026</td>
</tr>
<tr>
<td>Simens</td>
<td>1396</td>
<td>2292</td>
</tr>
<tr>
<td>grep</td>
<td>2702</td>
<td>4435</td>
</tr>
<tr>
<td>gzip</td>
<td>1535</td>
<td>3312</td>
</tr>
<tr>
<td>make</td>
<td>8553</td>
<td>14272</td>
</tr>
<tr>
<td>Ant</td>
<td>1320</td>
<td>1882</td>
</tr>
<tr>
<td>H3c</td>
<td>1701</td>
<td>5072</td>
</tr>
<tr>
<td>H3b</td>
<td>1439</td>
<td>2335</td>
</tr>
<tr>
<td>gzip</td>
<td>3019</td>
<td>4752</td>
</tr>
<tr>
<td>make</td>
<td>1535</td>
<td>3313</td>
</tr>
<tr>
<td>Ant</td>
<td>10817</td>
<td>16556</td>
</tr>
<tr>
<td>RBF</td>
<td>10701</td>
<td>5779</td>
</tr>
<tr>
<td>Ochiai</td>
<td>1272</td>
<td>3186</td>
</tr>
<tr>
<td>Crosstab</td>
<td>1270</td>
<td>4758</td>
</tr>
<tr>
<td>Tarantula</td>
<td>10305</td>
<td>6094</td>
</tr>
<tr>
<td>Kulpynzki</td>
<td>887</td>
<td>2873</td>
</tr>
<tr>
<td>Simple-Matching</td>
<td>12403</td>
<td>3091</td>
</tr>
<tr>
<td>BraunBanquet</td>
<td>1314</td>
<td>18142</td>
</tr>
<tr>
<td>Dennis</td>
<td>2005</td>
<td>14272</td>
</tr>
<tr>
<td>Mountford</td>
<td>4005</td>
<td>1320</td>
</tr>
<tr>
<td>Fossum</td>
<td>1314</td>
<td>4435</td>
</tr>
<tr>
<td>Pearson</td>
<td>1535</td>
<td>4435</td>
</tr>
<tr>
<td>Gower</td>
<td>1535</td>
<td>4435</td>
</tr>
<tr>
<td>Michael</td>
<td>3092</td>
<td>6094</td>
</tr>
<tr>
<td>Pierce</td>
<td>1270</td>
<td>2873</td>
</tr>
<tr>
<td>Baroni-Urbani/Buser</td>
<td>1270</td>
<td>3091</td>
</tr>
<tr>
<td>Tarwid</td>
<td>1270</td>
<td>3091</td>
</tr>
</tbody>
</table>

Jump to Slide 105
Comparison between $D^*$ and Other Techniques

• The effectiveness of $D^2$ is better than the other 12 similarity coefficient-based fault localization techniques.

• From the following table, we also observe that $D^*$ (with an appropriate value of *) performs better than other fault localization techniques, regardless of the subject programs, and the best- or worst-case.

  – The cell with a black background gives the smallest * such that $D^*$ outperforms others.

<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unix</td>
<td>Simens</td>
</tr>
<tr>
<td>$D^2$</td>
<td>1805</td>
<td>1754</td>
</tr>
<tr>
<td>$D^3$</td>
<td>1667</td>
<td>1526</td>
</tr>
<tr>
<td>$D^4$</td>
<td>1594</td>
<td>1460</td>
</tr>
<tr>
<td>$D^5$</td>
<td>1507</td>
<td>1435</td>
</tr>
<tr>
<td>$D^*$</td>
<td>1386</td>
<td>2693</td>
</tr>
<tr>
<td>H3b</td>
<td>1701</td>
<td>1439</td>
</tr>
<tr>
<td>H3c</td>
<td>1655</td>
<td>1396</td>
</tr>
<tr>
<td>Tarantula</td>
<td>3394</td>
<td>2453</td>
</tr>
<tr>
<td>Ochiai</td>
<td>1906</td>
<td>1796</td>
</tr>
</tbody>
</table>
Outline

• Motivation and Background
• Execution Dice-based Fault Localization
• Suspiciousness Ranking-based Fault Localization
  – Program Spectra-based Fault Localization
  – Code Coverage-based Fault Localization
  – Statistical Analysis-based Fault Localization
  – Neural Network-based Fault Localization
  – Similarity Coefficient-based Fault Localization
• Empirical Evaluation
• Theoretical Comparison: Equivalence
• Mutation-based Automatic Bug Fixing
• Conclusions
Theoretical Comparison: Equivalence
As discussed earlier the general norm for comparing fault localization techniques has been to use *empirical data*. If technique $\alpha$ is better than technique $\beta$, then it should lead programmers to the location of fault(s) faster than $\beta$. Multiple metrics have been proposed to do this such as the ones used in our research. Case studies can be quite expensive and time-consuming to perform. Often a lot of data has to be analyzed.

But is empirical comparison always required...especially when trying to show that two techniques will be equally effective?
Comparing Fault Localization Techniques (2)

- Note that the suspiciousness of a statement is irrelevant from an absolute sense.
  - It only matters how the suspiciousness of two (or more) statements compare with respect to each other (i.e., relative to one another).

- Supposing we have two statements $s_1$ and $s_2$ with suspiciousness values of 5 and 6, respectively. This means that $s_2$ is ranked above $s_1$ as it is more suspicious.

- However, $s_2$ would still be ranked above $s_1$ if the suspiciousness values were 6 and 7, or 50 and 60, respectively – the relative ordering of $s_1$ and $s_2$ is still maintained.

- Thus, subtracting the same constant from (or adding it to) the suspiciousness of every statement will have no effect on the final ranking. The same applies for multiplication/division operations.
Recall the suspiciousness computation of Kulczynski

\[
\text{suspiciousness}(s) = \frac{N_{CF}}{N_{UF} + N_{CS}}
\]

It now becomes clear that an identical ranking will be produced by

\[
\text{suspiciousness}(s) = \left(\frac{N_{CF}}{N_{UF} + N_{CS}}\right) + 1 \quad \text{or} \quad \text{suspiciousness}(s) = \left(\frac{N_{CF}}{N_{UF} + N_{CS}}\right) \times 10
\]

This is why D* was constructed the way it was

Any operation that is order-preserving can be safely performed on the suspiciousness function without changing the ranking.

If the ranking does not change...then the effectiveness will not change either. \textit{We can exploit this!}
Consider a program $P$ with a set of elements $M$. Let $\text{rank}(r,s)$ be a function that returns the position of statement $s$ in ranking $r$.

Two rankings $r_\alpha$ and $r_\beta$ (produced by using two techniques $L_\alpha$ and $L_\beta$ on the same input data) are equal if

- $\forall s \in M, \text{rank}(r_\alpha,s) = \text{rank}(r_\beta,s)$.
- Two rankings are equal if for every statement, the position is the same in both rankings.

If two fault localization techniques $L_\alpha$ and $L_\beta$ always produce rankings that are equal, then the techniques are said to be equivalent, i.e., $L_\alpha \equiv L_\beta$ and therefore will always be equally as effective (at fault localization).

So is the equivalence relation useful?

Certainly! In at least two scenarios it holds great potential

- Eliminating the need for time-consuming case studies.
- Making suspiciousness computations more efficient.
Eliminating the Need for Case Studies (1)

- Take the example of [Abreu et al. 2009] where
  - The authors use of the Ochiai coefficient to compute suspiciousness.
  - The coefficient is compared to several other coefficients \textit{empirically}.
  - Among others, it is compared to the Jaccard and Sorensen-Dice coefficients.
- We posit that this was unnecessary, as per the equivalence relation.

\[
suspiciousness(s) = \frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS}} \quad \text{Jaccard} \\
\]

\[
suspiciousness(s) = \frac{2N_{CF}}{2N_{CF} + N_{UF} + N_{CS}} \quad \text{Sorensen-Dice}
\]

- Via a set of order-preserving operations, both can be reduced to:

\[
suspiciousness(s) = \frac{N_{CF}}{N_{UF} + N_{CS}} \quad \text{Jaccard} \equiv \text{Sorensen-Dice}
\]

**Eliminating the Need for Case Studies (2)**

- As it turns out the coefficient *Anderberg* also evaluates to the same form. Ochiai was empirically compared to Anderberg.

\[
\text{Jaccard} \equiv \text{Sorensen-Dice} \equiv \text{Anderberg}
\]

- In fact the authors also compared Ochiai to the *SimpleMatching* and *Rogers and Tanimoto* coefficients, the both of which are also equivalent to one another.

\[
\text{SimpleMatching} \equiv \text{Rogers and Tanimoto}
\]

Such redundant comparisons could have been avoided by making use of the fault localization equivalence relation.
Making Computations More Efficient (1)

- As shown, if Jaccard were the chosen fault localization technique, using the suspiciousness function

\[
suspiciousness(s) = \frac{N_{CF}}{N_{CF} + N_{UF} + N_{CS}}
\]

would give the same results as using

\[
suspiciousness(s) = \frac{N_{CF}}{N_{UF} + N_{CS}}
\]

- We should go with the simplest computation as it is expected to be faster.
We performed an additional case study on the 7 programs of the Siemens suite.

Observed the relative time saved in computing suspiciousness for all the statements in a faulty program, by using the simplified form of Jaccard ($J^*$) as opposed to the original ($J$).

- The quantity ($J-J^*$) represents the computational time that is saved.
- $((J-J^*)/J) \times 100\%$ represents the relative time saved, i.e., efficiency gained.

- 100 trials were performed per faulty version.

- Difference in times was computed to nanosecond precision.
### Making Computations More Efficient (3)

<table>
<thead>
<tr>
<th>Programs</th>
<th>Average Percentage Time Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>35.37%</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>39.21%</td>
</tr>
<tr>
<td>schedule</td>
<td>44.62%</td>
</tr>
<tr>
<td>schedule2</td>
<td>49.74%</td>
</tr>
<tr>
<td>replace</td>
<td>41.65%</td>
</tr>
<tr>
<td>tcas</td>
<td>52.46%</td>
</tr>
<tr>
<td>tot_info</td>
<td>47.68%</td>
</tr>
</tbody>
</table>

- The savings in terms of time are quite significant.

- Using the equivalence relation can thus, help reduce techniques to simplified forms, thereby greatly increasing efficiency.
Programs with Multiple Faults
Programs with Multiple Faults

• One bug at a time

• A good approach is to use "fault-focused" clustering.
  – Divide failed test cases into clusters that target different faults
  – Failed test cases in each fault-focused cluster are combined with the successful tests for debugging a single fault.
Outline

• Motivation and Background
• Execution Dice-based Fault Localization
• Suspiciousness Ranking-based Fault Localization
  – Program Spectra-based Fault Localization
  – Code Coverage-based Fault Localization
  – Statistical Analysis-based Fault Localization
  – Neural Network-based Fault Localization
  – Similarity Coefficient-based Fault Localization
• Empirical Evaluation
• Theoretical Comparison: Equivalence
• **Mutation-based Automatic Bug Fixing**
• Conclusions
Mutation-based Automatic Bug Fixing

Mutation as a Fault Generation Aid

• For research experiments, large comprehensive data sets are rarely available

• Need faulty versions of programs to perform all kinds of experiments on, but don’t always have a way to get them

• Recently many researchers have relied on mutation
  – Mutants generated can represent realistic faults
  – Experiments that use these mutants as faulty versions can yield trustworthy results
  – As opposed to seeding faults, mutant generation is automatic
Mutation as a Fault Fixing Aid?

If mutating a correct program can produce a realistic fault, can mutating an incorrect program produce a realistic fix?

• Supposing we wanted to write program $P$

• But we ended up writing a faulty program $P'$
  – We know $P'$ is faulty because at least one test case in our test set results in failure when executed on $P'$

• Mutate $P'$ to get $P''$

• If $P'' = P$ … we automatically fixed the fault in $P'$
## Our Solution

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Fault Localization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>The Good</strong>: Can result in potential fixes for faulty programs automatically.</td>
<td><strong>The Good</strong>: Can potentially identify the location of a fault in a program.</td>
</tr>
<tr>
<td><strong>The Bad</strong>: We have no idea as to where in a program a fault is, and so we do not know how to proceed. Randomly examining mutants can be prohibitively expensive.</td>
<td><strong>The Bad</strong>: Even if we locate the fault, we have no idea as to how to fix the fault. This is left solely as the responsibility of the programmers/debuggers.</td>
</tr>
</tbody>
</table>

*So…what if we combined the two?*
Conclusion
**What We Have Discussed**

- **Existing and new fault localization techniques**
  - Many of them use the same information (statement coverage and execution results) to identify suspicious code likely to contain program bug(s)

- **A strategy to automatically suggest fixes for faults** that
  - makes as few assumptions as possible about the software being debugged
  - is generally applicable to different types of software and programming languages
  - still manages to produce some useful information even when it is unable to fix faults automatically

*Present a framework to automate the debugging process.*