SemFix
Research Question

Can we find bugs existing within code?

Can code be analyzed to automatically find and suggest bug fixes?

Can this effectively reduce costs of development and production?
Contributions

While finding bugs automatically is good, fixing bugs automatically is better.

Saves developers time by automatically producing bug fixes based on
SemFix Process

1. Fault Isolation
2. Statement-level specification inference
3. Program Synthesis
Fault Isolation

A list is made from every “suspicious” line in the code and ranked accordingly.

Suspiciousness of a line is how often the line is executed in successful and failing executions. Greater the number of failures, greater the score.
Program Synthesis

- Does not blindly test combinations of components
- Utilizes the repair constraint to make smart decisions
- Only tests on a subset of the test suite to improve scalability and efficiency. Initially, only the failing tests are retested
- Iteratively synthesises more complicated statements to find the simpler solutions earlier
- Simplifies solutions to make them more readable
Example of Fault Localization & Ranking

```c
int is_upward_preferred(int inhibit, int up_sep, int down_sep) {
    int bias;
    if (inhibit)
        bias = down_sep; // fix: bias = up_sep + 100
    else
        bias = up_sep;
    if (bias > down_sep)
        return 1;
    else
        return 0;
}
```

<table>
<thead>
<tr>
<th>Test</th>
<th>inhibit</th>
<th>up_sep</th>
<th>down_sep</th>
<th>Expected output</th>
<th>Observed output</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
<td>110</td>
<td>1</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>pass</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-20</td>
<td>60</td>
<td>1</td>
<td>0</td>
<td>fail</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>pass</td>
</tr>
</tbody>
</table>

```

<table>
<thead>
<tr>
<th>Line</th>
<th>Score</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.75</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0.6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0.5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>
Repair Algorithm

```plaintext
1: Input:
2: P : The buggy program
3: T : A test suite
4: RC : A ranked list of potential bug root-cause
5: Output:
6: r: A repair for P
7: 
8: while RC is not EMPTY and not TIMEOUT do
9:  rc = Shift(RC) // A repair candidate
10:  S = ∅       // A test suite for repair generation
11:  T_f = ExtractFailedTests(T, P);
12:  while T_f ≠ ∅ do
13:      S = S ∪ T_f
14:      new_repair = Repair(P, S, rc)
15:      if new_repair == null then
16:          break
17:      end if
18:  end while
19:  P' = ApplyRepair(P, new_repair)
20:  T_f = ExtractFailedTests(T, P');
21: while new_repair not null then
22:      return new_repair
23:  end if
24: end while

26: function Repair(P, S, rc)
27:  C = GenerateRepairConstraint(P, S, rc);
28:  level = 1   // The complexity of a repair
29:  new_repair = Synthesize(C, level);
30:  while new_repair == null and level ≤ MAX_LEVEL do
31:      level = level + 1
32:      new_repair = Synthesize(C, level);
33:  end while
34:  return new_repair
```

Increase Synthesis Complexity
SemFix versus Genprog

SemFix: 48/90 fixes
Genprog: 16/90 fixes

<table>
<thead>
<tr>
<th>Bug type</th>
<th>Const</th>
<th>Arith</th>
<th>Comp</th>
<th>Logic</th>
<th>Code Missing</th>
<th>Redundant Code</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>14</td>
<td>14</td>
<td>16</td>
<td>10</td>
<td>27</td>
<td>9</td>
<td>90</td>
</tr>
<tr>
<td>SemFix</td>
<td>10</td>
<td>6</td>
<td>12</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>48</td>
</tr>
<tr>
<td>GenProg</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcas</td>
<td>38 / 24</td>
<td>38 / 19</td>
<td>35 / 16</td>
<td>34 / 12</td>
<td>34 / 11</td>
</tr>
<tr>
<td>Schedule</td>
<td>5 / 1</td>
<td>3 / 1</td>
<td>4 / 1</td>
<td>4 / 0</td>
<td>4 / 0</td>
</tr>
<tr>
<td>Schedule2</td>
<td>4 / 4</td>
<td>3 / 2</td>
<td>4 / 2</td>
<td>3 / 3</td>
<td>2 / 1</td>
</tr>
<tr>
<td>Replace</td>
<td>7 / 6</td>
<td>7 / 5</td>
<td>8 / 5</td>
<td>7 / 6</td>
<td>6 / 4</td>
</tr>
<tr>
<td>Grep</td>
<td>2 / 0</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>2 / 0</td>
<td>2 / 0</td>
</tr>
<tr>
<td>Total</td>
<td>56 / 35</td>
<td>52 / 27</td>
<td>52 / 24</td>
<td>50 / 21</td>
<td>48 / 16</td>
</tr>
</tbody>
</table>
Time Analysis

SemFix was roughly twice as fast as GenProg at running through its algorithm for every size test suite.
Effectiveness Over Project Scale

**SUCCESS OF REPAIR (SIR)**

- **Time bound = 4 mins.**

![Graph showing success of repair over project scale.](image)

- **Total**
- **Semfix**
- **GenProg**

- **Overall** 90 programs from SIR
  - SemFix repaired 48/90, GenProg repaired 16/90 for 50 tests.

- **GenProg running time is >3 times of SemFix**
Fallbacks of SemFix

The types of bugs not fixed by SemFix are divided into 5 groups:

• SemFix puts a bound on the number of times it symbolically executes in loops to avoid infinite loops.
• SemFix does not work well with arrays.
• Symbolic execution does not consider floating-point variables.
• The repair is not a 1-line fix.
• Other unknown and unidentified reasons.
Evaluation Questions

How effective is SemFix at fixing non trivial bugs?

SemFix can only be as effective as its test suite.

Overhead

<table>
<thead>
<tr>
<th>Subject</th>
<th>Tcas</th>
<th>Schedule</th>
<th>Schedule2</th>
<th>Replace</th>
<th>Grep</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratio</td>
<td>6.9</td>
<td>2.8</td>
<td>2.5</td>
<td>1.36</td>
<td>2.2</td>
<td>4.16</td>
</tr>
</tbody>
</table>

Known problems Multi line fixes, floating point errors, wrong function calls, is this too many flaws?
Discussion Questions

1. How would SemFix perform on a less than perfect test suite?

2. If this test suite had 1 failing test, and that test was written incorrectly, what would SemFix produce?
Discussion Questions

3. How does SemFix scale compared to the size of the Test Suite?

4. What would you consider to be the hardest category of bugs to fix?
Discussion Questions

5. What would be the most useful category of bugs to be automatically fixed?

6. How can SemFix use explicit specification of program behavior to more effectively repair a larger set of bugs?
References