SemFix: Program Repair via Semantic Analysis

presenter name(s) removed for FERPA considerations
Key Idea

Create a better and faster semantic based program repair tool to generate bug fixes than the known search based program repair techniques.

Locate faults using statistical analysis.

Create repair constraints using symbolic execution.

Constraint solved by iterating over a layered space of repair expressions.
Study Questions

Does SemFix successfully modify the program such that semantic analysis is comparable to genetic repair?

Can SemFix generate a valid repair even if the repair code does not exist anywhere in the program?
How it works - SemFix

Fault isolation

Tarantula

Statistical fault localization

Ranked by suspiciousness rating

Statement-level specification inference

KLEE

Symbolic execution to create repair constraints

Convert expression to non-deterministic
Fault Isolation

Uses Tarantula to rank suspicious statements against test suite

Ranking is formed statistically according to \( susp(s) \) function

\[
susp(s) = \frac{\text{failed}(s)/\text{total failed}}{\text{passed}(s)/\text{total passed} + \text{failed}(s)/\text{total failed}}
\]

\( \text{failed}(s) \) & \( \text{passed}(s) \): failing and passing executions where statement \( s \) occurs

\( \text{total failed} \) & \( \text{total passed} \): total failing and passing executions
Statement-level Specification Inference

Generating repair constraints - given input and output pairs

Uses KLEE: a static symbolic execution engine

In SemFix context - variable directly affected by potential defect, treated as a symbol

Values of accessible variables gathered at location of new symbols

Collect the path condition as well as the symbolic output

Generates repair constraint, for each explored path
Program Synthesis

Implemented in Perl

Component-based synthesis

Conjoin constraints into well-formed constraint

Corresponds to function f, a valid repair

Valid repairs are constructed incrementally according to the valid repair

Starting with constants

If it fails, moves to the next level of components

On pass tested against entire test suite again
**Code**

```c
1 int is_upward_preferred(int inhibit, int up_sep,
   int down_sep) {
2    int bias;
3    if(inhibit)
4       bias = down_sep; // fix: bias = up_sep + 100
5    else
6       bias = up_sep;
7    if (bias > down_sep)
8       return 1;
9    else
10       return 0;
11 }
```

Fig. 1. Code excerpt from Tcas

**Constraint**

\[ f(1,11,110) > 110 \land f(1,0,100) \leq 100 \land f(1,-20,60) > 60 \]
Algorithm 1 Repair algorithm

1: Input:
2: $P$: The buggy program
3: $T$: A test suite
4: $RC$: A ranked list of potential bug root-causes
5: Output:
6: $r$: A repair for $P$
7: 
8: while $RC$ is not EMPTY and not TIMEOUT do
9:     $rc = \text{Shift}(RC)$ // A repair candidate
10:     $S = \emptyset$ // A test suite for repair generation
11:     $T_f = \text{ExtractFailedTests}(T, P)$
12:     while $T_f \neq \emptyset$ do
13:         $S = S \cup T_f$
14:         new_repair = Repair($P, S, rc$)
15:     end while
16:     if new_repair == null then
17:         break
18:     end if
19:     $P' = \text{ApplyRepair}(P, \text{new_repair})$
20:     $T_f = \text{ExtractFailedTests}(T, P')$
21: end while
22: if new_repair not null then
23:     return new_repair
24: end if
25: 
26: function Repair($P, S, rc$)
27:     $C = \text{GenerateRepairConstraint}(P, S, rc)$
28:     $level = 1$ // The complexity of a repair
29:     new_repair = $\text{Synthesize}(C, level)$
30:     while new_repair == null and level $\leq$ MAX_LEVEL do
31:         $level = level + 1$
32:         new_repair = $\text{Synthesize}(C, level)$
33:     end while
34:     return new_repair
35: end function
Comparison with GenProg

SemFix outperforms GenProg in all programs

Except Schedule2 - most bugs are code missing bugs

Runs faster

GenProg has to search, compile and test program variants

Higher success rate
<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>

Const: wrong constant
Arith: wrong arithmetic expression
Comp: wrong comparison operator
Logic: wrong logic operator
Why Repair using Semantic Analysis?

Advantages

Faster than genetic automatic repair (GenProg)

Doesn’t have to search or compile separate programs variants

Faster than enumeration repair synthesis

Doesn’t require a given specification (program sketching)

Uses symbolic execution for checking possible expression modifications

Similar to Angelic Debugging

Disadvantages
Discussion

1. How might the choice of statistical debugging metrics affect SemFix?
   a. Instead of Tarantula

2. Is this effective enough when it does not target vulnerabilities outside of the given test cases?

3. Would you personally use this method of automatic program repair? Why or why not?

4. Are there any situations where SemFix may not be ideal to use over another program repair method?

5. Can this possibly be applied, at some capacity, for complete sub-routine (or even...