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

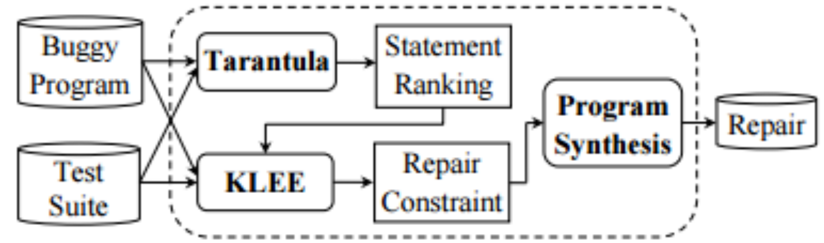

Fig. 3. Architecture of SEMFIX tool.

Fault Isolation

Uses Tarantula to rank suspicious statements against test suite

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

failed(s) & *passed(s)*: failing and passing executions where statement s occurs

totalfailed & *totalpassed:* total failing and passing executions

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

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 accord

- Starting with constants
- If it fails, moves to the next level of components
- On pass tested against entire test suite again

TABLE III
THE CATEGORIZATION OF BASIC COMPONENTS

Level	Conditional Statement	Assign Statement
1	Constants	Constants
2	Comparison ($>, \geq, =, \neq$)	Arithmetic ($+, -$)
3	Logic ($\wedge, \vee$)	Comparison, Ite
4	Arithmetic ($+, -$)	Logic
5	Ite, Array Access	Array Access
6	Arithmetic ($*$)	Arithmetic ($*$)

Code

```
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;
}
```

Fig. 1. Code excerpt from `Tcas`

$$f(1,11,110) > 110 \wedge f(1,0,100) \leq 100 \wedge f(1,-20,60) > 60$$

Constraint

Algorithm 1 Repair algorithm

```
Input:
P : The buggy program
T : A test suite
RC : A ranked list of potential bug root-cause
Output:
r: A repair for P

while RC is not EMPTY and not TIMEOUT do
    rc = Shift(RC) // A repair candidate
    S = ∅          // A test suite for repair generation
    T_f = ExtractFailedTests(T, P);
    while T_f ≠ ∅ do
        S = S ∪ T_f
        new_repair = Repair(P, S, rc)
        if new_repair == null then
            break
        end if
        P′ = ApplyRepair(P, new_repair)
        T_f = ExtractFailedTests(T, P′);
    end while
    if new_repair not null then
        return new_repair
    end if
end while

function Repair(P, S, rc)
    C = GenerateRepairConstraint(P, S, rc);
    level = 1 // The complexity of a repair
    new_repair = Synthesize(C, level);
    while new_repair == null and level ≤ MAX_LEVEL do
        level = level + 1
        new_repair = Synthesize(C, level);
    end while
    return new_repair
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 V
COMPARING THE SUCCESS RATE BETWEEN SEMFIX (SF) AND GENPROG (GP). X IN [X] ON THE TOP OF EACH COLUMN DENOTES THE NUMBER OF TESTS.

Program	[10] SF/GP	[20] SF/GP	[30] SF/GP	[40] SF/GP	[50] SF/GP
Tcas	38 / 24	38 / 19	35 / 16	34 / 12	34 / 11
Schedule	5 / 1	3 / 1	4 / 1	4 / 0	4 / 0
Schedule2	4 / 4	3 / 2	4 / 2	3 / 3	2 / 1
Replace	7 / 6	7 / 5	8 / 5	7 / 6	6 / 4
Grep	2 / 0	1 / 0	1 / 0	2 / 0	2 / 0
Total	56 / 35	52 / 27	52 / 24	50 / 21	48 / 16

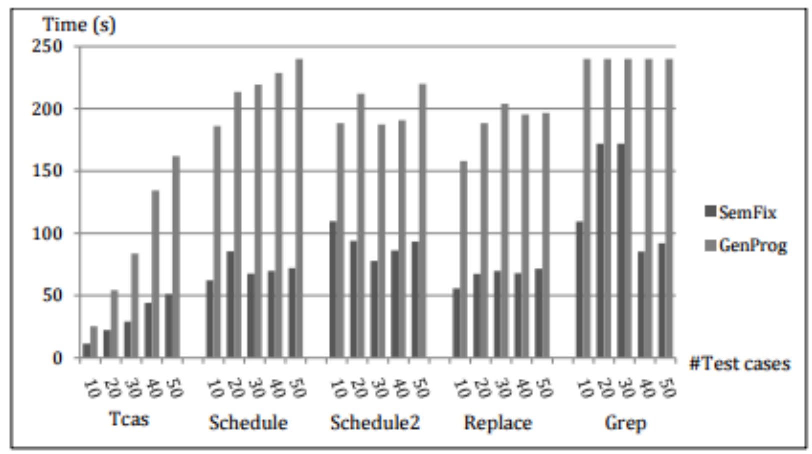

TABLE VI

SEMFIX (SF) VS. GENPROG (GP) IN REPAIRING DIFFERENT CLASS OF BUGS WITH 50 TESTS.

Bug type	Const	Arith	Comp	Logic	Code Missing	Redundant Code	All
Total	14	14	16	10	27	9	90
SemFix	10	6	12	10	5	5	48
GenProg	3	0	5	3	3	2	16

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