Motivating Example

- Listing 1 contains a buffer overflow bug that can be fixed using our tool.
- The code comments on line 7 indicate that buffer overflow is possible at this point.

Steps used in Our Approach

1. Input Saturation: The input saturation principle consists of limiting the possible values that the buffer index variable can take.
2. SMT Constraint System Used for Bug Detection:
   - (set = logic ALPPEIRNA)
   - (declare - fun f (int) int)
   - (assert (- x (* 2 *)) 0)
   - (assert (= F ARRAY_SIZE) 0)
   - (assert (= i BUFFER_INDEX) 0)
3. Bug Type Classification: How to classify the bug?
4. Bug Detection Time: Is there a bug?
5. Repair Time: How to fix the bug?
6. Our Approach: Parameterized SMT-C code patches. This approach is generalizable and can be applied to other bug checkers that we have developed.

Results I

- Figure 2 presents the results of running our tool on 19 "memcpy" programs contained in the open source Juliet test suite. The test suite contains CWE-121 test cases.
- Figure 3 shows the total patch generation overhead.

Results II

- Figure 4 illustrates the results of running our tool on 19 "memcpy" programs contained in the open source Juliet test suite.
- Figure 5 presents the total patch generation overhead.

Contributions

1. An algorithm for generation of "in-place" and "not in-place" bug fixes
2. A novel approach for bug fix generation based on input saturation
3. Semi-automated patch insertion based on source files differential views
4. Automated check for behavior preserving of the patched program

Algorithm

Input:
- Slottable program execution paths set \( \Gamma \)
- Control Flow Variant (CFV) set \( \mathcal{C} \)
- Repair Inference: How to fix the bug?
- Failure Detection: Is there a bug?

Output:
- In-place quick fix generation algorithm
- Not in-place quick fix generation algorithm

Experiments

- Research Questions:
  1. How do the generated patches work for bug fixing?
  2. What is the overall computational overhead of our tool?
  3. Which types of generated fixes are behavior preserving?

- Methodology:
  We evaluate our approach on 38 C open source programs contained in the Juliet test suite CWE-121.

- Setup:
  For testing purposes we used a system having an 84-04 Linux kernel 3.13.0-32-GIT, Intel X5-2320 CPU @ 2.60GHz 𝑛 + 4

References


Table 1: Bug detection and patches generation results

<table>
<thead>
<tr>
<th>Test Programs</th>
<th>LOC</th>
<th>Patches</th>
<th>Time (s)</th>
<th>CG1</th>
<th>CG2</th>
<th>CG3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Program 1</td>
<td>100</td>
<td>20</td>
<td>30</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>Test Program 2</td>
<td>200</td>
<td>40</td>
<td>60</td>
<td>20</td>
<td>40</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 2: Comparison of time cost between our system and GCC

<table>
<thead>
<tr>
<th>Tool</th>
<th>LOC</th>
<th>Time (s)</th>
<th>CG1</th>
<th>CG2</th>
<th>CG3</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>100</td>
<td>100</td>
<td>20</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>Our Tool</td>
<td>200</td>
<td>120</td>
<td>30</td>
<td>60</td>
<td>90</td>
</tr>
</tbody>
</table>

Table 3: Bug fixing results

<table>
<thead>
<tr>
<th>Test Programs</th>
<th>LOC</th>
<th>Patches</th>
<th>Time (s)</th>
<th>Detection</th>
<th>Fixing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Program 1</td>
<td>100</td>
<td>20</td>
<td>30</td>
<td>80%</td>
<td>90%</td>
</tr>
<tr>
<td>Test Program 2</td>
<td>200</td>
<td>40</td>
<td>60</td>
<td>70%</td>
<td>80%</td>
</tr>
</tbody>
</table>

Table 4: Program behavior preserving