International Conference on Computer Safety, Reliability & Security, SAFECOMP'15, Delft, the Netherlands

Automated Generation of Buffer Overflow Quick Fixes using Symbolic Execution and SMT

Technische Universität München, Department of Informatics,
Paul Muntean, Vasantha Kommanapalli, Andreas Ibing, and Claudia Eckert

Introduction

1. According to the CWE/SANS top 25 of most dangerous software errors [1], buffer overflow errors are ranked on 3rd place
2. Problem Statement:
   Provide “in-place” and not “in-place” code patches which can be used independently to remove a buffer overflow bug by using an available bug detector (checker)
3. We need an approach through which one can fix the bugs automatically by generating code patches
4. Definition of Ideal Code Patch [2]:
   An ideal fix covers all bug-triggering inputs and introduces no new bugs
5. Basic program repair consists of 4 steps:
   - Failure Detection: Is there a bug?
   - Bug Diagnosis: What is the cause for the bug?
   - Bug Cause Localization: Where is the bug located?
   - Repair Inference: How to fix the bug?
6. Our Approach:
   Parameterised SMT C code patches. This approach is generalizable and can be applied to other bug checkers that we have developed

Motivating Example

- Listing 1 contains a buffer overflow bug at line number 13 since the buffer index variable “data” can take numeric values which are out of the buffer range
- The lines beginning with the plus symbol “+” in Listi1, (code lines 6, 7 and 12, 14) depict the two generated fixes
- The code comments on line 7 indicate other possible fixes which will most like change the program behavior
- The two fixes can be used independently to fix the bug and follow the same patch pattern (e.g., **if** (condition) **then**{ } and **else**{ } branches)

```
1.  void  foo _ bad(){
2.  int data = -1;
3.  char input_buf [ CHAR_ARRAY_SIZE] = “ “;
4.  if ( fgets (input_buf , CHAR_ARRAY_SIZE , stdin) ! = NULL) {
5.     data = atoi (input _ buf);
6.    + if ( data > 9 || data < 0 )                          -> not in-place fix
7.    + exit ( EXIT_FAILURE);  // data = 9;  or data = rand ()  % 9;  or return 0;
8.    } else {
9.       printLine (“fgets() failed “) ; }
10.  int i, buffer [10] = { 0 };
11.  if ( data >= 0 ){
12.    +if (data <= 9 && data >=0 ){                         -> in-place fix
13.          buffer [ data ] = 1; // buffer overflow bug, index out of range   -> bug location
14.    +} else {exit (EXIT_FAILURE) ; }  // stop program execution
15.       for ( i= 0 ; i < 10; i++) { printIntLine (buffer [i]) ; }
16.    } else {
17.       printLine (“ ERROR : Array index is negative.” ) ; }
18.  }
```

Listing 1. Motivating Example

Steps used in Our Approach

1. **Input Saturation:** The input saturation principle consists of basically limiting the possible values which the buggy variable can take
2. **SMT Constraint System Used for Bug Detection:**

```
(set – logic AUFNIRA )
(declare – fun b () Int )
(declare – fun c () Int )
% c is the buffer size
(assert ( = c 10 ) )
(assert ( > = b c ) )
(check – sat )
( exit )
```

Listing 3. First oracle (excerpt) used to detect the bug

```
( set – logic AUFNIRA )
( set – option : produce-models true )
( declare – fun saturation () Int )
( declare – fun b () Int )
% c is the buffer size
( declare -fun c () Int )
( assert ( = c 10 ) )
( assert ( > = b c ) )
( assert ( < saturation c ) )
( assert ( >= saturation ( c - 1) ) )
( check – sat )
(get – value ( saturation ) )
( exit )
```

Listing 4. Second oracle (excerpt) used to compute the numeric values needed in the final code patch

3. **Bug Type Classification:** It is based on the unique identifier reported by the bug checker
4. **Patch Pattern Selection:** Based on the bug type classification the patch pattern(s) are selected
5. **Constraint Values Selection:** The symbolic variable c (% c is the buffer size) will be selected to constrain the possible values of the buffer index variable
6. **Generating SMT Constraint Values:** The generation of the constraint values is based on the previously stored SMT-Lib system depicted in Listing 4 and new SMT-Lib constraints
7. **Generating Final Code Patches:** After solving the constraint system obtained at step 6, the obtained value(s) will be inserted in the previously selected patch pattern, step 4

Results II

- Figure 1 presents the results of running our tool on 19 “memcpy” programs contained in the open source Juliet test suite [3], CWE-121 test case
- Figure 2 depicts the run-times of our tool on 39 “fgets” programs contained in the open source Juliet test suite [3], CWE-121 test case
- In figures 1 and 2, we can observe that the patch generation time is considerably lower than the bug detection time
- The overall bug detection time is indicated with yellow bars for the “fgets” (1) and “memcpy” (2) programs in Figure 3
- The black bars on top of the yellow bars depicted in Figure 3 represent the total overhead introduced by the patch generation algorithm for the “fgets” (1) and “memcpy” (2) programs
- Note that the highest bar was obtained for Control Flow Variant (CFV) 12 depicted in Figure 2 This bar is higher than the other bars because CFV 12 has far more control flow conditions than the other analysed programs

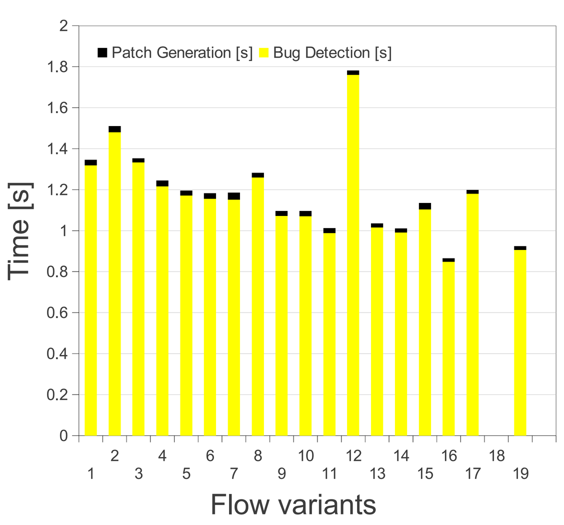

Figure 1. Quick fix generation for CWE-121, “memcpy” programs

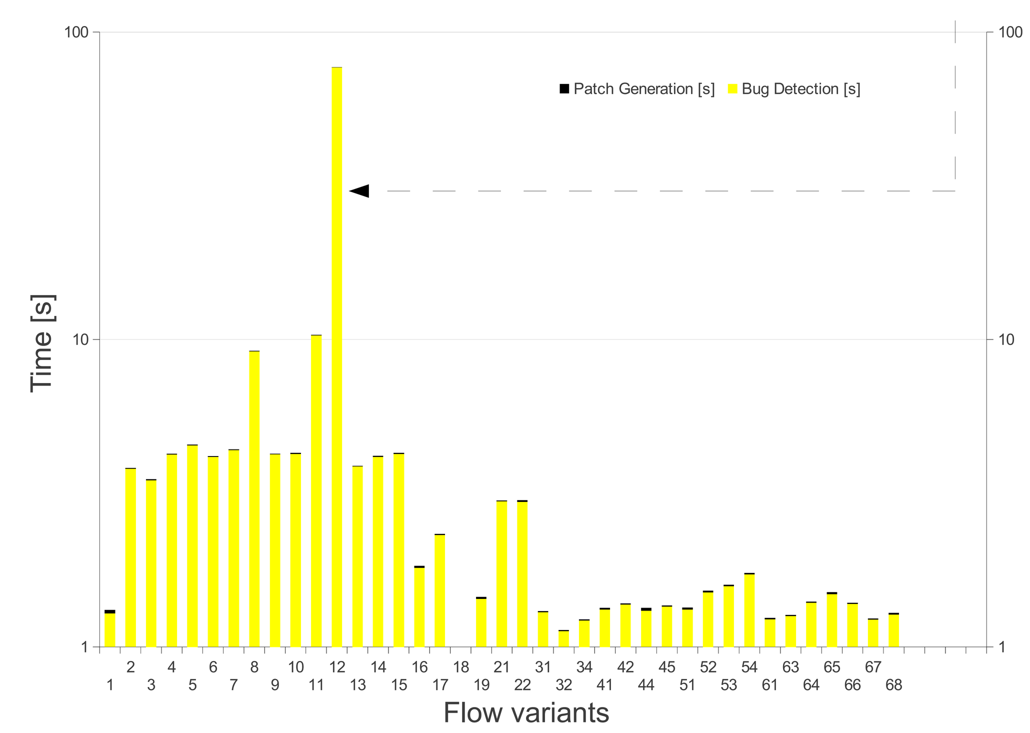

Figure 2. Quick fix generation for CWE-121, “fgets” programs (note: logarithmic scale)

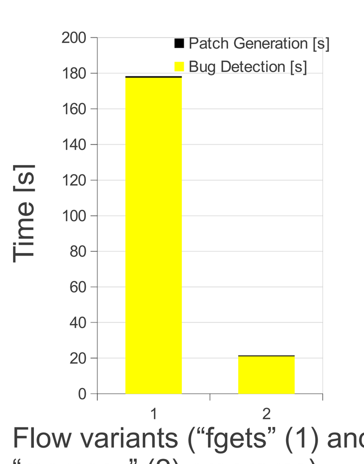

Figure 3. Total patches generation overhead

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
- Satisfiable program execution paths

Output
- Output is a list of refactorings which are used to fix the bugs

Loop 1
- Loop 1 is used for generation of the in-place quick fixes

Loop 2
- Loop 2 is used for generation of the not in-place quick fixes

Input : Satisfiable program execution paths set $S_{paths} := \{s_k | 0 \leqslant k \leqslant n, \forall n \geqslant 0\}$
Output: Refactoring set $R_{set} := \{r_j | 0 \leqslant j < 2\}$

```
W_set := {w_k | 0 ≤ k ≤ n, ∀ n ≥ 0};          // set of working lists, k'th list
N_set := {n_t | 0 ≤ t ≤ n, ∀ n ≥ 0};          // set of nodes
N_set := Ø; W_set := Ø;                       // initializing both nodes set and working list set to empty set
countBP := 0; countGQF := 0                   // init. Counters, count buggy paths and generated fixes
R_set := Ø;
while(Sat_paths.hasNext()) do
    if(hasBug(S_k)) then
        countBP := countBP + 1;               // count the buggy paths
        i := startIndex(s_k);                 // set the start index of the patch
        w_k := setWorkList(s_k);              // set the detected buggy path into the work list
        NLocs := 1;                           // number of quick fix locations
        C := 0;                               // quick fix locations counter
        // if the work list length greater than o else skip path
        if(getLength(w_k) > 0) then
            n_t := initNode(w_k);             // the node at which the bug was detected
            N_set := N_set ∪ {n_t};           // add a node for the in-place fix
            r_j := refact(n_t);               // create a new bug refactoring
            R_set := R_set ∪ {r_j};           // add new refactoring to the set R
            while (i > 0 ∧ C < NLocs) do
                fNode := {w_k,i}              // get next node from work list located at index i
                if(isQuickFixNode(fNode)) then
                    n_t+1 := fNode;           // store current node
                    N_set := N_set ∪ {n_t+1}  // add the node for a not in-place fix
                    setConsObject(w_k);       // store constraint
                    if(notAffectedPaths(S_paths, n_t+1)) then
                        pLoc := probLoc(n_t+1);
                        putMarker(pLoc);              // put new marker
                        r_j+1 := refact(n_t+1);       // create a new bug refactoring
                        R_set := R_set ∪ {r_j+1};     // add refactoring
                        countGQF := countGQF + 1;     // count the generated fixes
                    end
                    C := C + 1;               // increase not in-place quick fix locations counter
                end
                i := i - 1;                   // go one step backwards on the path
            end
        end
        k := k + 1;                           // get next satisfiable program execution path
    end
end
```

Listing 2. Quick fix generation algorithm

Experiments

- **Research Questions**
  RQ1: What is the overall computational overhead of our tool?
  RQ2: Are the generated patches useful for bug fixing?
  RQ3: Is the behavior of the patched program preserved?
- **Test Programs**
  We evaluated our approach on 58 C open source programs contained in the Juliet test suite CWE-121 [3]
- **Methodology**
  We ran our refactoring generation tool on each of the programs and generated two types of patches used for fully automatically fixing the detected bugs
- **Setup**
  For testing purpose we used a system having an 64-bit Linux kernel 3.13.0-32.57, Intel i5-3230 CPU @ 2.60GHz × 4

Results I

- Table 1 depicts the overall computational overhead introduced by the patch generation tool w.r.t. the bug detection time

Test Programs	# LOC	# Paths	# Affected Paths	# Nodes	# Not “in-place” Locations	Patches Generation [s]	Prevented
CWE-121 memcpy	1980	39	0	2918	18	0.424	✓
CWE-121 fgets	8771	641	20	231337	38	0.755	✓
Total	10751	680	20	234255	56	1.197	✓

Table 1. Bug detection and patches generation results

- Table 2 shows that there is no compilation difference between the patched programs and the unpatched programs. This is because the generated fixes have a small size, Lines of Code (LOC), and introduce no compilation overhead

Test Programs	Bug Detection + Patch Generation [s]	GCC Recompile Time [s]	Total [s]	GCC Compilation [s]	Ratio
CWE-121 memcpy	21.454	2.813	24.267	2.813	8.6x
CWE-121 fgets	178.276	6.713	184.989	6.713	27.5x
Total	199.730	9.526	209.256	9.526	36.1x

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

- (✓*) depicted in column 4, of Table 3, indicates that in total for eight C programs the not in-place fix was not applicable since it would have changed the program behavior

Test Programs	Recompile	“in-place” Fix	Not “in-place” Fix
CWE-121 memcpy	✓	✓	✓
CWE-121 fgets	✓	✓	✓*

Table 3. Bug fixing results

- Table 4 shows if the program behavior was preserved after the patch insertion for the two types of generated fixes

Test Programs	# Programs	# IPrograms	# IPaths	% Ratio
CWE-121 memcpy	18	0	0	0
CWE-121 fgets	38	8	20	14.2
Total	56	8	20	14.2

Table 4. Program behavior preserving

Conclusion and Future Work

1. Generated patches are compilable, do not need any human refinement and can be semi-automatically inserted into buggy programs with the help of our re-factoring wizard
2. We think that our approach can be applied to high quality projects since the generated patches remove the bug and preserve the program behavior
3. The generated patches remove the bug and do not change the program behavior for program input which does not trigger the bug
4. Our experimental results show that our tool is efficient and successfully removed all bugs
5. In future we want to use our approach in order to fix other type of bugs and on larger C programs w.r.t. LOC

References

[1] Mitre. 2011 CWE/SANS Top 25, http://cwe.mitre.org/top25/
[2] Z. Gu et al. Has the bug really been fixed?. Proceedings of the ICSE'10, 2010
[3] United States, National Institute of Standards and Technology (NIST): Juliet Test Suite v1.2 for C/C++, online: http://samate.nist.gov/SRD/testsuites/juliet/Juliet_Test_Suite _v1.2_for_C_Cpp.zip

GEFÖRDERT VOM
Bundesministerium für Bildung und Forschung
BMBF-FKZ: 01IS13020