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

Paul Muntean(✉), Vasantha Kommanapalli, Andreas Ibing, and Claudia Eckert

Department of Informatics, Technical University Munich, Munich, Germany
{paul,kommana,ibing,eckert}@sec.in.tum.de

Abstract. In many C programs, debugging requires significant effort and can consume a lot of time. Even if the bug's cause is known, detecting a bug in such programs and generating a bug fix patch manually is a tedious task. In this paper, we present a novel approach used to generate bug fixes for buffer overflow automatically using static execution, code patch patterns, quick fix locations, user input saturation and Satisfiability Modulo Theories (SMT). The generated patches are syntactically correct, can be semi-automatically inserted into code and do not need additional human refinement. We evaluated our approach on 58 C open source programs contained in the Juliet test suite and measured an overhead of 0.59 % with respect to the bug detection time. We think that our approach is generalizable and can be applied with other bug checkers that we developed.

Keywords: Program repair · Symbolic execution · Software bugs

1 Introduction

> "If one tries to put or to retrieve data from a non existing place/index he is going to make a mess" *Mother Nature Law.*

According to the 2011 CWE/SANS top 25 of most dangerous software errors [23] which can lead to serious vulnerabilities in software, buffer overflows are ranked on 3rd place after SQL injection and OS command injection. Buffer overflows can generate risky resource management vulnerabilities as the recent Heartbleed bug [24] confirms. This bug generates a buffer over-read in the OpenSSL library by leaking sensitive information to the outside world without the need for the attacker to have root access on the attacked system and without leaving any trace on the attacked system. This proves that buffer overflows can lie undiscovered in software for many years and can lead to extremely dangerous information leaks in highly used open source software.

In this paper we focus on fault localization and repairing of buffer overflow bugs by leveraging precise information (failure detection, bug diagnosis, buggy variables (program variables which are directly responsible for bug appearance),

F. Koornneef and C. van Gulijk (Eds.): SAFECOMP 2015, LNCS 9337, pp. 441–456, 2015.
DOI: 10.1007/978-3-319-24255-2_32

e.g., buffer index or buffer size) provided by our buffer overflow checker [13]. The failure detection and bug diagnosis data is used to generate quick fixes for buffer overflows and to support the repair process of removing the bug with a refactoring wizard. A novel algorithm is used to detect possible insertion locations in code for the generated code patches ((a) "in-place"—directly before the statement which contains the bug and (b) by searching for other, not "in-place" locations where the bug can be fixed). Our approach for generating program repairs is based on: code patch patterns, SMT solving and possible quick fix locations searching in program execution paths which could affect the program behavior by inserting a patch at a not "in-place" location. The generated patches are sound (e.g., do not change the behavior of the program for input which does not trigger the bug), final (no further human refinement needed), human readable (no alien code), syntactically correct and compilable.

We address offline behavioral repair [25] (by modifying the source code). Others have addressed state [8] or test-suite based program repair such as GenProg [19] and PAR [18]. The defect class which we address is inappropriate index variable assignment which results in an incorrect usage of the buffer index range. The fix defect class consists of input checks based on semi-defined patch patterns. The aim of the quick-fix is fail-secure error mitigation (e.g., to prevent that an attacker exploits the error in order to gain system access). The final version of the patch is determined using SMT solving.

Program repair lies at the conjunction of two dimensions (first, an *oracle* is needed to decide what is incorrect in order to detect the bug (first dimension) and another *oracle* to tell what should be kept correct for sake of non-regression (second dimension)) of software correctness [25]. We used the same SMT constraint system which was used to trigger the bug for defining what is incorrect in the program. Additionally, we created a second SMTLib (constraint system definition language used by the Z3 [5] solver) constraint system consisting of the previously mentioned constraint system and new SMTLib constraints used to impose input saturation constraints on the buggy variable.

Our patches are generated automatically and inserted semi-automatically offline with the possibility to insert them also online.

Our problem statement: Provide code patches ("in-place" or not "in-place") which can be used independently to remove a buffer overflow bug using a bug detector (checker).

In summary we make the following contributions:

- An algorithm for generation of "in-place" and not "in-place" bug fixes, Sect. 4.1.
- A novel approach for bug fix generation based on input saturation, Sect. 4.2.
- Semi-automated patch insertion based on source files differential views, Sect. 4.3.
- Automated check for behavior preserving of the patched program, Sect. 6.4.

Further we present related work in Sect. 2 and a motivating example of a buffer overflow bug and why automated bug repair merits future research in Sect. 3. We present the algorithm used to search for quick fix locations and generation

of quick fixes in "in-place" (at the location where the bug was detected) and not "in-place" in Sect. 4. We discuss implementation details of our tool in Sect. 5 and present experimental results and the evaluation in Sect. 6. Finally, we conclude in Sect. 7.

2 Related Work

Source code patches for quick fixing bugs can be generated in different repairing ways [12], from free form bug reports [1,2,33], from statically defined patch patterns [11,18], from test suite using SMT solver [7,28], from test suite and genetic programming [19,34], by replacing the unsafe *libc* [29,32], functions with safe functions [3]. Hafiz et al. [30] addressed **buffer overflows quick fixing** by replacing unsafe library functions with safe alternatives. Cowan et al. [4] have used **static analysis for generating code patches** based on four approaches in which the buffer overflow vulnerabilities can be *defended*. Jacobs [16] has proposed to use **buffer overflow refactoring patterns** as an extension for the C language called SMART C. In recent years, many **quick fix generation tools for buffer overflows** have been proposed: AutoPaG [21], SafeStack [17], DYBOC [31], TIED [6], LibsafePlus [6], LibsafeXP [20], HeapShield [9].

To the best of our knowledge the AutoPAG [21] tool developed by Lin an colleagues is most similar to our approach from the backward visiting of program statements perspective. Our tool can not be compared with AutoPAG from the point of view of computation time and quick fix quality at this stage of development since AutoPAG has several limitations which we will briefly list. Our algorithm stops the search after encountering the first not "in-place" bug fix location whereas AutoPAG tries to detect all possible not "in-place" bug fix location by running a repeated inefficient data flow analysis (no program execution paths used). AutoPAG is not aware of program execution paths and uses a rudimentary backward information flow propagation approach based on the sequential ordering of program statements. The analysis (no SMT solver used) is repeated until there are no visited variables in the previously constructed set of tainted variables. This set can contain all program variables and can generate a significant overhead as already mentioned in the AutoPAG paper.

3 Motivating Example

In this section we present two real-world bug fixes as an example to highlight the fact that bug patch generation is not a trivial task. It needs deep insights into the functionality of the program and merits further study. There are typically an endless number of programs who adhere to a formal specification. As such, a bug can be fixed with infinite number of functionally correct patches. The automatically generated patches will change the behavior of the program or not. We present two distinctive patches depicted in Listing 1 on lines 5–6 and 11–13 with "+" and by using an italic font. Note, that these two fixes do not change program behavior for program input which does not trigger the bug. Listing 1

contains on line 6 code comments we present other possible quick fixes usable to remove the buffer overflow bug located at line 12 which most likely will change program behavior.

Listing 1 displays a C code snippet extracted from the test case CWE-121 [22] which is contained in [27]. The code snippet contains a buffer overflow bug at line 12 which can be removed by using one of the two patches depicted in Listing 1 on lines 5–6 and 11–13. Note, that the patch structure, the used constraint variables and the bug fix insertion locations are different for each buggy C program.

Listing 1. Buffer overflow bug due to missing input checks

```
0. void foo_bad(){
1.  int data = -1;
2.  char input_buf[CHAR_ARRAY_SIZE] = "";
3.  if (fgets(input_buf,CHAR_ARRAY_SIZE,stdin) != NULL){
4.    data = atoi(input_buf);
5.  + if (data > 9 || data < 0)
6.  + exit(EXIT_FAILURE); // data = 9; or data = rand() % 9; or return 0;
7.  }else{
8.    printLine("fgets() failed.");}
9.  int i, buffer[10] = { 0 };
10. if (data >= 0){
11. + if (data <= 9 &&data >= 0){
12.       buffer[data] = 1;  // Buffer overflow bug, index out of range
13. +}else{exit(EXIT_FAILURE);} // stop program execution
14.   for(i = 0; i < 10; i++){printIntLine(buffer[i]);}
15. }else{
16.   printLine("ERROR: Array index is negative.");}
17. }
```

Finding the *right* program variables in order to impose a constraint through a patch is a hard task because, in the worse case the values selection depends on all the other program variables. Determining not "in-place" bug fix locations is not a trivial task and this should be based on correct bug detection and on a kind of backward program execution technique on all program execution paths which contain the bug. In general, the insertion location and structural form of the quick fix patch can influence the overall program behavior. Thus, care should be taken that a patch is syntactically correct, compilable and does not change program behavior for program input which does not trigger the bug.

4 Quick Fixes Generation

In this section we present our quick fix locations search algorithm, the steps needed to automatically generate buffer overflow fixes and the mechanism for inserting the patches semi-automatically into the buggy program.

4.1 Quick Fix Locations Search Algorithm

The Algorithm 1 contains two phases as follows: (a) Finding the first program execution path which contains the buggy statement and generating the "in-place" quick fix. This quick fix will be suggested to the user in the GUI only if it is sound (e.g., the buffer size is equal on all buggy program executions paths).

Note, that the buffer index and size are context-sensitive. In case there are different buffer sizes on different paths then in order to preserve the soundness of the patched program a complex "in-place" patch can be generated containing one "if" branch and N "if else" branches (N represents the number of different buffer sizes on each buggy path). Furthermore, the size (LOC) of this patch grows exponentially with the number of paths containing different buffer sizes which renders such a quick fix to be not always practical (*). (b) Traversing the current selected path in backward program execution order from the location where the bug was found until a not "in-place" fix location is detected and generating a new quick fix at that location. At this program location the program execution can be safely finished (e.g., exit(EXIT_FAILURE);) (this will not change the program behavior for input which does not trigger the bug) if the buffer index is out of bounds or a numeric value can be set if desired (this will not terminate program execution and most likely will change program behavior). The second quick fix is sound as it can be observed that it does not change program behavior for program input which does not trigger the bug—similar to the first "in-place" quick fix. Quick fix (b) represents an alternative for the first quick fix which is not always feasible (e.g., (*)) and will be suggested in the GUI only if it does not change program behavior for input which does not trigger the bug and for each buggy program execution path at least one not "in-place" quick fix was successfully generated. This is assessed with the counters *countBP* and *countGQF* indicated in Algorithm 1 which must be equal (each buggy path has a not "in-place" quick fix associated) when the algorithm finishes the search. If the counters are not equal when the search algorithm finishes then there is at least one path where a not "in-place" bug fix location was not found. Thus, the whole quick fix will be not offered in the GUI since there could exist one program execution path on which the bug was not fixed.

Phases (a) and (b) are repeated for all program execution paths which contain the buggy statement (line number and file name) where the bug was detected as follows: first, the algorithm searches for possible insertion locations (e.g., "in-place" and not "in-place") for buffer overflow quick fixes and second, it generates bug fixes. The algorithm uses: *startIndex()* to set the start index from where to search on the initial path, *setWorkList()*, to initialize the buggy first path, *initNode()*, to initialize the node at which the bug was found and *refact()*, to create a new refactoring. We now extend the notation, S_{paths}, consisting of all program execution paths, W_{set}, used to hold the current selected execution path, N_{set}, is a set of nodes used to store "in-place" and not "in-place" path nodes (these represent program locations where refactorings will be later on inserted) and R_{set}, is the set of refactorings. In line 3, N_{set}, and, W_{set} are initialized to empty set. In line 6, the algorithm picks a new path from the, S_{paths}, in each new iteration. Upon verifying that the chosen path contains the buffer overflow bug (previously detected), *hasBug(*s_k*)*, the start index, i, and the initial buggy path, w_k, are initialized. In line 11 and 12, the number of quick fix locations, *NLocs*, is initialised to 1 and the quick fix location counter, C, to 0 respectively. On encountering the condition statement, getLength(w_k) $>$ 0, the algorithm

Algorithm 1. Quick fix locations searching and patches generation

```
   Input: Satisfiable program execution paths set S_Paths := {s_k| 0 ≤ k ≤n, ∀ n ≥ 0}
   Output: Refactorings set R_set := {r_j| 0 ≤ j < 2}
 1 W_set := {w_k| 0 ≤ k ≤ n, ∀ n ≥ 0}; // set of working lists, k'th list
 2 N_set := {n_t| 0 ≤ t ≤ n, ∀ n ≥ 0}; // set of nodes
 3 N_set := ∅; W_set := ∅; // initializing both nodes set and working list set to empty set
 4 countBP :=0; countGQF :=0; // init. counters, count buggy paths and generated fixes
 5 R_set :=∅;
 6 while ((Sat_paths.hasNext)) do
 7     if (hasBug(s_k)) then
 8         countBP := countBP + 1; // count the buggy paths
 9         i := startIndex(s_k); // set the start index of the path
10         w_k := setWorkList(s_k); // set the detected buggy path into the work list
11         NLocs := 1; // number of quick fix locations
12         C := 0; // quick fix locations counter
13          // if the work list length greater than 0 else skip path
14         if (getLength(w_k) > 0) then
15             n_t := initNode(w_k); // the node at which the bug was detected
16             N_set :=N_set ∪ {n_t}; // add a node for the in-place fix
17             r_j := refact(n_t); // create a new bug refactoring
18             R_set :=R_set ∪ {r_j}; // add new refactoring to the set R
19             while (i >0 ∧ C <NLocs) do
20                 fNode := {w_k,i}; // get next node from work list located at index i
21                 if (isQuickFixNode(fNode)) then
22                     n_t+1 := fNode; // store current node
23                     N_set:=N_set ∪ {n_t+1}; // add the node for a not in-place fix
24                     setConsObject(w_k); // store constraint
25                     if (notAffectedPaths(S_Paths, n_t+1)) then
26                         pLoc := probLoc(n_t+1);
27                         putMarker(pLoc); // put new marker
28                         r_j+1 := refact(n_t+1); // create a new bug refactoring
29                         R_set :=R_set ∪ {r_j+1}; // add refactoring
30                         countGQF := countGQF + 1; // count the generated fixes
31                     end
32                     C := C + 1; // increase not in-place quick fix locations counter
33                 end
34                 i := i - 1; // go one step backwards on the path
35             end
36         end
37         k := k + 1; // get next satisfiable program execution path
38     end
39 end
```

checks the working list, w_k, length if it is greater than 0 and then initializes the node where the bug was found updating the nodes set, N_{set}. In line 17 a new refactoring is created updating the refactorings list in line 18. From line 19 to line 35, the algorithm traverses the path backwards in order to find a not "in-place" fix location of the bug until the index value, i, is greater than 0 and the counter value, C, is less than number of quick fix locations, $NLocs = 1$. While visiting each path node it checks for potential not "in-place" locations, *isQuickFixNode(fNode)*. Upon encountering a not "in-place" location, it stores the current node, n_{t+1}, and then N_{set} is updated. This node is used for generating the bug patches. In line 24 the constraint object is set at the index, k. The algorithm traverses the current selected program execution path to check if there are any influenced paths using, *notAffectedPaths*(S_{Paths}, n_{t+1}). At this stage of development a simple check is performed in order to see if the context-sensitive buggy variable appears on the right hand side of an expression (e.g., var = expr.; e.g. expr. = a binary expression, expr. contains our buggy variable which will

be constrained with the patch). Furthermore, it is needed to be checked if the other influenced variables (e.g., var) are dead or live variables along a program path. In future we plan to compute a *distance-bounded weakest precondition* [10] (our engine supports the weakest precondition computation; loop and recursion invariants are not supported) in order to check if program behavior is preserved or not.

In case the algorithm finds no influenced paths then a new refactoring is created and added to R_{set}, line 29. Note, that in case of using "exit(EXIT_FAILURE)" in the not "in-place" quick fix than no check for influenced paths is needed. In line 32 the counter, C, is incremented by 1 which indicates that a second refactoring was created and the index value, i, is decremented by 1 so that the algorithm proceeds one step backwards on the current path in line 34. Note, that the algorithm can accommodate the search for more than one not "in-place" location by increasing the value of *NLocs* and updating the detection rules.

4.2 Bug Detection with SMT

Our contribution lies in bridging the gap between a buffer overflow bug report provided by an existing buffer overflow checker and automated generation of one or more quick fixes (quick fix structure, insertion location and values used inside the patches) which remove (automatically assessed by re-running the bug detector on the patched program) the buffer overflow bug.

The bug localization is based on the buffer overflow checker contained in our static analysis engine [13]. The buffer overflow checker returns the location of the bug containing the file name, line number and a unique ID which defines the type of the bug. Based on the bug report ID the following steps are performed automatically. The SMTLib constraint system which was used to detect the bug (from the buffer overflow checker) is selected. After obtaining the system a `SMTConstraintObject` object is instantiated containing the following attributes: the buffer size, the offset and the previous mentioned SMTLib constraint system. Next, we introduce the patch creation process consisting of the following 7 steps.

Step 1. Input Saturation: Listing 1 contains at line 4 a not "in-place" quick fix location for the buffer overflow bug which can be addressed with a missing input check, lines 5–6. Due to the missing input check the values of the index variable `data` used in `buffer[data]` can take values outside the buffer index interval [0, 9] which leads to a buffer overflow or underflow. In order to determine if the index variable `data` can take values outside the allowed interval [0, 9] a SMT constraint system is generated. The SMT constraint system is provided as input to the Z3 [5] SMT solver which will output the message `SAT` if `data` can take values outside the allowed interval. In order to remove the buffer overflow bug we decided to generate two types of quick fixes ("in-place" and not "in-place") which are based on the input saturation principle. The input saturation principle consists of basically limiting the possible values which the index variable `data` can take to only values which are contained in the buffer index range. The generated quick fixes represent additional checks which limit the upper and lower values of `data`

(see Listing 1). The upper allowed value for `data` should not be larger than the allowed `buffer[data]` upper index bound value, 9, and not smaller than 0.

Step 2. SMT Constraint System used for Bug Detection: The original SMT constraint system used to detect the buffer overflow bug (excerpt presented in Listing 2) had 317 LOC. We depict only the SMTLib statements which matter most in our context and changed the names of the symbolic variables for brevity. During buffer overflow/underflow detection the checker uses SMTLib statements which represent path constraints and other specific statements for buffer overflow or underflow checking. The statement in bold font located on line 5 in Listing 2 represents the constraint which we get from the our checker `(assert (>= data bufferSize))` in case of checking for an buffer overflow. In the case of an buffer underflow check the checker adds to the constraint system the statement `(assert (< data 0))`. If one of these two constraints are satisfied then this means that the variable `data` can take values outside the range of the buffer. Thus, a buffer overflow or underflow bug report will be issued. The value of `data` depicted in Listing 2 with `b` is constraint to be greater or equal 10. The solver answers to this constraint system from Listing 2 as `SAT`. Thus, the set of possible solutions for `b` is contained in the set $[10, +\infty)$. This means that if the program variable `data` takes any value greater or equal to 10 than a buffer overflow bug will be detected. The checker is checking each possible execution path by asking the Z3 solver if the SMT constraint system is satisfiable or not. If a bug report is issued then a `SMTConstraintObject` will be instantiated. The buffer size, buggy variable and the SMT constraint system used to trigger the bug are added as attributes to the previously generated `SMTConstraintObject` object.

Listing 2. First oracle

```
0. (set-logic AUFNIRA)
1. (declare-fun b () Int)
2. (declare-fun c () Int)
3. % c is the buffer size
4. (assert (= c 10 ))
5. (assert (>= b c))
6. (check-sat)
7. (exit)
```

Listing 3. Second oracle

```
0. (set-logic AUFNIRA)
1. (set-option:produce-models true)
2. (declare-fun saturation () Int)
3. (declare-fun b () Int)
4. % c is the buffer size
5. (declare-fun c () Int)
6. (assert (= c 10 ))
7. (assert (>= b c))
8. (assert (< saturation c))
9. (assert (>=(saturation (c-1)))
10. (check-sat)
11. (get-value (saturation))
12. (exit)
```

Step 3. Bug Type Classification: The bug type classification is based on the checker which was used to detect the bug. The bug checker generates a report containing a unique identifier for each type of bug detected. Currently we have other checkers (information flow checker [26], infinite loop checker [15], integer overflow checker and race condition checker [14]) which can run in parallel and have unique checker identifiers. We used for our checkers a unique ID which was saved in the checker bug report. Based on the generated bug identifier we can decide which type of bug we are dealing with. After obtaining the unique bug identifier the bug patch pattern is selected which will be used for bug fixing.

Step 4. Patch Pattern Selection: Based on the bug type classification we select the patch pattern(s) which can be used to fix this type of bug. Our patch patterns consist of empty C code skeletons where certain values have to be

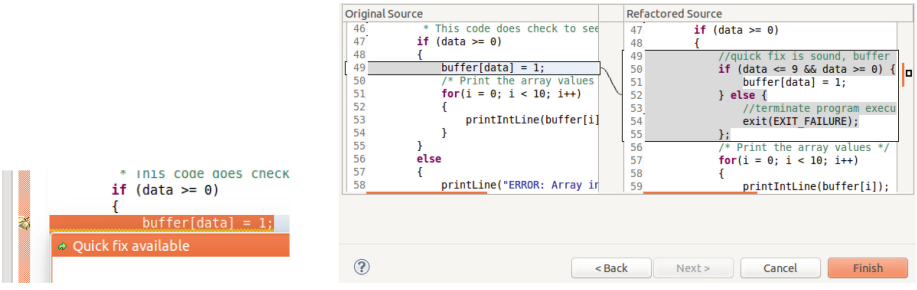
computed based on the used SMT solver (e.g., (1) +if (buff_size > N || buff_size < 0); e.g., (2) +if (buff_size <= N && buff_size >= 0);). N represents the buffer size determined during static analysis. Note, that N can have different values on different program execution paths.

Step 5. Constraint Values Selection: We construct our SMT constraint system based on the attributes stored in the `SMTConstrintObject` object. These attributes have to be added to the SMT constraint system which was used to detect the bug. After solving the SMT constraint system we will obtain the numeric values which will be inserted into the previous selected patch patterns.

Step 6. Generating SMT Constraint Values: The generation of the constraint values is based on the previously stored SMTLib system as an attribute of the `SMTConstraintObject`. The new SMT constraint system (see Listing 3) contains the same SMT statements presented in Listing 2 plus some new SMTLib statements used to perform the calculation of the needed value which will be later on used inside our selected patch(es). Note, that the newly added SMTLib statements are marked with bold font in Listing 3. The added SMTLib statements are used to perform input saturation on the variable `data`. The solver answers to this constraint system from Listing 3 as `SAT`. After solving the generated SMT system we obtain the value 9 for `data`. This value will be used later when we generate our final code patches. `b` represents the symbolic variable `data` and the symbolic variable `saturation` represents our constraint variable used to constrain the variable `data`. The symbolic variable `saturation` is used to constrain the solution space of the real variable (source code variable) `data`. The symbolic variable `saturation` can have as solution only the numeric value 9.

Step 7. Generating Final Code Patches: After solving the constraint system from Listing 3 we obtain the numeric value `9` as solution for the symbolic variable `saturation`. The value 9 will be inserted in the previously selected patch patterns in order to constrain the possible values which `data` can take. After this step, we obtain code patches which are syntactically correct, can be compiled and could be further on edited after insertion if desired.

4.3 Semi-automatic Patch Insertion Wizard



(a) (b)

Fig. 1. Patch insertion wizard (Color figure online)

The buffer overflow checker places a bug marker depicted in the Fig. 1(a) with a yellow bug icon, on the left of the C statement if the statement contains a buffer overflow bug. By pressing on this bug marker the user can start the code refactoring wizard. The code refactoring wizard is composed of two user pages. The first user page is used to make patches selections (in-place or not in-place fix, only one can be selected at a time). The second page depicted in the Fig. 1(b) contains a differential files view presenting the differences between the original file containing the bug and the modified file with the selected patch(es) inserted. The user has the possibility to navigate between this two pages using the buttons "<Back", "Next>" and "Cancel" in order to compare the results of applying the in-place or the not in-place quick fix. Finally, by pressing on the "Finish" button the user accepts the selected quick fix, the patch will be written in the file and the wizard will be stopped.

5 Implementation

We have integrated our bug fixing tool into our existing Static Analysis Engine (SAE) [13] which is developed as an Eclipse IDE plug-in. We implemented a refactoring wizard based on the Eclipse Language Tool Kit (LTK), JFace and CDT in order to introduce semi-automatically the generated bug patches into the buggy program. Our bug patching technique is composed of two steps. First, the bug detection analysis is performed. If the bug is detected then this will be marked with a marker. Second, the bug fixing algorithm starts to search backwards on the buggy path until it detects a first not "in-place" location. If such a bug fix location is found then our tool marks visually the location in code with another marker. The backward searching algorithm can be easily updated in order to accommodate the suggestion of multiple quick fixing locations which can be addressed with other techniques than input saturation.

6 Evaluation

6.1 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. We used our previously developed buffer overflow checker for bug detection and classification. The time needed to generate the patches and the total time needed to run the bug detection were measured in milliseconds and then converted to seconds [s]. We used as test system an 64-bit Linux kernel 3.13.0-32.57, Intel i5-3230 CPU @ 2.60 GHz × 4. Note, that we replaced in the sound not "in-place" quick fix depicted in Listing 1 the string "exit(EXIT_FAILURE);" with "data = 9;", which is equivalent to "data = (bufferSize − 1);" (bufferSize can have different values for different program paths) in order to see if our apporach for detecting affected paths works. Note, that by using the not "in-place" quick fix depicted in Listing 1 (contains "exit(EXIT_FAILURE);" instead of "data = 9;")

the program behavior is preserved w.r.t. program input which does not trigger the bug. We evaluated our approach by addressing three research questions:

RQ1: What is the overall computational overhead of our tool? We wanted to find out what was the overhead introduced by our patch generation tool with respect to the bug detection time.

RQ2: Are the generated patches useful for bug fixing? We wanted to find out if the final generated patches containing the values obtained from the Z3 solver are useful for the bug fixing.

RQ3: Is the behaviour of the patched program preserved? We wanted to find out if the generated patches change the program behaviour. Finally, we addressed threats to validity of our approach.

6.2 RQ1: Performance of Our Tool

We addressed RQ1 by measuring the performance of our tool in terms of the patch generation overhead compared with the bug detection time.

Figure 2(a) presents the results of running our tool on 19 memcpy programs contained in CWE-121. The introduced overhead is calculated by comparing the times represented with black bars (patch generation time) located on top of the yellow bars (bug detection time). The total overhead of 1.97 % was obtained by comparing the bug detection time, 21.030 [s] ($21.454[s] - 0.424[s]$), and the patch generation time, 0.424 [s], column 7 of Table 1.

Figure 2(b) contains the results obtained during patch generation for the 39 fgets programs contained in CWE-121. We used the same index enumeration which was used in the open source Juliet test suite [27] in order to have a clear mapping between analyzed programs and programs extracted from the test suite. In comparison with the Fig. 2(a) we used in Fig. 2(b) a logarithmic

Table 1. Bug detection and patches generation results

Test programs	#LOCS	# 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 2. Comparison of time cost between our system and GCC

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

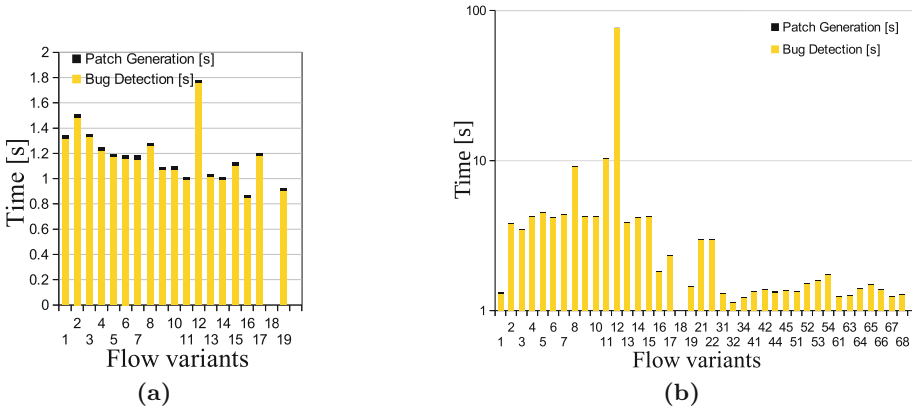


Fig. 2. Quick fix generation for memcpy and fgets programs (Color figure online)

scale in order to make the results better readable. From Fig. 2(b) we observe that the patch generation times indicated with black bars on top of the yellow bars are considerably lower than the bug detection times indicated with yellow bars. The total overhead of 0.4 % was obtained by comparing the bug detection time, 17.7521 [s], (17.8276[s] − 0.755[s]) and the patch generation time (0.755 [s]) contained in column 7 of Table 1.

The obtained results show that the patch generation time grows by a factor less than 2 (from 0.424 [s] to 0.755[s]) if the number of execution paths increases by a factor of 16.4x (641/39, see Table 1 3rd column) and the number of nodes by a factor of 79.2x (231337/2918, see Table 1, 5th column). We demonstrated that our approach is applicable to open source C programs and the induced overhead is under 1 %.

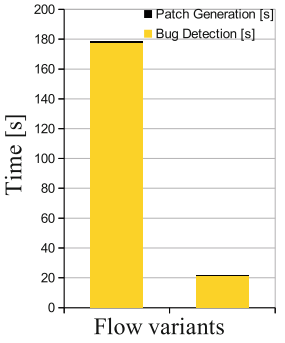


Fig. 3. Total overhead (Color figure online)

Figure 3 presents the overall overhead with yellow bars (the bug detection time) for the fgets and memcpy programs. The black bars on top of the yellow bars represent the overhead introduced by the patch generation algorithm for the fgets and memcpy programs. The patch generation overhead is 1.197 [s] which represents 0.59 % from the bug detection time of 199.730 [s] indicated in column 7 of Table 1 and in column 2 of Table 2.

Table 2 shows that there is no compilation difference between the patched programs and the un-patched programs. This is because our patches have a small size and introduce no compilation overhead. We observed an overhead decrease from 1.97 % (memcpy programs) to 0.4 % (fgets programs) for 79.27 times (231337/2918, see Table 1, 5'th column) more nodes and for 16.4 times (641/39, see Table 1, 3'rd column) more paths. With regard to RQ1, the results

confirm that the patch generation overhead is 0.59 % when compared to the bug detection time.

6.3 RQ2: Usefulness of the Generated Fixes

We addressed RQ2 by considering following scenarios: First, the syntactical correctness of our generated patches and if the code can be recompiled after the patch was inserted. Second, if the bug patch was useful for removing the detected bug. Third, the usefulness of the not "in-place" patch which is depicted in Table 3, column 4. Table 3, column 2 shows if the resulted program after the insertion of the "in-place" or the not "in-place" patches remained compilable. Columns 3 and 4 depicted in Table 3 indicate if the bug was removed by inserting the patch "in-place" (bug location) or at the not "in-place" location.

Table 3, column 3 shows that all the bug could be removed by inserting the patch at the place where the bug was detected. Table 3, column 4 shows that all the bugs were removed by inserting the patch at the not "in-place" location except the ones indicated with ✓*. The notation "N (M)" was used to denote the control flow variant "N" and the number of detected affeted paths, "M" contained in the Juliet test case CWE_121_fgets [27]. In total 8 C programs: 42 (3), 45 (2), 61 (1), 63 (4), 64 (5), 66 (2), 67 (1), 68 (2) contained 20 $((3)+(2)+(1)+(4)+(5)+(2)+(1)+(2))$ affected paths. An affected path contained at least one usage of the constrained variable (e.g., "data") in another statement as the path was traversed in program execution order. Thus, the program behavior can be in this way influenced by the set of values that the constraint variable can take after it was constrained. Note, that this is not a sufficient condition to guarantee soundness. Thus, the results presented in Table 3 confirm RQ2, that the generated bug patches were useful for removing the bugs.

Table 3. Bug fixing results

Test programs	Recompile	"in-place" Fix	Not "in-place" Fix
CWE-121 memcpy	✓	✓	✓
CWE-121 fgets	✓	✓	✓*

6.4 RQ3: Program Behavior Preserved After Patch Insertion

We addressed RQ3 by checking if the inserted patch at the not "in-place" location influences other existing program paths. The abbreviations in Table 4 mean: Total number of programs containing influenciable paths (IPrograms), Influenciable Paths (IPaths), % Ratio represents the ratio between the total number of programs to the total number of programs containing at least one influenced path. Our Algorithm 1 visits not "in-place" candidate nodes in backward program execution manner in order to find bug fixing locations. Next, it checks if by patching the found node contained in the affected path the behavior of the program will change. If the algorithm finds an affected path then the not "in-place" quick fix will be not generated since it could influence other variables contained in the affected paths. We successfully avoided changing the behavior of all the analyzed

Table 4. Programs behavior preserving

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

programs by proposing the fix at the bug location which is indicated in column 3 of Table 3. For 14.2 % of the programs (56/8, **# Programs/# IPrograms** presented in Table 4 in columns 2 and 3) we avoided changing the behaviour by not proposing the not "in-place" quick fix. Thus, we can confirm that for 85.8 % (100 % − 14.2 %) of the analyzed programs the program behaviour did not changed with regard to RQ3.

6.5 Threats to Validity

Internal Validity: In case we did not interpret the results of our execution measurements right then the overhead of 0.59 % could not be achievable. To avoid time measurement mistakes we carefully designed our time measuring mechanism and measured for all the 58 programs three times. Some of the decisions we make are static (select type of patch patterns for a bug type) and some are dynamic (SMT constraint system solving). Thus, only the dynamic decisions can influence the overhead introduced by our tool. We are aware that in case the bug checker cannot detect or diagnose the bug type then our approach would suffer from imprecision or does not work at all.

External Validity: We are aware that there are some threats which could hinder our approach from being generalizable for large programs. We think that our patch generation approach can be generalized since we followed the basic automatic program repair steps (failure detection, bug diagnosis, bug cause localization and repair inference). We think that 0.59 % overall overhead is negligible and by addressing other types of checks than input saturation or by using other bug patterns no major time increase would be noticeable. Thus, programs containing long execution paths would not increase the overhead significantly with respect to the bug detection time.

7 Conclusion and Future Work

We presented a novel approach which can be used to automatically fix buffer overflow bugs by generating bug patches using static execution and SMT solving. Our automatically generated patches do not need any human refinement, are compilable and can be semi-automatically inserted into buggy programs with the help of our refactoring wizard. Our experimental results show that our tool is efficient and successfully removed all bugs. We think that our approach can be applied to high quality projects since the generated quick fixes remove the bug and preserve program behavior.

We are confident to say that our approach can be applied in future in conjunction with other types of bug checkers [14,15,26] which we developed.

Acknowledgments. This research is funded by the German Ministry for Education and Research (BMBF) under grant number 01IS13020.

References

1. Aho, A.V., et al.: A minimum-distance error-correcting parser for context-free languages. SIAM J. Comput. **1**(4), 305–312 (1972)
2. Chen, L., et al.: R2Fix: automatically generating bug fixes from bug reports. In: Proceedings of the 2013 IEEE 6th ICST
3. Crispin, C., et al.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX SSYM 1998
4. Crispin, C., et al.: Buffer overflows: attacks and defenses for the vulnerability of the decade*. In: DARPA Discex 2000
5. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Deepak, G., et al.: TIED, LibsafePlus: tools for runtime buffer overflow protection. In: Proceedings of the 13th Conference on USENIX Security Symposium, SSYM 2004
7. DeMarco, F., et al.: Automatic repair of buggy if conditions and missing preconditions with SMT. In: Proceedings of the CSTVA 2014
8. Demsky, B., Rinard, M.: Automatic detection and repair of errors in data structures. In: Proceedings of the ACM SIGPLAN OOPSLA 2003
9. Emery, D.B.: HeapShield: library-based heap overflow protection for free. UMass CS TR 06-28 (2006)
10. Gu, Z., et al.: Has the bug really been fixed? In: Proceedings of the ICSE 2010
11. Haddad, H.M., Shahriar, H.: Rule-based source level patching of buffer overflow vulnerabilities. In: Proceedings of the 10th ITNG 2013
12. Harrold, M.J., et al.: Fault prediction, localization, and repair. Dagstuhl Seminar 13061, February 2013
13. Ibing, A.: SMT-constrained symbolic execution for eclipse CDT/Codan. In: Proceedings of the 3th WS-FMDS 2013
14. Ibing, A.: Path-sensitive race detection with partial order reduced symbolic execution. In: Canal, C., Idani, A. (eds.) SEFM 2014 Workshops. LNCS, vol. 8938, pp. 311–322. Springer, Heidelberg (2015)
15. Ibing, A., Mai, A.: A fixed-point algorithm for automated static detection of infinite loops. In: Proceedings of the 16th IEEE HASE 2015
16. Jacobs, M., Lewis, E.C.: SMART C: a semantic macro replacement translator for C. In: Proceedings of the Sixth IEEE SCAM 2006
17. Jin, H., et al.: SafeStack: automatically patching stack-based buffer overflow vulnerabilities. IEEE Trans. Dependable Secure Comput. **10**(6), 368–379 (2013)
18. Kim, D., et al.: Automatic patch generation learned from human-written patches. In: Proceedings of the International Conference on Software Engineering, ICSE 2013
19. Le Goues, C., et al.: Genprog: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
20. Lin, Z.: LibsafeXP: a practical and transparent tool for run-time buffer overflow preventions. In: Proceedings of the 7th Annual IEEE Information Assurance Workshop, IAW 2006
21. Lin, Z., et al.: AutoPaG: towards automated software patch generation with source code root cause identification and repair. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, ASIACCS 2007
22. Mitre: CWE-121. http://cwe.mitre.org/data/definitions/121.html

23. Mitre: 2011 CWE/SANS Top 25. http://cwe.mitre.org/top25/
24. Mitre: Heartbleed Bug. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160
25. Monperrus, M.: A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014
26. Muntean, P., et al.: Context-sensitive detection of information exposure bugs with symbolic execution. In: Innovative Software Development Methodologies and Practices, InnoSWDev 2014
27. NIST: Juliet Test Suite v1.2 for C/C++
28. Satish, C., et al.: SemFix: program repair via semantic analysis. In: Proceedings of the International Conference on Software Engineering, ICSE 2013, pp. 772–781
29. Sauciuc, R., Necula, G.: Reverse execution with constraint solving. Technical report No. UCB/EECS-2011-67, May 2011
30. Shaw, A., et al.: Automatically fixing C buffer overflows using program transformations. In: Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks, DSN 2013
31. Sidiroglou, S., Giovanidis, G., Keromytis, A.D.: A dynamic mechanism for recovering from buffer overflow attacks. In: Zhou, J., López, J., Deng, R.H., Bao, F. (eds.) ISC 2005. LNCS, vol. 3650, pp. 1–15. Springer, Heidelberg (2005)
32. Smirnov, A., et al.: Automatic patch generation for buffer overflow attacks. In: Proceedings of the Third International Symposium on Information Assurance and Security, IAS 2007, pp. 165–170
33. Westley, W.: Patches as better bug reports. In: International Conference on Generative Programming and Component Engineering, GPCE 2006
34. Westley, W., et al.: Automatically finding patches using genetic programming*. In: International Conference on Software Engineering, ICSE 2009