A Unit-Based Symbolic Execution Method for Detecting Heap Overflow Vulnerability in Executable Codes

Chapter *in* Lecture Notes in Computer Science · June 2022

DOI: 10.1007/978-3-031-09827-7_6

4 authors, including:

Sara Baradaran
University of Southern California
6 PUBLICATIONS **11** CITATIONS

SEE PROFILE

Mahdi Heidari
Isfahan University of Technology
4 PUBLICATIONS **10** CITATIONS

SEE PROFILE

A Unit-Based Symbolic Execution Method for Detecting Heap Overflow Vulnerability in Executable Codes

Maryam Mouzarani(✉), Ali Kamali, Sara Baradaran, and Mahdi Heidari

Department of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran
mouzarani@iut.ac.ir, {a.kamali,s.baradaran,heidari}@ec.iut.ac.ir

Abstract. Symbolic execution has been a popular method for detecting vulnerabilities of programs in recent years, yet path explosion has remained a significant challenge in its application. This paper proposes a method for improving the efficiency of symbolic execution and detecting heap overflow vulnerability in executable codes. Instead of applying symbolic execution to the whole program, our method initially determines test units of the program, which are parts of the code that might contain heap overflow vulnerability. This is performed through static analysis and based on the specification of heap overflow vulnerability. Then, it applies symbolic execution to the test units and extracts a constraint tree for each unit. Every node in this tree contains the path and vulnerability constraints on the unit input data for executing and overflowing heap buffers in that node. Solving these constraints gives us input values for the test unit that execute the desired nodes and cause heap overflow. Finally, we use curve fitting and treatment learning to approximate the relation between system and unit input data as a function. Using this function, we generate system inputs that enter the program, reach vulnerable instructions in the desired test unit, and cause heap overflow in those instructions. This method is implemented as a plugin for *angr* framework and evaluated using a group of benchmark programs. The experiments show its superiority over similar tools in accuracy and performance.

Keywords: Unit testing · Symbolic execution · Executable codes · Heap overflow · Machine learning

1 Introduction

A wide variety of program analysis and vulnerability detection techniques have been introduced in the past decades, among which symbolic execution has attracted a great deal of attention [10]. Although symbolic execution is theoretically sound and complete [4], it may run into challenges in analyzing real-world programs, such as path explosion. Here, the number of program execution

L. Kovács and K. Meinke (Eds.): TAP 2022, LNCS 13361, pp. 89–105, 2022.
https://doi.org/10.1007/978-3-031-09827-7_6

paths may grow exponentially, making storing and exploring the program paths impractical. Some solutions have been proposed to overcome this challenge such as pruning infeasible paths [20], function and loop summarization [13,14], state merging [13,15], and compiler optimizations [12].

Some researchers have applied machine learning methods to improve symbolic execution and contain path explosion [6–9,11,19]. For instance, in [11], symbolic execution is simply applied to a given program unit rather than the entire program to limit the scope of symbolic analysis and avoid path explosion. In this method, symbolic execution is used to analyze the constraints of execution paths in the unit and calculate appropriate unit input data covering all paths in the test unit. Then, the curve fitting technique [3] is employed to approximate the relationship between system inputs and the given test unit input data. Finally, system inputs are generated that are correlated to the calculated unit input data. This method is not used to detect a specific class of vulnerability, and it does not contain details on how to determine the test units in a program.

This paper extends the idea presented in [11] and proposes a method for detecting heap overflow vulnerability in executable codes. Our method clearly defines how to automatically determine test units in executable code according to our specification of heap overflow vulnerability. We apply symbolic execution to each unit and, given the specification of heap overflow vulnerability, calculate path and vulnerability constraints in each execution path of the unit. We generate unit input data to explore a test unit, reach the vulnerable statements, and cause heap overflow by solving the calculated constraints. Similar to the method in [11], we estimate the relationship between the program input data and that of the test unit by simulating the program execution and using machine learning techniques. In this way, we generate test data that enters into the program from the beginning and activates vulnerability in the desired instruction of the test unit.

Our method has been implemented as a plugin for *angr* framework [23] and is available in [1]. We have evaluated the performance and accuracy of our method using NIST SARD benchmark vulnerable programs [2] and a designed complex program, presented in Listing 2, that contains more functions and more complicated path constraints compared to the benchmark programs. Our solution has been compared with two similar heap overflow detection tools named MACKE [18] and Driller [21]. The experiments show that our method performs more efficiently and accurately than these tools for detecting heap overflow vulnerability.

To summarize, our contributions are as follows:

- Specifying heap overflow vulnerability in executable codes and presenting a method to automatically determine test units in a program accordingly
- Revising the testing algorithm presented in [11] to focus on detecting heap overflow vulnerability more efficiently
- Implementing and evaluating the total solution to demonstrate the advantages of unit-based symbolic execution against similar methods for detecting heap overflow vulnerability

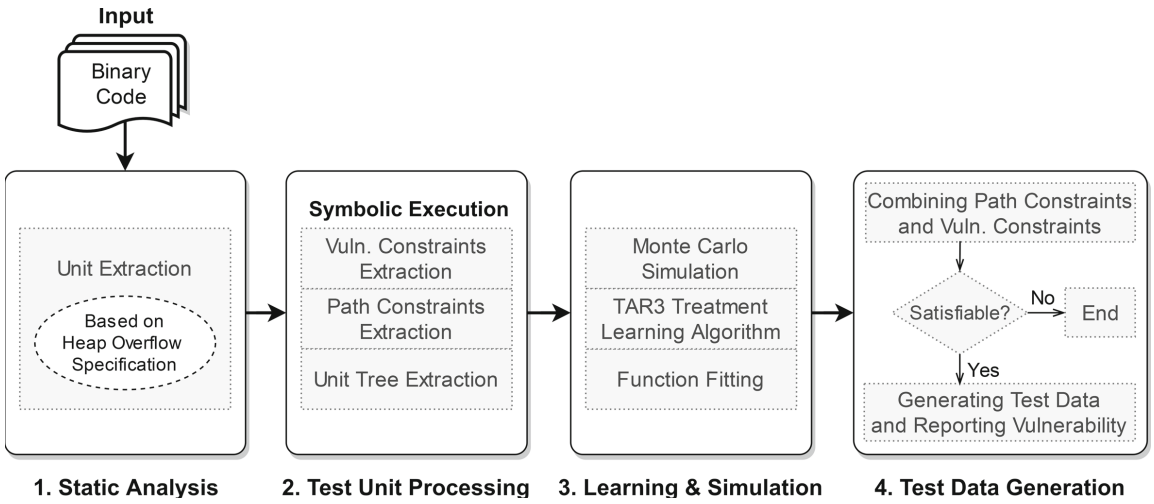

Fig. 1. Architecture of the proposed method

The remainder of this paper is structured as follows: In Sect. 2, the proposed method is described in detail. Section 3 evaluates the implemented method, and finally, Sect. 4 concludes the paper and presents some future works.

2 Method Overview

Our proposed method consists of four major phases, as illustrated in Fig. 1. In the first phase, the program executable code is statically analyzed to identify test units based on the specification of heap overflow vulnerability. To make the process clearer, Fig. 2 illustrates a program containing various possibly vulnerable units for which we explain the steps of our proposed solution briefly. Our method recognizes the test unit, shown in black, statically according to the specification of heap overflow vulnerability. In fact, we are interested in finding unit input data i_k and its relevant system input data I_k that causes heap overflow in the unit. In the second phase, we analyze all execution paths in the unit through symbolic execution and consider the rest of the program as a black box. More precisely, we perform symbolic execution in this phase and create a constraint tree for the extracted unit that contains path and vulnerability constraints on unit input data for each possibly vulnerable statement. In the third phase, Monte Carlo simulation is performed, and the whole program is executed with multiple system input values. If system input I_k reaches the test unit with input value i_k and causes the execution of a node n in the unit tree, we annotate node n with the pair (I_k, i_k) to record which input data causes executing that node. Then, for each possibly vulnerable node in the unit tree, we use function fitting technique [22] to estimate the relation between system and unit input data as a function. Finally, in the fourth phase, we use the calculated path and vulnerability constraints and the estimated function to generate system input data that enters the program, reaches the test unit, and causes heap overflow in vulnerable statements. In the following, we explain each phase in more detail.

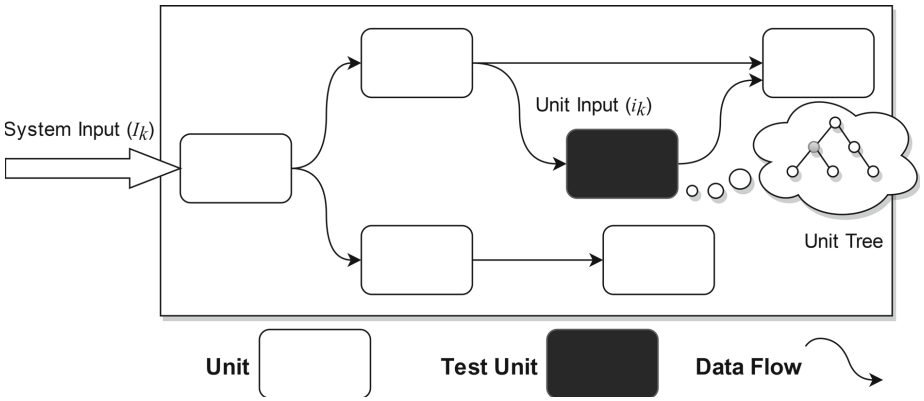

Fig. 2. Schematic of a program as a system containing a vulnerable unit with an input i_k having a relevant system input I_k obtained from curve fitting and treatment learning

2.1 Static Analysis

In the first phase, we analyze the program executable code statically and search for functions that might contain heap overflow vulnerability. To locate possibly vulnerable statements in executable code, we first specify how heap overflow vulnerability appears. We use the general vulnerability specification method presented in [17] to describe vulnerabilities as a sequence of pairs $\{CONT, Rule\}$ that specify the data concerned in a vulnerability and the conditions on it for the vulnerability activation, respectively. We describe heap overflow using this method and based on VEX language since our proposed solution is implemented as a plugin for *angr*, that translates binary instructions into VEX intermediate language. Figure 3 presents our specification of heap overflow as a multi-event vulnerability.

This specification consists of two events: allocating a heap buffer and storing some data into that buffer. Actually, the symbol $\triangleright$ represents the sequence of two events. In this specification, the following containers are considered:

- $CONT1$: address of the allocated heap memory (the address returned from the *malloc* function)
- $CONT2$: length of the allocated heap memory (the input argument of the *malloc* function which is assumed to be a constant value)
- $CONT3$: address on which arbitrary data is stored using a *store* instruction
- $CONT4$: data stored using a *store* instruction

According to the *Rule* illustrated in the second part of the specification, this vulnerability occurs when some data is stored in a heap buffer using the *Ist_Store* VEX instruction. The source and destination of the *store* operation are defined in the *Ist_Store.data.tmp* and *Ist_Store.addr* sections of this VEX instruction, respectively. If length of the source data is more than size of the destination heap buffer, or if $length(CONT4)$ is greater than $CONT2$, then heap overflow occurs. According to this specification, we identify allocated heap buffers ($CONT1$) in

$$(\{CONT1, CONT2\}, True) \triangleright (\{CONT3, CONT4\}, Rule)$$

1. $CONT1 = malloc(CONT2)$
2. $CONT3 = Ist\_Store.addr$
3. $CONT4 = Ist\_Store.data.tmp$

$Rule$:

$$CONT1 \leq CONT3 < CONT1 + CONT2$$
$$and$$
$$len(CONT4) > CONT2$$

Fig. 3. Our specification of heap overflow vulnerability in VEX language

executable codes and search for functions in which some data is stored in these buffers. Such functions are considered as test units. It is worth mentioning that since the address of a heap buffer is dynamically determined at run time, we use the local variable that stores the address of the allocated heap buffer. Since this variable is located in the stack memory, we use it to follow the usage of that buffer through the program statements and functions. One of our challenges in recognizing test units in executable codes is following the usage of the heap buffer in nested function calls. Under such circumstances, the heap buffer address may be sent to other functions as an argument. To overcome this challenge, we use calling conventions to follow the local variable holding address of the heap buffer, which is passed to other functions or returned from function calls. We also assume that the length of the allocated heap buffer in the *malloc* function is a constant value, and it is available during static analysis.

2.2 Processing the Test Units

In this phase, symbolic execution is applied to each unit after determining the test units using static analysis, and a constraint tree is extracted. In this tree, each node is annotated with metadata obtained from symbolic execution that shows the system state at that point of the program. This metadata contains the path constraints from the beginning of the test unit to the given node, the node constraints, and the vulnerability constraints in that node. Vulnerability constraints are calculated based on the length of the heap buffer for nodes in which some data is stored into a heap buffer. These constraints are according to the vulnerability specification in Fig. 3.

2.3 Learning and Simulation Process

After extracting the constraint tree, the program execution is simulated, and its behavior is learned in the third phase of our solution. Details of the operations in this phase are presented in Algorithm 1. This algorithm is a revised version of

Algorithm 1. Cover(S, U, T)

Input: System S with inputs I with $d = |I|$, unit U with inputs i and constraint tree T obtained from applying symbolic execution to the unit U

1: Perform n-factor combinatorial MC simulation over space R^d
2: $(V, v) \leftarrow \{(a, b) |\ a$ is a system level vector and b is the corresponding monitored unit level vector$\}$
3: **for** node n in T using BFS **do**
4: **if** n is in a possibly vulnerable path **then**
5: **if** n and n'siblings are covered **then**
6: $V' \leftarrow \{a \in V |\ a$ cover $n\}$ and $V'' \leftarrow V \backslash V'$
7: $(I_n, R_n, \_) \leftarrow$ RunTar3(I, V, V', V'')
8: $\forall j \in I_n$ store the range $r_j \in R_n$ for j
9: **else if** n is satisfiable but not covered **then**
10: $\boldsymbol{i} \leftarrow$ model for $Const(n)$
11: $C \leftarrow Term(n)$
12: $(I_n, i_n, f_n) \leftarrow$ ComputeMap$(C, V, v, n, Parent(n), \boldsymbol{i})$
13: **end if**
14: **end if**
15: **end for**
16: **for** node n in T that n is possibly vulnerable and satisfiable **do**
17: **if** n is covered **then**
18: Generate input using $Const(n), VulConst(n)$ and $\forall j \in I \backslash I_n$ use $r_j \in R_n$
19: **else**
20: Generate input using $Const(n), VulConst(n)$ and f_n
21: **end if**
22: **end for**

the *Cover* algorithm presented in [11], and our modifications are shown in blue. In this algorithm, the terms $Term(n)$, $Const(n)$, and $VulConst(n)$ refer to the node constraints, the path constraints from the beginning of the test unit to the given node n, and the vulnerability constraints of node n, respectively.

In this algorithm, first in lines 1 and 2, we perform n-factor Monte Carlo simulation on the program and generate possible combinations of input data. We execute the program with these inputs and monitor it to annotate nodes of the unit tree with the system and unit data pairs (I_k, i_k) that reach the aforementioned nodes during the program execution.

Next, in lines 3 and 4, we explore the constraint tree and analyze only nodes located in a possibly vulnerable path, a path from the root to a leaf in the constraint tree that contains some nodes in which a *store* operation to a heap buffer is performed. In contrast to the algorithm in [11], which processes all nodes in the constraint tree at this step, we limit our analysis to a group of nodes according to the vulnerability specification to improve the efficiency of our method. In lines 5 to 8, we check if these nodes and their siblings have been executed during the simulation, then we use *TAR3* treatment learning algorithm [16] to estimate the range of system inputs that could explore the desired node in the unit. Otherwise, in lines 9 to 12, for each uncovered node

whose path constraints are satisfiable, a function named *ComputeMap* is called that estimates the relation between system and unit input data as a function f. The algorithm of this function is presented in Algorithm 2. By solving the path constraints of the node, we generate a unit input data that reaches the node. We give this data to the function f, and it returns appropriate system input that could explore the desired node in the unit tree during the execution.

2.4 Test Data Generation

Until now, we have only considered path constraints in generating system input data. In lines 16 to 22 of Algorithm 1, for each node containing a possibly vulnerable statement whose path constraints are satisfiable, we try to generate system input data consistent with both path and its calculated vulnerability constraints. To do so, in lines 17 and 18, for each node that has been covered in the simulation step, we solve the path and vulnerability constraints of the node and generate appropriate unit input data to cause overflow in that node. In the last step, using the range of system inputs ($r_j \in R_n$) calculated by *TAR3* algorithm, we find relevant system input data for the desired unit input data.

Next, in lines 19 and 20, for each node that has not been covered in the simulation step, we use the fitted function f to find relevant system input data for the unit input data consistent with calculated path and vulnerability constraints.

To summarize the difference between our *Cover* algorithm and the one presented in [11], first in line 4, we improve the performance of our analysis by only considering nodes in potentially vulnerable paths, while in [11], all the nodes are analyzed in this step even though they might not contain any vulnerability. Next, we calculate both path and vulnerability constraints, and this is statically performed using symbolic execution. However, the algorithm in [11] only considers the path constraints calculated gradually using dynamic symbolic execution by generating new input data that explores uncovered paths in the unit. Thus, we calculate the constraints more quickly. Since our symbolic analysis is restricted to a single function, dynamic symbolic execution accuracy and coverage advantages over symbolic execution are not significant here. Finally, we consider the path and vulnerability constraints in lines 16 to 22 for generating system input data that reaches the vulnerable nodes and causes heap overflow. In contrast, the algorithm in [11] only considers the path constraints for generating system input data that covers the nodes of the unit.

ComputeMap Algorithm. We have used the same algorithm, shown in Algorithm 2, as introduced in [11] for the *ComputeMap* function. We describe this algorithm here to make the whole process clear for the reader. Due to the complexity of applying curve fitting to a large set of data and the presence of a large number of parameters, the algorithm fits the program behavior into a function by initially considering the constraints of each node ($Term(n)$) individually. Thus, the unit input variables related to the node constraints and the constraints applied to each variable are first extracted. Based on these variables, a subset

Algorithm 2. ComputeMap($C, I, V, v, n, n', \boldsymbol{i}$)

Input: Constraint C, System vectors V, Unit vectors v, a node n that we want to cover, a node n' that is in the parent hierarchy of n and a model $\boldsymbol{i}$ for $Const(n)$
Output: (I_n, i_n, f_n) where $i_n = Vars(C)$ and $I_n = f(i_n)$

1: $i_n = Vars(C)$
2: $\boldsymbol{i_n}$ = restriction of $\boldsymbol{i}$ to i_n
3: $V' \leftarrow \{a \in V |\ a$ is in 20% of points closet to $Const(n)\}$ and $V'' \leftarrow V \backslash V'$
4: $(I_n, R_n, Smooth) \leftarrow$ RunTar3(I, V, V', V'')
5: **if** *Smooth* **then**
6: Build map $I_n = f(i_n)$ ▷ curve fitting step
7: **else**
8: **if** n' exists **then**
9: $C \leftarrow C \wedge Term(n')$
10: $(I_n, i_n, f_n) \leftarrow$ ComputeMap$(C, I, V, v, n, Parent(n'), \boldsymbol{i})$
11: **else**
12: $n'' \leftarrow n$
13: **while** $Parent(n'')$ exists **do**
14: $C \leftarrow C \wedge Term(Parent(n'')$
15: $n'' \leftarrow Parent(n'')$
16: $V' \leftarrow \{a \in V |\ a$ cover $n''\}$
17: **if** $|V'| \geq Threshold$ **then**
18: break
19: **end if**
20: **end while**
21: $V'' \leftarrow V \backslash V'$
22: $(I_n, R_n, \_) \leftarrow$ RunTar3(I, V, V', V'')
23: $i_n = Vars(C)$
24: Build map $I_n = f(i_n)$ ▷ curve fitting step
25: **end if**
26: **end if**

of unit inputs for which there are path constraints is extracted. Then, the path constraints associated with these inputs are calculated. In the next step, the first 20% of system inputs that are more compatible with this constraints subset are selected as a set V'. Afterward, *TAR3* algorithm is applied to the set, and if a smooth[1] relationship is established there between, the function f is built using curve fitting. Otherwise, the process is repeated recursively by adding parent node constraints to the given node in order to establish a smooth relationship.

If a smooth relationship is not found by including all terms in $Const(n)$, in lines 11 to 24 we walk up through the unit tree to find a parent node with enough system input values in its annotation. Such node is covered in the simulation step with appropriate number of input data (I_k, i_k) that helps to better estimate the function f using the curve fitting algorithm.

[1] The smoothness of a function is a stronger case than the continuity of the function. A *smooth function* is a function having continuous derivatives up to a specific order.

3 Evaluation

The proposed method has been implemented as a plugin for *angr*, a symbolic execution framework for binary analysis [23]. In our implementation, *string* data type is also supported for detecting heap overflow in *string manipulation* functions in addition to *int*, *short*, *unsigned int*, *char*, *float*, *double*, and *enum* data types supported in the proposed approach in [11].

We have designed two experiments to evaluate our solution; in the first experiment, a set of 90 programs from NIST SARD benchmark [2] containing heap overflow vulnerability has been used. The vulnerability occurs in these programs when some constant data is copied into a heap buffer using *strcpy*, *strcat*, *memcpy*, and *memmove* functions. More precisely, out of 90 test programs, 15 programs have the vulnerability in calling *strcpy* function, 15 programs in calling *strcat*, 30 programs in calling *memcpy*, and 30 programs in calling *memmove*. To better evaluate our proposed method, we have made the path constraints in the test programs more complicated by adding an additional *if* statement to the vulnerable paths. In addition, instead of copying constant data into a heap buffer, we have copied an input variable into that buffer to create a vulnerability constraint in the test unit. A vulnerable function in one of these benchmark programs is presented in Listing 1 as an example, and our added *if* statement is underlined in line 16. The same *if* statement has been similarly added to all benchmark programs. In the second experiment, we have created a test program with several functions and more complicated path and vulnerability constraints to better evaluate the efficiency of our method. The source code of this program, along with its details, is presented in Listing 2.

In both experiments, we have compared our implemented solution with two other tools that use the similar method for detecting vulnerabilities in C programs, namely MACKE [18] and Driller [21].

Driller is a fuzzing tool that uses evolutionary algorithms to generate multiple input values from an initial seed and explore the program paths. If the process is trapped in a part of the program because of a conditional statement and the fuzzer fails to generate consistent input values for that condition, symbolic execution is applied to calculate the constraint and generate appropriate data. Then, the fuzzer generates input data based on the obtained data from symbolic execution to detect more in-depth vulnerabilities. Our proposed method is compared with this tool as it applies symbolic execution and uses *angr* framework to improve the coverage of program analysis. Driller is also among the most popular vulnerability detection tools, given its satisfactory performance in detecting vulnerabilities [21].

MACKE is a framework written on top of the KLEE symbolic execution engine [5] for compositional analysis of C programs [18]. In this framework, the program is divided into different units, and symbolic execution is performed for detecting vulnerabilities in each unit. It recognizes each function through static analysis and considers it as a test unit. MACKE also analyzes the call graph and the program control flow graph statically to identify possible function call scenarios. This way determines whether a function containing vulnerable state-

```
1  // Filename: CWE122_Heap_Based_Buffer_Overflow__c_CWE193_char_cpy_34.c
2  void CWE122_Heap_Based_Buffer_Overflow_char_cpy_34_bad(char * source)
3  {
4     char * data; = NULL;
5     struct fp * ptr = NULL;
6     CWE122_Heap_Based_Buffer_Overflow_char_cpy_34_unionType myUnion;
7     // FLAW: Did not allocate space based on the source length
8     data = (char *)malloc(20*sizeof(char));
9     if(data == NULL) { exit(-1); }
10    ptr = (struct fp *)malloc(sizeof(struct fp));
11    if(ptr == NULL) { exit(-1); }
12    myUnion.unionFirst = data;
13    {
14       char * data = myUnion.unionSecond;
15       ptr->fptr = printLine;
16       if(source[0] == '7' && source[1] == '/' && source[2] == '4'
         && source[3] == '2' && source[4] == 'a' && source[5] == '8'
         && source[75] == 'a')
17       {
18          /* POTENTIAL FLAW:
19          data may not have enough space to hold source */
20          strcpy(data, source);
21       }
22       ptr->fptr("That's OK!");
23       printLine(data);
24       free(data);
25       free(ptr);
26    }
27 }
```

Listing 1. A sample vulnerable unit in benchmark programs

ments may be executed in a sequence of possible function calls. After performing symbolic execution in each test unit and calculating appropriate input data that reveals vulnerabilities in the unit, the tool reports possible vulnerabilities in each unit with the relevant unit inputs as the proof of concept. Since MACKE does not consider the constraints of the path from the beginning of the program to the test unit, it may generate several false positives in this step. Therefore, the last analysis step removes alarms related to the units recognized as unreachable during the static analysis. However, since this analysis is only based on the possible function call scenarios, our evaluations demonstrate false positive and negative alarms in MACKE outputs regardless of the path constraints.

3.1 Experiment 1

Table 1 shows the results of our first experiment for testing NIST SARD benchmark programs. The columns in this table represent, from left to right, the number of true positives (TP), true negatives (TN), false positives (FP), false negatives (FN), and the accuracy metric which is calculated as shown in (1).

Table 1. The results of evaluating the approaches on the benchmark programs

Tool	TP	TN	FP	FN	Accuracy
Driller	90	116	0	0	1.00
MACKE	54	108	8	36	0.78
Our method	90	116	0	0	1.00

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Each test program in NIST SARD contains a vulnerable statement in a function whose name contains the word "*bad*" and one or two functions whose names contain the word "*good*" that use similar statements without vulnerabilities. Thus, a precise tool is expected to have one true positive alarm and one or two true negative alarms for each test program. As shown in Table 1, our tool and Driller could precisely detect all vulnerabilities in the test programs with no false alarms. However, MACKE had 8 false positive and 36 false negative alarms in analyzing the benchmark programs.

We have also compared the execution time of the tools in this experiment, as shown in Fig. 4. This figure shows the average analysis time that each tool has spent on the test programs with a vulnerable function, e.g., *strcpy*, *memcpy*, etc. As observed, the performance of our proposed method has significant superiority over the Driller's. Although the analysis time of MACKE in this experiment has been less than that of our tool, it has generated more false alarms and less accurate results. Additionally, MACKE only generates local input data for executing a single unit and does not consider the path constraints on system input data for reaching the beginning of a test unit. On the contrary, our proposed method generates accurate test data for executing the whole program from the beginning and reaching the vulnerable statement in the test unit. This might be one reason that testing and analyzing a program takes more time in our method.

3.2 Experiment 2

In the second experiment, we have analyzed our designed test program with six vulnerable statements, presented in Listing 2, using all three tools. The designed complex program is a simple authentication code by which the user can carry out the sign-up and sign-in operations. The user should run the program and enter the username and password in the console to sign-up. If the condition in line 115 is satisfied, the vulnerable function *signup* would be called. In this function, two heap buffers are allocated in lines 9 and 11. As there are two copy operations with *memcpy* function calls in lines 14 and 17, this function is identified as a test unit by our solution. There is a path constraint, in line 6, in this function, therefore if the input strings for username and password satisfy the path constraints in lines 115 and 6, and the length of them be more than the length of the destination heap buffers in the copy operations, they would

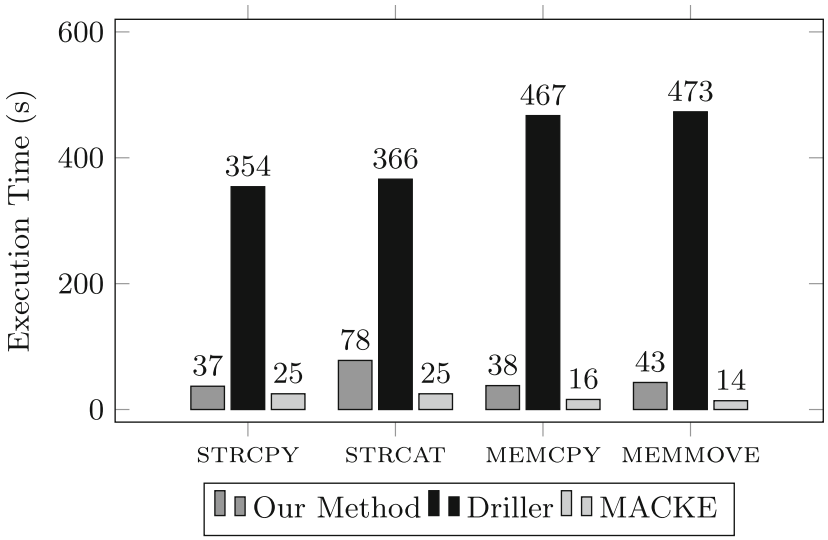

Fig. 4. Comparison of tools analysis time on the benchmark programs

cause heap overflow. Note that the path constraint in line 115 is out of the test unit and should be recognized through machine learning. Our implemented solution calculates the path constraint in line 6 using symbolic execution since this constraint is inside the unit. Then, it generates appropriate input data for the *scanf* operations in lines 113 and 114, consistent with both path constraints inside and outside the unit. There are two other test units in this program, *check* and *authentication* functions, that cause heap overflow by calling *strcpy* and *memcpy* functions respectively. The same challenge exists in these functions for our solution to calculate the path constraints inside the test unit and estimate the ones outside of it.

The results of this experiment are demonstrated in Table 2. As shown in this table, MACKE could detect only one of these vulnerabilities in the given program as it seems that could not analyze complicated path conditions. Driller could detect four vulnerabilities in the test program, and it has generated two false negative alarms. In contrast, our tool has detected all six vulnerabilities precisely. It has generated appropriate test data for the whole program that enters the program from the beginning and causes heap overflow in the vulnerable statements of the test units. Besides, the testing time in our tool has been much less than that of Driller. Therefore, the results of this experiment demonstrate the advantage of restricting the scope of symbolic execution for detecting a specific vulnerability class.

Table 2. The results of evaluating the approaches on the complex test program

Tool	TP	TN	FP	FN	Time(s)
Driller	4	0	0	2	3549
MACKE	1	0	0	5	335
Our method	6	0	0	0	374

```
#define def_user "admin"
#define def_pass "password"
// The function "signup" as a vulnerable unit
void signup(char *username, char *password)
{
   if((username[1] >= '0' && username[1] <= '9')
    && !strncmp(password, "passW0rd", 8))
   {
      // FLAW: Did not allocate space based on the username length
      char *tmp_user = (char *)(malloc(50*sizeof(char)));
      // FLAW: Did not allocate space based on the password length
      char *tmp_pass = (char *)(malloc(50*sizeof(char)));
      /* POTENTIAL FLAW:
      data may not have enough space to hold source */
      memcpy(tmp_user, username, strlen(username));
      /* POTENTIAL FLAW:
      data may not have enough space to hold source */
      memcpy(tmp_pass, password, strlen(password));
      if(strlen(tmp_pass) < 12)
      {
         printf("The selected password is too weak\n");
         return;
      }
      int fd = open(tmp_user, O_WRONLY|O_CREAT, 0777);
      if(fd < 0)
      {
         printf("An unexpected problem occurred!\n");
         return;
      }
      write(fd,tmp_pass, sizeof(tmp_pass));
      printf("%s your registration was successful\n", tmp_user);
   }
   else if(!(username[1] >= '0' && username[1] <= '9'))
      printf("The second letter of username must be a number\n");
   else
      printf("The password must start with the word <passW0rd>\n");
}
// The function "check" as a vulnerable unit
bool check(char *username, char *password)
{
   // FLAW: Did not allocate space based on the username length
   char *tmp_user = (char *)(malloc(50*sizeof(char)));
   // FLAW: Did not allocate space based on the password length
   char *tmp_pass = (char *)(malloc(50*sizeof(char)));
   if((username[0] >= 'A' && username[0] <= 'Z')
    && (username[1] >= '0' && username[1] <= '9'))
   {
      /* POTENTIAL FLAW:
      data may not have enough space to hold source */
      strcpy(tmp_user, username);
      /* POTENTIAL FLAW:
      data may not have enough space to hold source */
      strcpy(tmp_pass, password);
```

```
52       if(!strcmp(tmp_user, def_user) && !strcmp(tmp_pass, def_pass))
53          return true;
54       char passwd[50];
55       int fd = open(tmp_user, O_RDONLY);
56       if(fd < 0)
57       {
58          printf("An unexpected problem occurred!\n");
59          return false;
60       }
61       read(fd, passwd, sizeof(passwd));
62       if(!strcmp(passwd, tmp_pass)) { return true; }
63    }
64    return false;
65 }
66 // The function "signin" without any vulnerable statement
67 bool signin(char *username, char *password)
68 {
69    if(check(username, password))
70    {
71       printf("%s you logged in successfully\n", username);
72       return true;
73    }
74    else
75    {
76       printf("The username or password is wrong\n");
77       return false;
78    }
79 }
80 // The function "authentication" as a vulnerable unit
81 void authentication(char *username, char *password)
82 {
83    // FLAW: Did not allocate space based on the username length
84    char *tmp_user = (char *)(malloc(80*(sizeof(char))));
85    // FLAW: Did not allocate space based on the password length
86    char *tmp_pass = (char *)(malloc(80*(sizeof(char))));
87    /* POTENTIAL FLAW:
88    data may not have enough space to hold source */
89    memcpy(tmp_user, username, strlen(username));
90    /* POTENTIAL FLAW:
91    data may not have enough space to hold source */
92    memcpy(tmp_pass, password, strlen(password));
93    int loginCnt = 0;
94    for(; loginCnt < 3; loginCnt++)
95    {
96       bool signin_res = signin(tmp_user, tmp_pass);
97       if(signin_res) { break; }
98       printf("The username or password is invalid, try again :");
99       printf("(%d from %d)\n",(loginCnt+1),3);
100      printf("Enter username : "); scanf("%s",tmp_user);
101      printf("Enter password : "); scanf("%s",tmp_pass);
102   }
103   if(loginCnt == 3) { printf("Please try later\n"); }
104 }
```

```
105 int main (int argc, char *argv[])
106 {
107    char *username = (char *)(malloc(100*(sizeof(char))));
108    char *password = (char *)(malloc(100*(sizeof(char))));
109    if(argc >= 3) { authentication(argv[1], argv[2]); }
110    else
111    {
112       printf("Register new user\n");
113       printf("Enter username :"); scanf("%s", username);
114       printf("Enter password :"); scanf("%s", password);
115       if(username[0] >= 'A' && username[0] <= 'Z')
116          signup(username, password);
117       else
118          printf("The selected username is not valid,
                it must start with an uppercase letter");
119    }
120 }
```

Listing 2. Source code of the designed complex program

4 Conclusion and Future Works

While symbolic execution is sound and complete in theory, this method faces challenges in testing real-world programs, such as path explosion. The number of symbolic execution states may be exponential, and the whole program may not be analyzed thoroughly.

We proposed a method for applying symbolic execution technique to detect heap overflow vulnerability in executable codes. In this method, we limit the scope of symbolic execution to the test units. We also presented a method for determining the test units in the program according to the specification of heap overflow vulnerability in executable codes. In this method, we generate appropriate unit input data for detecting heap overflow according to the path and vulnerability constraints in vulnerable statements of the test unit. Then, we use machine learning techniques to estimate the relation between system and unit input data as a function and find consistent system input data that enters into the program from the beginning, causes execution of vulnerable statements in the test unit, and reveals heap overflow vulnerability. The experiments showed that this method achieves more efficient and accurate results in detecting vulnerabilities in complex programs compared to similar tools.

In the future, we are going to extend our solution to detect other vulnerability classes in executable codes, such as stack-based buffer overflow, use after free, and double free. We have to specify these vulnerabilities so that the implemented solution can automatically identify test units based on them. We also have to revise the *Cover* algorithm to calculate the vulnerability constraints based on the specified vulnerabilities and generate appropriate test data to detect them in the programs. Additionally, we are going to study other machine learning techniques for estimating the program behavior to improve the efficiency of the *ComputeMap* algorithm.

References

1. Heap Overflow Detection Tool. https://github.com/SoftwareSecurityLab/Heap-Overflow-Detection
2. National Institute of Standards and Technology in Software Assurance Reference Dataset Project. https://samate.nist.gov/SRD. Accessed 4 Mar 2022
3. Arlinghaus, S.L., Arlinghaus, W.C., Drake, W.D., Nystuen, J.D.: Practical Handbook of Curve Fitting (1994)
4. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3) (2018). https://doi.org/10.1145/3182657
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008, pp. 209–224. USENIX Association (2008). https://doi.org/10.5555/1855741.1855756
6. Cha, S., Hong, S., Bak, J., Kim, J., Lee, J., Oh, H.: Enhancing dynamic symbolic execution by automatically learning search heuristics. IEEE Trans. Softw. Engi., 1 (2021). https://doi.org/10.1109/TSE.2021.3101870
7. Cha, S., Lee, S., Oh, H.: Template-guided concolic testing via online learning, pp. 408–418. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3238147.3238227
8. Cha, S., Oh, H.: Concolic testing with adaptively changing search heuristics. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, pp. 235–245. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3338906.3338964
9. Chen, J., Hu, W., Zhang, L., Hao, D., Khurshid, S., Zhang, L.: Learning to accelerate symbolic execution via code transformation. In: Millstein, T. (ed.) 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 109, pp. 6:1–6:27. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2018). https://doi.org/10.4230/LIPIcs.ECOOP.2018.6
10. Chen, T., Zhang, X.S., Guo, S.Z., Li, H.Y., Wu, Y.: State of the art: dynamic symbolic execution for automated test generation. Future Gener. Comput. Syst. **29**(7), 1758–1773 (2013). https://doi.org/10.1016/j.future.2012.02.006
11. Davies, M., Păsăreanu, C.S., Raman, V.: Symbolic execution enhanced system testing. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 294–309. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_23
12. Dong, S., Olivo, O., Zhang, L., Khurshid, S.: Studying the influence of standard compiler optimizations on symbolic execution. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 205–215 (2015). https://doi.org/10.1109/ISSRE.2015.7381814
13. Godefroid, P.: Compositional dynamic test generation. SIGPLAN Not. **42**(1), 47–54 (2007). https://doi.org/10.1145/1190215.1190226
14. Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA 2011, pp. 23–33. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/2001420.2001424

15. Hansen, T., Schachte, P., Søndergaard, H.: State joining and splitting for the symbolic execution of binaries. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 76–92. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04694-0_6
16. Menzies, T., Hu, Y.: Data mining for very busy people. Computer **36**(11), 22–29 (2003). https://doi.org/10.1109/MC.2003.1244531
17. Mouzarani, M., Sadeghiyan, B.: Towards designing an extendable vulnerability detection method for executable codes. Inf. Softw. Technol. **80**, 231–244 (2016). https://doi.org/10.1016/j.infsof.2016.09.004
18. Ognawala, S., Ochoa, M., Pretschner, A., Limmer, T.: MACKE: compositional analysis of low-level vulnerabilities with symbolic execution. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pp. 780–785. Association for Computing Machinery, New York (2016). https://doi.org/10.1145/2970276.2970281
19. Păsăreanu, C.S., et al.: Combining unit-level symbolic execution and system-level concrete execution for testing Nasa software, ISSTA 2008, pp. 15–26. Association for Computing Machinery, New York (2008). https://doi.org/10.1145/1390630.1390635
20. Schwartz-Narbonne, D., Schäf, M., Jovanović, D., Rümmer, P., Wies, T.: Conflict-directed graph coverage. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 327–342. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_23
21. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: In: NDSS (2016). https://doi.org/10.14722/ndss.2016.23368
22. Strang, G.: Linear Algebra and Its Applications. Thomson, Brooks/Cole, Belmont (2006). http://www.amazon.com/Linear-Algebra-Its-Applications-Edition/dp/0030105676
23. Wang, F., Shoshitaishvili, Y.: Angr - the next generation of binary analysis. In: 2017 IEEE Cybersecurity Development (SecDev), pp. 8–9 (2017). https://doi.org/10.1109/SecDev.2017.14