IET Research Journals

Submission Template for IET Research Journal Papers

IET Journals
The Institution of Engineering and Technology

ISSN 1751-8644
doi: 0000000000
www.ietdl.org

A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes

Sara Baradaran[1]*, *Mahdi Heidari*[1], *Ali Kamali*[1], *Maryam Mouzarani*[1]

[1] *Department of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran*
* *E-mail: mouzarani@iut.ac.ir*

Abstract:
Memory corruption is a serious class of software vulnerabilities, which requires careful attention to be detected and removed from applications before getting exploited and harming the system users. Symbolic execution is a well-known method for analyzing programs and detecting various vulnerabilities, e.g., memory corruption. Although this method is sound and complete in theory, it faces some challenges, such as path explosion, when applied to real-world complex programs. In this paper, we present a method for improving the efficiency of symbolic execution and detecting four classes of memory corruption vulnerabilities in executable codes, i.e., heap-based buffer overflow, stack-based buffer overflow, use-after-free, and double-free. We perform symbolic execution only on test units rather than the whole program to avoid path explosion. In our method, test units are considered parts of the program's code, which might contain vulnerable statements and are statically identified based on the specifications of memory corruption vulnerabilities. Then, each test unit is symbolically executed to calculate path and vulnerability constraints of each statement of the unit, which determine the conditions on unit input data for executing that statement or activating vulnerabilities in it, respectively. Solving these constraints gives us input values for the test unit, which execute the desired statements and reveal vulnerabilities in them. Finally, we use machine learning to approximate the correlation between system and unit input data. Thereby, we generate system inputs that enter the program, reach vulnerable instructions in the desired test unit, and reveal vulnerabilities in them. 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.

1 Introduction

Memory corruption vulnerabilities are prevalent and detrimental software weaknesses, which potentially occur when programming with low-level languages (usually C and C++). These vulnerabilities allow attackers to access the program's memory directly and read from or write to it arbitrary values. Although low-level languages provide few security guarantees [16], their flexibility and efficiency encourage programmers to use them in developing critical software, such as operating system kernels, drivers, and OS services. Therefore, exploiting memory corruption vulnerabilities might seriously harm the system owners.

A wide variety of program analysis and vulnerability detection methods have been introduced over the past decades. Among them, symbolic execution is a promising one, which systematically explores the program execution paths with high coverage. In this method, input values are represented as symbols to calculate symbolic constraints on input data for each execution path and generate sufficiently diverse test data for analyzing a program's possible behaviors [10]. Although symbolic execution is theoretically sound and complete [4], it may run into challenges in analyzing real-world programs. A well-known example of these challenges is path explosion that emerges since the number of program execution paths grows exponentially, and it makes storing and exploring the paths of large programs infeasible.

Some researchers have applied machine learning techniques to improve symbolic execution and prevent path explosion [6–9, 11, 15]. For instance, in [11], symbolic execution is applied to a test unit rather than the entire program to limit the scope of symbolic analysis and avoid path explosion. In the suggested method, a combination of concrete and symbolic execution is applied to calculate the constraints of execution paths in each test unit and generate appropriate test data to explore these paths by solving the calculated constraints. Then, curve fitting technique [3] is employed to approximate the correlation between system inputs and the test unit inputs as a function. Using this function, new system input data are generated to reach the test units and traverse various execution paths in them. 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.

We extend the idea presented in [11] and propose a method for detecting four classes of memory corruption vulnerabilities in executable codes, i.e., heap-based buffer overflow, stack-based buffer overflow, use-after-free, and double-free. We present formal specifications for these vulnerability classes to be used to automatically locate test units in executable codes. Given the specification of a vulnerability class, test units are symbolically executed to calculate path and vulnerability constraints of the desired execution paths in each test unit. Then, a set of unit input data is generated by solving the calculated constraints to explore test units, reach vulnerable statements, and activate their vulnerabilities. Similar to the method in [11], we estimate the relationship between the program and unit inputs by simulating the program execution and using machine learning techniques. Thereby, we generate test data that enter into the program from the beginning and reveal vulnerabilities in the desired instructions of a test unit.

The proposed method is implemented as a plugin for *angr* [16], which is a flexible, modular, and scalable binary analysis framework, supporting a variety of architectures. We have named our implemented method *UbSym* as it employs **U**nit-**b**ased **Sym**bolic execution for detecting vulnerabilities in executable codes. The performance and accuracy of *UbSym* have been evaluated using a group of NIST SARD benchmark vulnerable programs [1] and a number of complex programs that contain more functions and more complicated path constraints in comparison with the benchmark programs. We have also compared *UbSym* with two tools, MACKE [14] and Driller [17], which detect memory corruption vulnerabilities with similar methods. The experimental results show that our method

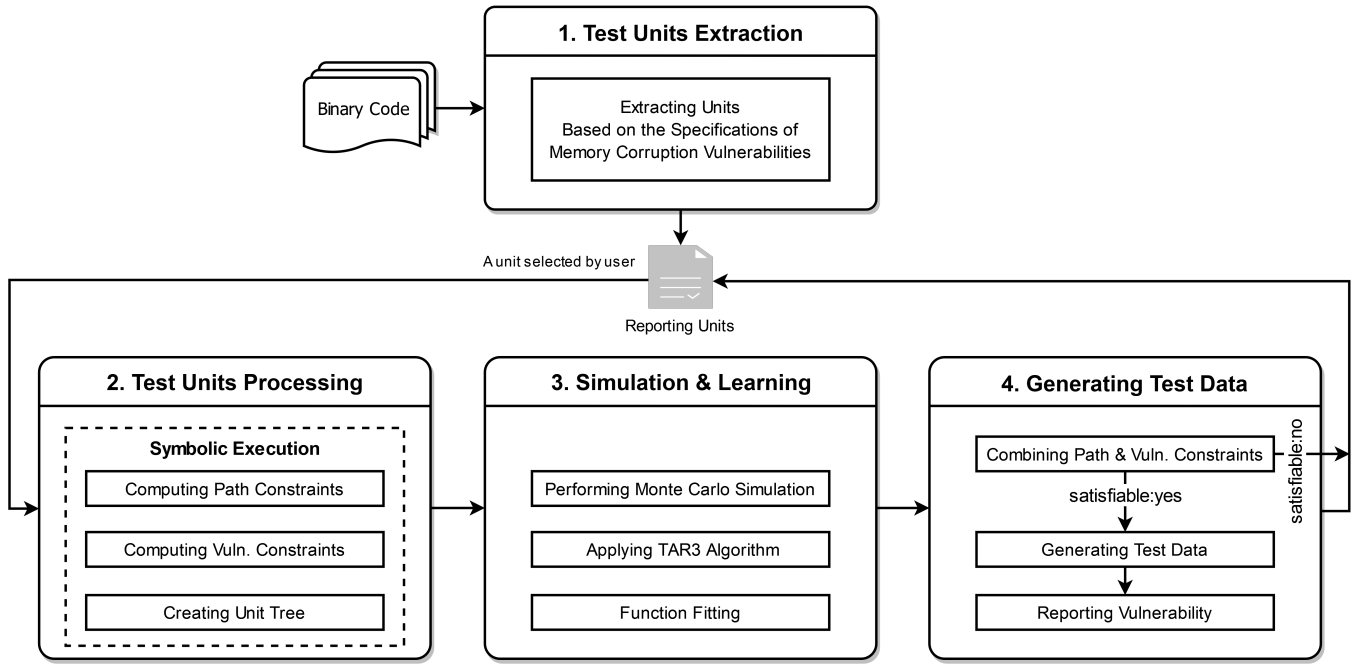


Fig. 1: Architecture of the proposed method

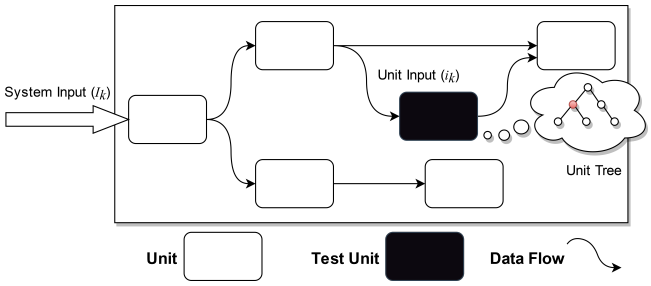


Fig. 2: Schema of a program as a system containing a vulnerable unit with an input i_k having a relevant system input I_k obtained from curve fitting and treatment learning

performs more efficiently than these tools for detecting such vulnerabilities. The source code of *UbSym*, including the test cases and scripts to automatically reproduce test results, is publicly available in our github repository [2].

In summary, this paper makes the following contributions:

- Specifying four vulnerability classes, i.e., stack-based buffer overflow, heap-based buffer overflow, double-free, and use-after-free, in executable codes and presenting a method to automatically determine test units for each vulnerability class in a program accordingly.
- Revising the algorithm presented in [11] to calculate path and vulnerability constraints based on our vulnerability specifications and focus on detecting memory corruption vulnerabilities more efficiently.
- Implementing and evaluating the total solution to demonstrate the effectiveness of unit-based symbolic execution against similar methods for detecting memory corruption vulnerabilities.

The remainder of this paper is structured as follows: In Section 2, the proposed method is described in detail. Section 3 explains our experiments and evaluates the implemented method, and finally, Section 4 concludes the paper.

2 Method overview

Our proposed method is illustrated in Fig. 1. It consists of four phases: test unit extraction, test unit processing, learning and simulation, and generating test data.

In the first phase, the program's executable code is statically analyzed to identify test units based on the specification of each memory corruption vulnerability. To make the process clearer, Fig. 2 illustrates a program containing various units, which *UbSym* identifies its test units, shown in black, during the static analysis in the first step. To dynamically detect vulnerabilities in such a program, we are interested in finding an input data i_k for a test unit and its relevant system input data I_k that causes vulnerability activation in the suspicious statements of that unit. Thus, in the second phase, we analyze the test units with symbolic execution and consider the rest of the program a black box. In this step, we generate a constraint tree for each test unit, which contains path constraints on unit input data for each node and vulnerability constraints on unit input data for nodes suspected to have memory corruption vulnerability. In the third phase, we perform Monte Carlo simulation and execute the whole program with multiple system input values. If system input I_k reaches the test unit with input value i_k and causes the execution of node n in the unit constraint tree, we annotate node n with the pair (I_k, i_k) to record which input data cause executing that node. Then, for each possibly vulnerable node in the unit tree whose constraints are satisfiable, we use function fitting technique [18] 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 appropriate system input data that enter the program, reach the test unit, and cause vulnerability activation in vulnerable statements. In the following, we explain each phase in more detail.

2.1 Test units extraction

In the first phase, the program's executable code is statically analyzed to search for functions that probably contain vulnerable statements. In order to locate possible vulnerabilities, we first specify how each vulnerability class appears in executable codes. We use the general vulnerability specification method presented in [13] to describe memory corruption vulnerabilities. In this method, a vulnerability is described as a single or a sequence of events. Each event

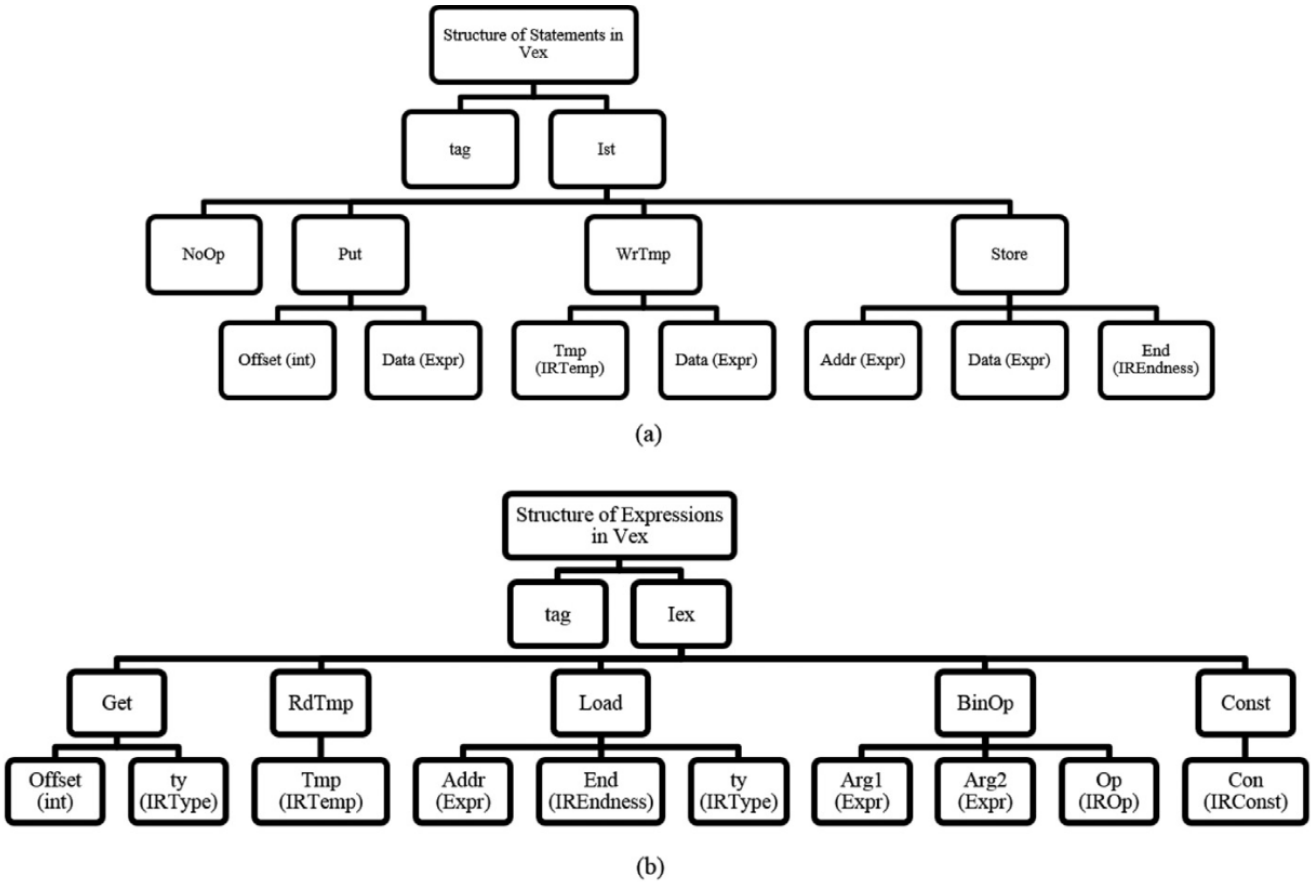


Fig. 3: The structure of statements and expressions in VEX. Part (a) shows the structure of some of the statements in VEX. Part (b) shows the structure of some of the expressions in VEX [13].

is represented as a pair of {CONT, Rule}, which {CONT} defines the location of data in a program statement that is of concern in a vulnerability class and the {Rule} defines the conditions on CONT data, which result in intended vulnerability activation.

It is worth mentioning that since we implement our method in *angr* framework and it translates binary instructions into VEX intermediate language, we define the containers and rules in our specifications based on the structure of VEX language. *Angr* translates each assembly instruction into one or multiple statements in VEX language. VEX statements have different types, such as `Store`, `WrTmp`, etc. In addition, each statement might contain one or more expressions of different types, such as `Load`, `Get`, etc. Fig. 3 presents the structure of the most common statements and expressions in VEX. Using the proposed method in [13] and based on VEX intermediate language, we specify four multi-event vulnerability classes, i.e., stack-based buffer overflow, heap-based buffer overflow, double-free, and use-after-free. In our specification, the symbol $\triangleright$ represents the order of events.

2.1.1 *Test unit of heap-based buffer overflow vulnerability:*

Our proposed specification for heap-based buffer overflow vulnerability is presented in Fig. 4. This specification consists of two events: allocating a heap buffer and storing some data in that buffer.

In the first event of this vulnerability specification, we locate `malloc` function calls and consider their length arguments and return values as `CONT2` and `CONT1`, respectively. Here, `CONT1` represents the local variable storing address of the allocated buffer.

In the second event of the specification, we look for `Store` statements and check the source and destination of these statements. To clarify the specification, the detailed structure of `Store` statement is represented in green in Fig. 4. As shown in this structure, the source and destination of the `Store` statement are defined in the `Store.data` and `Store.addr` sections, respectively. Also, the `Store.data` section is an expression with type `RdTmp`, in

({CONT1,CONT2},True) $\triangleright$ ({CONT3,CONT4},Rule1)

1. `CONT1 = malloc(CONT2)`
2. `STle(CONT3) = CONT4`

```
Container[HOF].tag = Ist_Store
Container[HOF].Store.addr = CONT3
Container[HOF].Store.data->tag = Iex_RdTmp
Container[HOF].Store.data->Iex.RdTmp.tmp = CONT4
```

Rule1:

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

Fig. 4: Specification of heap-based buffer overflow vulnerability in VEX language

which the `RdTmp.tmp` part refers to the temporary variable containing the source data of the store operation and is considered as `CONT4`. Moreover, the destination buffer address, which is defined by `Store.addr` part of the `Store` statement, is considered as `CONT3`. Regarding to Rule1, heap-based buffer overflow occurs when a store operation is performed to a memory location in the range of an allocated heap buffer and length of the source data is more than size of the destination heap buffer, or `length(CONT4)` > `CONT2`.

To determine the test units, we identify allocated heap buffers (`CONT1`) in executable codes and search for functions in which some data is stored in these buffers. Such functions are considered test units for this vulnerability. It is worth mentioning that since the address of a heap buffer is dynamically assigned at run time, we refer to the local pointer that stores the address of the allocated heap buffers in our static analysis. Since this pointer is a local variable and located in the stack memory, it is statically available and helps to follow the usage of heap buffers throughout the program statements and

```
1 void func()          func:
2 {                       push  rbp
3    /*buff definition*/  mov   rbp, rsp
4    char buff[20];       lea   rax, [rbp-32]
5    /*buff usage*/       mov   DWORD PTR [rax], 7102838
6    strcpy(buff, "Val"); pop   rbp
7 }                       ret
```

Fig. 5: An example of a C code and its equivalent assembly code for using stack memory based on x64 architecture

functions. We also assume that the length of the allocated heap buffer in the `malloc` function is a constant value and is available during the static analysis.

Another issue in static tracing of heap buffers arises in nested function calls. A heap buffer may be sent as an argument to other functions, and thus we need to trace it among the stacks of called functions. As a solution, for heap buffers passed to other functions or returned from function calls, we use the calling conventions in assembly languages to locate the local pointers holding the addresses of these buffers. The same challenge exists in identifying test units of the other memory corruption vulnerability classes, which is solved similarly.

2.1.2 Test unit of stack-based buffer overflow vulnerability: Identifying test units for stack-based buffer overflow vulnerability is more challenging in comparison with heap-based buffer overflow. This challenge arises since there is no specific instruction in executable codes for defining stack buffers. When a buffer is defined in the high-level source code, no instruction is accordingly added to the corresponding executable code. Therefore, unlike heap buffers, which are explicitly allocated by calling a dynamic memory allocation function, it is not possible to exactly determine the size and location of a stack buffer based on a specific instruction in executable codes.

As a solution, we estimate the location and maximum length of stack buffers according to the calling conventions and the standards of accessing stack variables in executable codes. When a function is called, its arguments, the return address (`rip`), and the base frame pointer of the caller function (`rbp`) are respectively pushed into the stack. Then, local variables of the called function are placed into the stack. The local variables of each function are accessed using the base frame pointer of its stack memory, which is stored in a specific register, e.g., `rbp` in x64 instruction set architecture. In this way, the start address of a stack buffer is calculated as `rbp-x`, which `x` helps to estimate the length of that buffer. If some data larger than `x` bytes is stored in such stack buffer, it overwrites not only other local variables but also the contents of the base frame pointer and probably the return address in the stack. Thus, we locate stack buffers using instructions that access memory buffers with address `rbp-x`, and estimate their maximum lengths as `x`.

As an example, Fig. 5 presents the high-level code of a function in C language and its equivalent x64 assembly code. As shown in this figure, the start address of `buffer` is `rbp-32`, and thus it is 32 bytes far from the stored base frame pointer in the stack. Therefore, we estimate the maximum length of `buffer` as 32 bytes.

A drawback of our method in estimating the buffer length is that we can only detect stack overflows that corrupt the `rbp` value. If stack-based buffer overflow occurs in a way that only local variables within the function are corrupted, but the overflowing data could not be long enough to overwrite the `rbp` value in the stack, our method is not able to detect the vulnerability.

To sum up, our proposed specification for stack-based buffer overflow vulnerability is presented in Fig. 6. This specification consists of three events: accessing the `rbp` register, accessing a stack buffer with address `rbp-x`, and storing some data in a stack buffer.

In the first event of the specification, we consider statements in which the program accesses the value of `rbp` register and stores it in a temporary variable. In this event, the value of `rbp` register is extracted and stored in a temporary variable using a `WrTmp` VEX statement. The source and destination of `WrTmp` are defined by `WrTmp.data` and `WrTmp.tmp` parts of this VEX

(CONT1,True) ▷ ({CONT2,CONT3},Rule1) ▷ ({CONT5,CONT6},Rule2)

1. `CONT1 = GET:I64(20)`
   `Container[SOF].tag = Ist_WrTmp`
   `Container[SOF].WrTmp.tmp = CONT1`
   `Container[SOF].WrTmp.data->tag = Iex_Get`
   `Container[SOF].WrTmp.data->Iex.Get.offset = 20`
2. `CONT4 = Add64(CONT2, CONT3)`
   `Container[SOF].tag = Ist_WrTmp`
   `Container[SOF].WrTmp.tmp = CONT4`
   `Container[SOF].WrTmp.data->tag = Iex_Binop`
   `Container[SOF].WrTmp.data->Iex.Binop.op = Iop_Add64`
   `Container[SOF].WrTmp.data->Iex.Binop.arg1 = CONT2`
   `Container[SOF].WrTmp.data->Iex.Binop.arg2 = CONT3`
3. `STle(CONT5) = CONT6`
   `Container[SOF].tag = Ist_Store`
   `Container[SOF].Store.addr = CONT5`
   `Container[SOF].Store.data->tag = Iex_RdTmp`
   `Container[SOF].Store.data->Iex.RdTmp.tmp = CONT6`

Rule1:

$$CONT1 == CONT2 \wedge CONT3 < 0$$

Rule2:

$$CONT4 \leq CONT5 < CONT2 \wedge len(CONT6) > |CONT3|$$

Fig. 6: Specification of stack-based buffer overflow vulnerability in VEX language

statement, respectively. According to the detailed structure of this statement, `rbp` register is accessed with a `WrTmp` statement that its `WrTmp.data` section is an expression of type `Get` in which `Get.offset` is equal to 20. We consider `WrTmp.tmp` as `CONT1` to record the temporary variable that stores the value of `rbp` register.

In the second event, we look for statements that compute the value of `rbp-x`. This operation is performed in a `WrTmp` statement in which the `WrTmp.data` section is a `Binop` expression of type `Add64`. The first argument of this operation, which is defined by `Binop.arg1`, is considered as `CONT2`, and the second one, defined by `Binop.arg2`, is considered as `CONT3`. Rule1 states that the value of `CONT2` must be equal to the temporary variable in which the content of the `rbp` register was stored in the previous step (`CONT1`). Also, `CONT3` must be a negative number as `rbp-x` is calculated using the `Add64` operator. The absolute value of the second argument in the `Binop` expression (|`CONT3`|) is assumed as the maximum size of the buffer. The result of this operation is stored in `WrTmp.tmp` part and is considered as `CONT4`.

In the third event, we locate `Store` statements that store some data in a stack buffer. In this statement, `Store.data` is an expression of type `RdTmp`, and the temporary variable storing the source data, defined by `RdTmp.tmp` section, is considered as `CONT6`. The destination buffer address is also defined by `Store.addr` part of the `Store` statement and is considered as `CONT5`. According to Rule2, stack-based buffer overflow occurs when the destination address in the `Store` statement (`CONT5`) is in the range of a stack buffer and the source data (`CONT6`) is larger than the maximum size of the destination stack buffer.

We use this specification and locate stack buffers throughout the program's executable code and consider the functions in which a store operation is performed on a stack buffer as test units. It is worth mentioning that we use the same method as in the previous section to trace the stack buffers in nested function calls.

2.1.3 Test unit of double-free vulnerability: Our proposed specification for double-free vulnerability is presented in Fig. 7. This vulnerability arises in a scenario with an allocation of a heap buffer and two times freeing of that buffer.

For the first event, we locate all `malloc` function calls to determine the addresses of pointers that point to the allocated buffers and consider them as `CONT1`. For the second and third events, we find all `free` function calls in the program's executable code, and extract their input argument demonstrating the pointer address of the released buffer as `CONT3` and `CONT4`. According to Rule1 and

({CONT1,CONT2},True) ▷ (CONT3,Rule1) ▷ (CONT4,Rule2)

```
1. CONT1 = malloc(CONT2)
2. free(CONT3)
3. free(CONT4)

Rule1:
                CONT1 ≤ CONT3 < CONT1 + CONT2
Rule2:
                CONT1 ≤ CONT4 < CONT1 + CONT2
```

Fig. 7: Specification of double-free vulnerability in VEX language

```
1  char * child1(char * source)
2  {
3      source = (char *)malloc(10*sizeof(char));
4      return source;
5  }
6  void child2(char * source, int n)
7  {
8      char * src_copy;
9      src_copy = (char *)malloc(n*sizeof(char));
10     strncpy(src_copy, source, n);
11     free(source);
12     /* some lines of code */
13     free(src_copy);
14 }
15 void parent()
16 {
17     int n = 10; char * source;
18     source = child1(source);
19     child2(source, n);
20     /* FLAW: freeing memory twice */
21     free(source);
22 }
23 int main(void)
24 {
25     parent(); return 0;
26 }
```

Fig. 8: An example of a C code having double-free vulnerability

Rule2, double-free vulnerability occurs when the memory address of a heap buffer is freed by calling the `free` function twice.

The test unit for double-free vulnerability is identified as a function that consists of the three events related to this specification, i.e., buffer allocation, buffer deletion, and buffer re-deletion. As an example, in the sample C code in Fig. 8, the memory allocation operation is performed in function `child1`, then the allocated heap buffer is deleted in function `child2`, and again deleted in function `parent`. In this example, `parent` is the function that contains all three events of the vulnerability specification, which happen sequentially. Hence, `parent` is considered a test unit for double-free vulnerability in this code.

2.1.4 Test unit of use-after-free vulnerability: Our proposed specification for use-after-free vulnerability is presented in Fig. 9. This vulnerability appears in a scenario with three events: allocating a heap buffer, deleting the same buffer, and then using that buffer in a read operation with a `WrTmp` statement or in a store operation with a `Store` statement. The `Not_Imp` value in this specification is assigned to the parts of the statements, which are not important.

According to this specification, we consider functions that contain `malloc`, `free`, and load or store operations on the same heap buffer as a test unit.

2.2 Processing the test units

In this phase, we perform symbolic execution for the extracted test units. For each test unit, a constraint tree is generated that represents basic blocks of the program as its nodes. We annotate each node with some metadata that indicates the system state at that point of the program and contains the path constraints from the beginning of the test unit to the given node, the node constraints, and vulnerability constraints. In the following, we use $Term(n)$, $Const(n)$,

({CONT1,CONT2},True) ▷ (CONT3,Rule1) ▷ ((CONT4,Rule2) | (CONT5,Rule3))

```
1. CONT1 = malloc(CONT2)
2. free(CONT3)
3. Not_Imp = LDle:I64(CONT4)
   Cont[UAF].tag = Ist_WrTmp
   Container[UAF].WrTmp.tmp = Not_Imp
   Container[UAF].WrTmp.data->tag = Iex_Load
   Container[UAF].WrTmp.data->Iex.Load.addr = CONT4
4. STle(CONT5) = Not_Imp
   Cont[UAF].tag = Ist_Store
   Container[UAF].Store.addr = CONT5
   Container[UAF].Store.data->tag = Iex_RdTmp
   Container[UAF].Store.data->Iex.RdTmp.tmp = Not_Imp

Rule1:
                CONT1 ≤ CONT3 < CONT1 + CONT2
Rule2:
                CONT1 ≤ CONT4 < CONT1 + CONT2
Rule3:
                CONT1 ≤ CONT5 < CONT1 + CONT2
```

Fig. 9: Specification of use-after-free vulnerability in VEX language

and $VulConst(n)$ to represent 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.

The vulnerability constraints are calculated according to the vulnerability specifications in the previous section. To make this step clearer, an example test unit and its corresponding constraint tree are presented in figure Fig. 10. In this figure, Node(2) and Node(4) represent basic blocks with probably vulnerable statements because they copy some data into the `msg` buffer. Thus, according to size of the `msg` buffer, a vulnerability constraint is calculated in addition to the path constraints for these nodes.

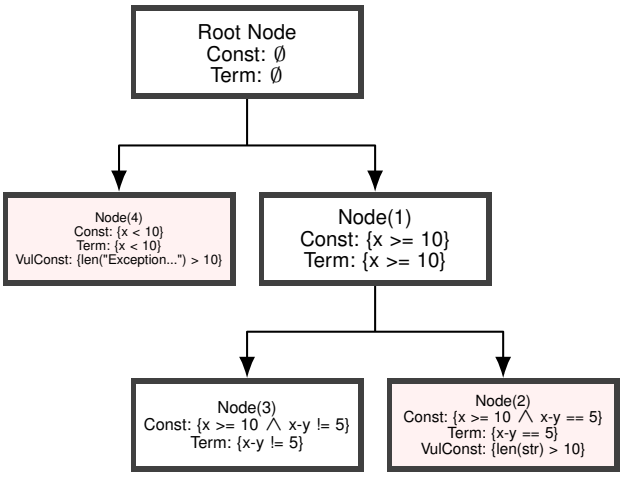
2.3 Learning and simulation process

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 the *Cover* algorithm presented in [11], and our modifications are shown in blue.

In this algorithm, first in lines 1 and 2, we perform n-factor Monte Carlo simulation on the program by picking values over a d-dimensional space for d input arguments. We generate possible combinations of input vectors and execute the program by giving them as system arguments. For each system input vector, we monitor the program execution and record the corresponding unit input vector to annotate nodes of the unit tree with pairs of the system and correlated unit input vectors (V_k, v_k) that reach these nodes.

Next, in lines 3 and 4, we explore the constraint tree and analyze only nodes located in a possibly vulnerable path. A vulnerable path is defined as a path from the root to a leaf in the constraint tree, which contains some nodes with probably vulnerable statements, e.g., a store operation to a stack buffer or a `free` function call. In contrast to the algorithm in [11], which processes all nodes of the constraint tree at this step, we restrict our analysis to fewer nodes according to the specification of the target vulnerability class to improve the efficiency of our method.

In lines 5 to 9, we check if these nodes and their siblings have been executed during the simulation. If yes, we use *TAR3* treatment learning algorithm [12] to estimate the range of system inputs that could explore the desired node in the unit. Otherwise, in lines 10 to 12, for each uncovered node whose path constraints are satisfiable, a function named *ComputeMap* is called that estimates the correlation between system and unit input data as a function f_n. The algorithm of this function is presented in Algorithm 2.



(a) Extracted unit tree

```
1  void testUnit(int x, int y, char * str)
2  {
3     char * msg = (char *)malloc(10);
4     if(x >= 10)
5     {
6        if (x-y == 5)
7        {
8           strcpy(msg, str);
9        }
10    }
11    else
12    {
13       strcpy(msg, "Exception...");
14    }
15    printf("%s", msg);
16 }
```

(b) Source code of a test unit

Fig. 10: An example of a test unit and its corresponding constraint tree

2.4 *Test data generation*

Until now, we have only considered path constraints in generating system input data. In lines 16 to 23 of Algorithm 1, for each node containing a possibly vulnerable statement whose path constraints are satisfiable, we attempt to generate system input data consistent with both path and its calculated vulnerability constraints. To do so, in lines 18 and 19, 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 reveal vulnerability in that node. In the last step, using the range of system inputs calculated by *TAR3* algorithm, we find relevant system input data for the intended unit input data.

Next, in lines 20 and 21, for each node that has not been covered in the simulation step, we use the fitted function f_n 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 consider 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 explore 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 line 17, to generate appropriate system input values that reach the intended nodes in the test unit and activate their vulnerabilities. In contrast, the algorithm in [11] only considers the path constraints for generating system inputs that cover the nodes of the unit.

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 $n\}$
7: $V'' \leftarrow V \backslash V'$
8: $(I_n, R_n, \_) \leftarrow$ RunTar3(I, V, V', V'')
9: $\forall j \in I_n$ store the range $r_j \in R_n$ for j
10: **else if** n is satisfiable but not covered **then**
11: $C \leftarrow Term(n)$
12: $(I_n, i_n, f_n) \leftarrow$ ComputeMap$(C, I, V, v, n, Parent(n))$
13: **end if**
14: **end if**
15: **end for**
16: **for** node n in T that n is possibly vulnerable and satisfiable **do**
17: $C_n \leftarrow Const(n) \wedge VulConst(n)$
18: **if** n is covered **then**
19: Generate input using C_n and $\forall j \in I_n$ use r_j from line 9
20: **else**
21: Generate input using C_n and f_n from line 12
22: **end if**
23: **end for**

2.4.1 ComputeMap algorithm: We have used the same algorithm as introduced in [11] for the *ComputeMap* function, which is presented in Algorithm 2. Here, we describe this algorithm to clarify the whole process for the reader. In summary, this algorithm attempts to define the correlation of system and unit input parameters as a function. 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 initially considers the constraints of each node individually. More precisely, instead of considering all parameters in $Vars(Const(n))$ as i_n, the algorithm first attempts to reduce the unit and system input parameters by selecting a subset of unit input parameters i_n appearing in $Vars(Term(n))$ and a subset of system input parameters I_n that are most effective on the values of parameters in i_n.

Thus, the unit input parameters related to the node constraints are extracted, and the first 20% of system vectors that are more compatible with this constraints subset are selected as a set V'. Afterwards, *TAR3* algorithm is applied to the sets $V' \subset V$ and $V'' = V \backslash V'$, and if a smooth relationship is established therebetween, the function f_n is built using curve fitting. Otherwise, the process is recursively repeated by adding parent node constraints to the given node in order to establish a smooth relationship. A smooth function is a function that has continuous derivatives up to a specific order.

Algorithm 2 ComputeMap(C, I, V, v, n, n')

Input: Constraints set C, System Inputs I, 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

Output: (I_n, i_n, f_n) where $i_n = Vars(C)$ and $I_n = f(i_n)$

1: $i_n = Vars(C)$
2: $V' \leftarrow \{a \in V |\ a$ is in 20% of points closet to $C\}$
3: $V'' \leftarrow V \backslash V'$
4: $(I_n, R, Smooth) \leftarrow$ 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$(C, I, V, v, n, Parent(n'))$
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**

If a smooth relationship is not found by including all terms in $Const(n)$, in lines 13 to 20, we walk up through the unit tree to find a parent node with enough system input vectors in its annotation. Such a node is covered in the simulation step with an appropriate number of input vectors (V_k, v_k) that helps to better estimate the function f_n using the curve fitting algorithm.

3 Evaluation

UbSym is implemented as a plugin for *angr* framework. In our implementation, `string` data type is also supported for detecting vulnerabilities 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 225 benchmark programs provided by the National Institute of Standards and Technology in Software Assurance Reference Dataset (SARD) project [1] is selected. This set is divided into four groups of vulnerable programs for evaluating the detection performance of four vulnerability classes. The test programs in this set contain a wide range of vulnerable instructions, which helps us to examine our method in various scenarios.

The test programs in NIST SARD are classified based on the classification of vulnerabilities in the Common Weakness Enumeration (CWE) database. We have tested our tool on the programs of classes CWE122_Heap_Based_Buffer_Overflow and CWE121_Stack_Based_Buffer_Overflow for heap-based and stack-based buffer overflow vulnerabilities. The vulnerability occurs in these programs when some constant data is copied into a heap or stack buffer using `strcpy`, `strcat`, `memcpy`, and `memmove` functions. 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 or stack buffer, we have copied an input variable, entered by the user as a command-line argument, into that buffer to create a vulnerability constraint in the test unit. A vulnerable function in one of these benchmark programs is presented in Fig. 11 as an example and our added `if` statement is underlined in line 16. The same `if` statement is similarly added to all benchmark programs.

We have also tested our tool on CWE416_Use_After_Free and CWE415_Double_Free benchmark programs to evaluate the performance of our method in detecting use-after-free and double-free vulnerabilities, respectively. The `if` statement mentioned above is also added to the vulnerable path in these programs. In CWE415_Double_Free programs, the `if` statement is placed on top of the second `free` function call and in CWE416_Use_After_Free programs, it is placed on top of the memory usage instruction.

In the second experiment, we have created a test program for each vulnerability class with several functions and more complicated path and vulnerability constraints to better evaluate the efficiency of our method. The source code of these programs, along with their details, are presented in the Appendix.

In both experiments, we have compared our implemented solution with two other tools that use similar methods for detecting various vulnerability classes in C programs, named MACKE [14] and Driller [17].

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 input data detecting more in-depth vulnerabilities. Our proposed method is compared with this tool as Driller 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 [17].

MACKE is a framework written on top of the KLEE symbolic execution engine [5] for analysis of C programs and detecting unhandled memory operations that result in memory out-of-bounds errors [14]. MACKE first recognizes each function through static analysis as a test unit. It also statically analyzes the call graph and the program's control flow graph to identify possible function call scenarios. This way, it determines the possibility of reaching a specific function containing vulnerable statements in a sequence of function calls. Then, it performs symbolic execution on each unit to generate appropriate unit input values that reveal vulnerabilities in that unit. Since MACKE does not consider the constraints of the path from the beginning of the program to the test unit, it might generate several false positives in this step. Therefore, in the last analysis step, it omits the detection records of unreachable functions from the previously analyzed function call chains.

3.1 Experiment 1

Table 1 represents the results of our first experiment in testing NIST SARD vulnerable programs. Each test program in this set has a function whose name includes the word `bad`, which contains a vulnerable statement, and one or multiple functions whose names include the word `good`, containing similar statements without vulnerability. Thus, we expect to achieve exactly one true positive alarm for each test program using a precise vulnerability detection mechanism. We have used three evaluation metrics in this experiment, i.e., precision, recall, and accuracy, as shown in (1), (2), and (3), respectively. In these equations, TP is the number of true positives, TN is the number of true negatives, FP is the number of false positives, and FN is the number of false negatives. Since *UbSym* detects vulnerabilities in the test programs of this experiment in less than 120 seconds, we have set the timeout value as 900 seconds. Thus, if a

```
1  void CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memcpy_34_bad(char * source)
2  {
3      char * data;
4      CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memcpy_34_unionType myUnion;
5      char dataBadBuffer[50];
6      char dataGoodBuffer[100];
7      /* FLAW: Set a pointer to a "small" buffer. This buffer will be used in the sinks as a destination * buffer in
        various memory copying functions using a "large" source buffer. */
8      data = dataBadBuffer;
9      data[0] = '\0'; /* null terminate */
10     myUnion.unionFirst = data;
11     {
12         char * data = myUnion.unionSecond;
13         {
14             /* POTENTIAL FLAW:
15             Possible buffer overflow if the size of data is less than the length of source */
16             if(source[0] == '7' && source[1] == '/' && source[2] == '4' && source[3] == '2'
                && source[4] == 'a' && source[5] == '8' && source[75] == 'a')
17             {
18                 memcpy(data, source, strlen(source)*sizeof(char));
19             }
20             data[100-1] = '\0'; /* Ensure the destination buffer is null terminated */
21             printLine(data);
22         }
23     }
24 }
```

Fig. 11: A sample vulnerable unit in benchmark programs

Table 1 Results of evaluating the approaches on the selected group of test programs in NIST benchmark programs

	Heap_Based_Buffer_Overflow			Stack_Based_Buffer_Overflow			Use_After_Free			Double_Free		
Method	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
MACKE	0.78	0.87	0.60	0.81	0.75	0.84	0.21	1.00	0.21	0.82	1.00	0.82
Driller	1.00	1.00	1.00	0.90	1.00	0.77	-	-	-	0.85	1.00	0.85
UbSym	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 2 Results of evaluating the approaches on the complicated test programs

	Heap_Based_Buffer_Overflow					Stack_Based_Buffer_Overflow					Use_After_Free					Double_Free				
Method	TP	FP	TN	FN	Time,s	TP	FP	TN	FN	Time,s	TP	FP	TN	FN	Time,s	TP	FP	TN	FN	Time,s
MACKE	3	0	0	3	290	3	0	0	3	193	1	0	0	3	141	0	0	0	4	140
Driller	4	0	0	2	3325	4	0	0	2	3433	-	-	-	-	-	3	0	0	1	3425
UbSym	6	0	0	0	517	6	0	0	0	1340	4	0	0	0	110	4	0	0	0	88

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

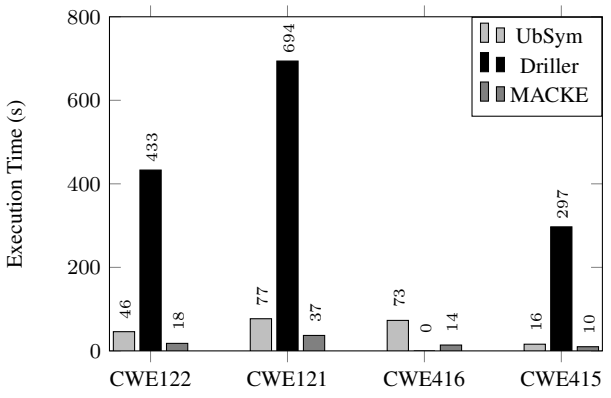


Fig. 12: The average analysis time of the tools for a group of test programs in NIST benchmark programs

tool does not detect a vulnerability in a test program in less than 900 seconds, we consider it a false negative.

Given the experimental results in Table 1, *UbSym* has detected all vulnerabilities of NIST SARD test programs with no false alarms. Considering the average execution time of analyzing each group of test programs, as shown in Fig. 12, *UbSym* was considerably faster than Driller. 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 out of the unit. On the contrary, our proposed method generates accurate test data values for running the whole program from the beginning and reaching the vulnerable statement in the test unit. This is a reason that causes our method to take more time to test and analyze a program.

It is worth mentioning that Driller only detects vulnerabilities making the program crash. Therefore, it has not been tested on CWE416_Use_After_Free programs in our experiments as it could not detect vulnerability in these programs.

3.2 Experiment 2

In the second experiment, we have designed four test programs that contain more vulnerable instructions and more complicated constraints in comparison with the test programs in the first experiment. The structures of these programs are similar, but they have different vulnerable statements. There are six vulnerability occurrences through three functions in programs that contain stack-based

or heap-based buffer overflow and four vulnerability occurrences through two functions in programs that contain use-after-free or double-free vulnerability. Also, there are various path constraints inside and outside of each test unit. Thus, *UbSym* is supposed to identify test units and analyze them to generate twenty true positive alarms in the test programs altogether. The source code of the test programs in this experiment, along with its details, is presented in the Appendix.

The results of this experiment are demonstrated in Table 2. In this experiment, we have set the timeout value for analyzing each program as 1 hour. As shown in Table 2, MACKE could detect only seven vulnerability occurrences in the given programs as it could not analyze complicated path conditions. As well, Driller could only detect eleven vulnerability occurrences since it takes much time to analyze all execution paths. On the contrary, *UbSym* has been able to precisely detect all vulnerabilities and it has generated appropriate test data for the whole program, which enter the programs from the beginning and reveal the vulnerability in the test units. Besides, the testing time for *UbSym* has been much less than that of Driller.

The results of these two experiments demonstrate the advantage of restricting symbolic execution scope and applying machine learning techniques in estimating the program's behavior for detecting various vulnerability classes.

4 Conclusion

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 states grows exponentially, and thorough analysis of real-world programs becomes infeasible.

We proposed a method for applying symbolic execution to detecting four classes of memory corruption vulnerabilities in executable codes, i.e., heap-based buffer overflow, stack-based buffer overflow, use-after-free, and double-free. We limit the scope of symbolic execution to the test units to avoid path explosion. We specified intended memory corruption vulnerability classes in executable codes and presented a method for automatically determining the test units in arbitrary programs accordingly. Using symbolic execution, we generate appropriate unit input data for detecting memory corruption vulnerabilities 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 enter into the program from the beginning, cause the execution of vulnerable statements in the test unit, and reveal their vulnerabilities. The experiments showed that this method achieves more efficient and accurate results in detecting vulnerabilities in complex programs compared to similar tools.

5 Appendix

The base structure of the designed complex programs is a simple authentication code by which users carry out sign-up and sign-in operations. The source code of these programs is presented in Fig 13. To generate the test program of each vulnerability class, the commented lines for each specific vulnerability should be uncommented. In the following, the structure of the test program that contains heap-based buffer overflow vulnerability is explained as an instance.

This program begins by receiving a username and password in the console to sign-up a user. If the condition in line 99 is satisfied, the vulnerable function `signup` would be called. In this function, two heap buffers are allocated in lines 6 and 7. As there are two copy operations with `memcpy` function calls in lines 14 and 17, our solution identifies this function as a test unit. There is a path constraint in this function in line 10; therefore, if the input strings for username and password satisfy the path constraints in lines 99 and 10, and their lengths are more than the lengths of the destination heap buffers in the copy operations, they would cause heap-based buffer overflow. Note that the path constraint in line 99 is out of the test unit and should be determined through machine learning. *UbSym* calculates the path constraints in line 10 using symbolic execution. It generates appropriate input data for the `scanf` operations in line 95, which are consistent with both path constraints inside and outside the unit.

There are two other test units in this program, `check` and `authentication` functions, which cause heap-based buffer 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 it. *UbSym* could successfully identify these units and generate appropriate test data for the whole program, which explore vulnerable instructions in the unit and cause heap-based buffer overflow in them.

6 References

1 National Institute of Standards and Technology in Software Assurance Reference Dataset Project, `https://samate.nist.gov/SRD`, last accessed 4 March 2022

2 UbSym, `https://github.com/SoftwareSecurityLab/UbSym`

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) (may 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. p. 209–224. OSDI'08, USENIX Association, USA (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 Transactions on Software Engineering pp. 1–1 (2021). https://doi.org/10.1109/TSE.2021.3101870

7 Cha, S., Lee, S., Oh, H.: Template-Guided Concolic Testing via Online Learning, p. 408–418. Association for Computing Machinery, New York, NY, USA (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. p. 235–245. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA (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, Germany (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 (sep 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: Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments. p. 294–309. VSTTE'12, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_23

12 Menzies, T., Hu, Y.: Data mining for very busy people. Computer **36**(11), 22–29 (2003). https://doi.org/10.1109/MC.2003.1244531

13 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

14 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. p. 780–785. ASE 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2970276.2970281

15 Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing Nasa Software. p. 15–26. ISSTA '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1390630.1390635

16 Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 138–157 (2016). https://doi.org/10.1109/SP.2016.17

17 Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Krügel, C., Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: NDSS (2016). https://doi.org/10.14722/ndss.2016.23368

18 Strang, G.: Linear algebra and its applications. Thomson, Brooks/Cole, Belmont, CA (2006)

```c
#define def_user "u-admin-"
#define def_pass "password"
void signup(char *username, char *password)
{
   /* Uncomment the following two lines for heap-based buffer overflow & use-after-free & double-free */
   // char* tmp_user = (char *)(malloc(50*sizeof(char)));
   // char* tmp_pass = (char *)(malloc(50*sizeof(char)));
   /* Uncomment the following line for stack-based buffer overflow */
   // char tmp_user[16], tmp_pass[16];
   if((username[1] >= '0' && username[1] <= '9') && !strncmp(password, "passW0rd", 8))
   {
      /* POTENTIAL FLAW: data may not have enough space to hold source */
      /* Uncomment the following line for heap-based & stack-based buffer overflow */
      // memcpy(tmp_user, username, strlen(username));
      /* POTENTIAL FLAW: data may not have enough space to hold source */
      /* Uncomment the following line for heap-based & stack-based buffer overflow */
      // 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);
      /* POTENTIAL FLAW: Free data here - line 30 frees data as well */
      /* Uncomment the following line for double-free */
      // free(tmp_user); free(tmp_pass);
   }
   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"); }
   /* Uncomment the following line for double-free & use-after-free & heap-based buffer overflow */
   // free(tmp_user);   free(tmp_pass);
   /* POTENTIAL FLAW: Use of data that may have been freed in line 30 */
   /* Uncomment the following line for use-after-free */
   // tmp_user[50-1] = '\0'; tmp_pass[50-1] = '\0';
}
bool check(char *username, char *password)
{
   /* Uncomment the following two lines for heap-based buffer overflow & use-after-free & double-free */
   // char* tmp_user = (char *)(malloc(50*sizeof(char)));
   // char* tmp_pass = (char *)(malloc(50*sizeof(char)));
   /* Uncomment the following line for stack-based buffer overflow */
   // char tmp_user[16], tmp_pass[16];
   if((username[0] >= 'A' && username[0] <= 'Z') && (username[1] >= '0' && username[1] <= '9'))
   {
      /* POTENTIAL FLAW: data may not have enough space to hold source */
      /* Uncomment the following line for heap-based & stack-based buffer overflow */
      // strcpy(tmp_user, username);
      /* POTENTIAL FLAW: data may not have enough space to hold source */
      /* Uncomment the following line for heap-based & stack-based buffer overflow */
      // strcpy(tmp_pass, password);
      if(!strcmp(tmp_user, def_user) && !strcmp(tmp_pass, def_pass)) {return true;}
      char passwd[50]; int fd = open(tmp_user, O_RDONLY);
      if(fd < 0) { printf("An unexpected problem occurred!\n"); return false; }
      read(fd, passwd, sizeof(passwd));
      if(!strcmp(passwd, tmp_pass)) { return true; }
   } return false;
}
bool signin(char *username, char *password)
{
   if(check(username, password))
   {
      printf("Hey %s, you logged in successfully\n", username);
      /* Uncomment the following line for double-free & use-after-free) */
      // free(username); free(password);
      return true;
   } else { printf("The username or password is wrong\n"); return false; }
}
void authentication(char *username, char *password)
{
   /* Uncomment the following two lines for heap-based buffer overflow & use-after-free & double-free */
   // char* tmp_user = (char *)(malloc(80*sizeof(char)));
   // char* tmp_pass = (char *)(malloc(80*sizeof(char)));
   /* Uncomment the following line for stack-based buffer overflow */
   // char tmp_user[32], tmp_pass[32];
   /* POTENTIAL FLAW: data may not have enough space to hold source */
   /* Uncomment the following line for heap-based & stack-based buffer overflow */
   // memcpy(tmp_user, username, strlen(username));
   /* POTENTIAL FLAW: data may not have enough space to hold source */
   /* Uncomment the following line for heap-based & stack-based buffer overflow */
   // memcpy(tmp_pass, password, strlen(password));
   signin(tmp_user, tmp_pass);
   /* POTENTIAL FLAW: Use of data that may have been freed in line 63 */
   /* Uncomment the following line for use-after-free */
   // tmp_user[80-1] = '\0'; tmp_pass[80-1] = '\0';
   /* POTENTIAL FLAW: Free data here - line 63 frees data as well */
   /* Uncomment the following line for double-free */
   // free(tmp_user);   free(tmp_pass);
}
int main (int argc, char *argv[])
{
   /* uncomment for stack-based buffer overflow */
   // char username[64]; char password[64];
   /* Uncomment the following two lines for heap-based buffer overflow & use-after-free & double-free */
   // char *username = (char *)(malloc(100*(sizeof(char))));
   // char *password = (char *)(malloc(100*(sizeof(char))));
   printf("Enter username :"); scanf("%s", username); printf("Enter password :"); scanf("%s", password);
   if(argc >= 3) { authentication(argv[1], argv[2]); }
   else
   {
      if(username[0] >= 'A' && username[0] <= 'Z') { signup(username, password); }
      else { printf("The selected username is not valid, it must start with an uppercase letter"); }
   }
}
```

Fig. 13: Source codes of the four designed complex programs