IET Software

Research Article

IET Journals
The Institution of Engineering and Technology

Smart fuzzing method for detecting stack-based buffer overflow in binary codes

ISSN 1751-8806
Received on 19th March 2015
Revised on 13th December 2015
Accepted on 19th January 2016
doi: 10.1049/iet-sen.2015.0039
www.ietdl.org

Maryam Mouzarani[1], Babak Sadeghiyan[1] ✉, Mohammad Zolfaghari[2]

[1]*Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran*
[2]*Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran*
✉ *E-mail: basadegh@aut.ac.ir*

Abstract: During the past decades several methods have been proposed to detect the stack-based buffer overflow vulnerability, though it is still a serious threat to the computer systems. Among the suggested methods, various fuzzers have been proposed to detect this vulnerability. However, many of them are not smart enough to have high code-coverage and detect vulnerabilities in feasible execution paths of the program. The authors present a new smart fuzzing method for detecting stack-based buffer overflows in binary codes. In the proposed method, concolic (concrete + symbolic) execution is used to calculate the path and vulnerability constraints for each execution path in the program. The vulnerability constraints determine which parts of input data and to what length should be extended to cause buffer overflow in an execution path. Based on the calculated constraints, the authors generate test data that detect buffer overflows in feasible execution paths of the program. The authors have implemented the proposed method as a plug-in for Valgrind and tested it on three groups of benchmark programs. The results demonstrate that the calculated vulnerability constraints are accurate and the fuzzer is able to detect the vulnerabilities in these programs. The authors have also compared the implemented fuzzer with three other fuzzers and demonstrated how calculating the path and vulnerability constraints in the method helps to fuzz a program more efficiently.

1 Introduction

Buffer overflow is a well-known software vulnerability. During the past decades, various methods have been suggested to detect this vulnerability. However, buffer overflow attacks are still considered as a serious threat to the computer systems [1]. One of the effective methods suggested for detecting different classes of vulnerabilities is smart fuzzing [2, 3]. Some smart fuzzers use concolic (concrete + symbolic) execution to analyse the target program, e.g. [4–7]. These fuzzers execute the program with concrete input data and calculate symbolic path constraints for the executed path. The path constraints define the characteristics of input data that makes the current path be executed. These constraints are used to generate new test data that traverse other execution paths of the program. Moreover, for each executed path, vulnerability constraints are calculated symbolically. The vulnerability constraints determine the characteristics of input data that activates a specific vulnerability in the executed path. By the help of vulnerability constraints, new test data are generated that activate vulnerabilities in that execution path. Concolic execution has two advantages, i.e. high code-coverage and low false positives (FP), which motivate smart fuzzers to use this method [8]. However, as far as we know there is no smart fuzzer that uses this method to detect buffer overflows in the binary codes.

In this paper, we present a concolic execution-based smart fuzzing method for detecting buffer overflows in binary codes. Since the proposed method analyses the binary code to generate new test data, it is useful even when the source code is not available. In our method, the path and buffer overflow constraints are calculated symbolically for each execution path. As we calculate vulnerability constraints for the variables that are stored in the stack memory, our method is able to detect stack-based buffer overflows and does not cover heap-based buffer overflows.

The generated vulnerability constraints determine to what length the input data should be extended to overflow a stack buffer. Unlike defining variables in the source codes, the variables and their lengths are not clearly defined in the binary codes. However, by considering the structure of the stack, its management and the general method of accessing local variables in the binary codes, we are able to estimate the length and addresses of variables in the stack. Thus, we first present a method for estimating the lengths and addresses of static local variables in each function. Based on the estimated lengths and addresses, buffer overflow constraints are calculated.

We use taint analysis to consider the instructions that are affected by input data in calculating the constraints. Hence, the vulnerability constraints take into account which parts of input data might be involved in overflowing a stack buffer. This helps to fuzz a program more intelligently. For example, if a specific field in a configuration file is used in a *strcpy()* function, our method focuses on this field and does not extend the length of all the fields in that file. Moreover, since the loops and well-known copy functions are more likely to be vulnerable [9], we process the calculated constraints of loops and well-known vulnerable functions with more priority.

Accordingly, our contributions in this paper are as follows:

- Presenting a concolic execution-based smart fuzzing method for detecting stack-based buffer overflows in the binary codes. In this method, the vulnerability constraints in each path are calculated by analysing the binary code and are combined with the path constraints in order to generate appropriate test data.
- Presenting a method for estimating addresses and lengths of static local variables in the binary codes. The estimation is based on the methods of accessing local variables and function arguments in the stack in the well-known assembly languages. The results of this estimation are used to calculate the stack overflow constraints.

The remaining of this paper is organised as following: Section 2 presents background information and related works. The proposed smart fuzzing method is described in Section 3. Section 4 describes the implementation details and evaluates the implemented fuzzer. The paper is concluded in Section 5.

IET Softw., 2016, Vol. 10, Iss. 4, pp. 96–107

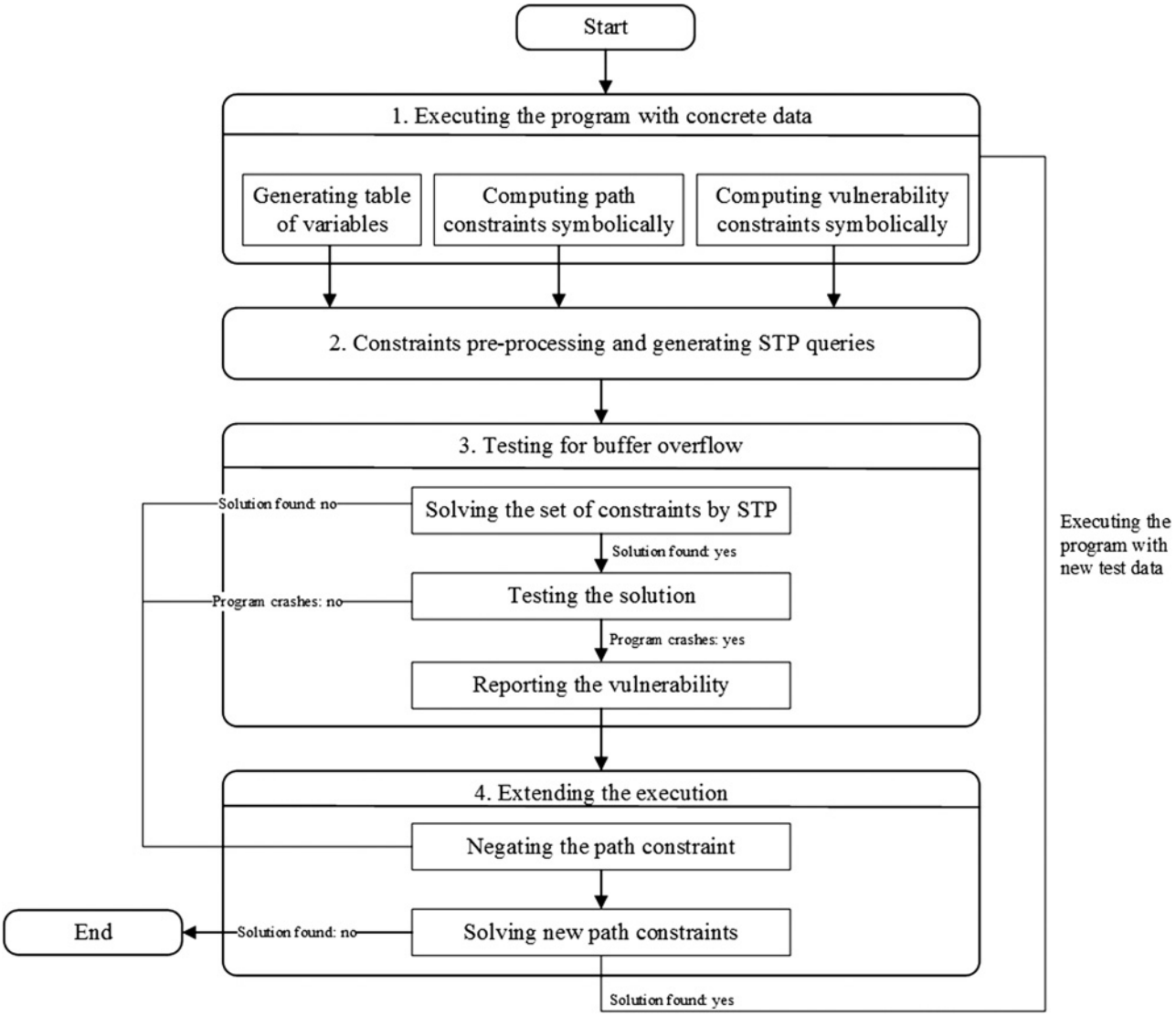

Fig. 1 *Steps of our proposed smart fuzzing method*

2 Background and related works

During the past years, various protection mechanisms have been proposed to prevent successful exploitation of buffer overflow vulnerabilities, such as stack cookies [10], data execution prevention [11] and address space layout randomisation [12]. However, many of these methods impose high overhead on the program execution and are not usable for large programs [1]. Moreover, various attack vectors are able to bypass these protection mechanisms and successfully exploit buffer overflows in the applications [1, 13]. For example, return oriented programming [14], information leakage [15] and the possibility of using user scripting or just-in-time compilation [16] allow the attackers to bypass these protection methods. Therefore, detection and removal of buffer overflow vulnerabilities is still required.

Fuzzing or fuzz testing is a software testing method that has been used for detecting software vulnerabilities for many years. In this method, the program is executed with numerous random data in order to analyse its different behaviours. The goal is to traverse as many execution paths as possible and analyse possible behaviours of the program. Smart fuzzing is an effective fuzzing method that performs an analysis on the target software to gather more information about it [2]. Based on this information, smart fuzzers generate new test data that traverse deeper paths in the program and increase the chance of detecting vulnerabilities.

Some smart fuzzers use concolic (symbolic + concrete) execution method to achieve higher code-coverage, e.g. [6, 7, 17, 18]. In this method, the program is first executed with some concrete input data. Then, the executed path is analysed and its constraints are calculated symbolically. The calculated path constraints are negated one by one, from the last to the first, and solved by a constraint solver, such as STP [19] or Z3 [20]. If the constraints are solved, some input data are generated based on the solution that traverse new paths in the program.

Some concolic execution-based smart fuzzers, i.e. EXE [4] and KLEE [5], calculate vulnerability constraints in addition to the path constraints to detect buffer overflow vulnerabilities. These fuzzers first analyse the program source code to identify the arrays and their lengths, and use this information to calculate the vulnerability constraints. When the program is executed with concrete data, these fuzzers calculate the path and vulnerability constraints for the executed path. For the instructions that access a pointer, they generate a vulnerability constraint to check whether the pointer is able to point out of its legitimate range. After the program execution, the vulnerability constraints are combined with the path constraints and queried from a constraint solver. If the constraint solver returns a solution for the set of constraints, new concrete input data is generated to detect vulnerabilities in the executed path.

Calculating the vulnerability constraints and combining them with the path constraints result in more intelligent fuzzing. In this method,

the test data are generated purposefully to execute a specific path and activate a specific vulnerability in that path. These data can be used as a proof for existence of the vulnerability in the program. However, as EXE and KLEE analyse the source code to compute the length of variables and generate the vulnerability constraints, they are not helpful when the source code is not available.

In recent years, a research trend is designing smart fuzzers that analyse the binary codes, instead of the source codes. Some reasons that make analysing binary codes more preferable are: reflection of the exact behaviour of the program, optimisations and bugs in the compilers, unavailability of the source codes and platform-specific details [21]. A well-known recent concolic execution-based smart fuzzer that detects buffer overflow in the binary codes is Dowser [9]. This fuzzer first analyses the binary code statically to locate the loops in the program. The idea is that in complex structures like loops, it is more possible that the programmer makes mistakes and vulnerabilities appear. Therefore, Dowser focuses on fuzzing the paths with more complex loops. This fuzzer uses concolic execution to generate test data that traverse new execution paths of the program. The difference between Dowser and other concolic execution-based fuzzers is that it explores the paths with complex loops with more priority. However, it does not calculate the vulnerability constraints to detect buffer overflow in the executed paths. In fact, Dowser only calculates the path constraints to generate test data that traverse different execution paths of the target program.

Concolic execution-based fuzzing has also been used to detect integer vulnerabilities [6, 7], division by zero and null pointer dereference [17] in binary codes. These fuzzers calculate the vulnerability constraints in addition to the path constraints to detect the specified vulnerabilities. However, there is no concolic execution-based smart fuzzer that detects stack-based vulnerabilities in the binary codes by calculating the path and vulnerability constraints.

We propose a smart fuzzing method that profits the advantages of concolic execution and detects stack-based buffer overflows in the binary codes. It instruments the binary program to calculate the path and vulnerability constraints and also estimate length of the stack buffers in the program. Based on the estimated lengths, our fuzzer calculates the vulnerability constraints for possible vulnerable statements in an execution path. Thus, the proposed fuzzer does not require the source code of the program to generate vulnerability constraints. Our fuzzer has three advantages that help it detect the vulnerabilities more efficiently. First, it generates test data by considering those parts of input data that affect on possible vulnerable instructions in the program. Moreover, it estimates the length of the stack buffer and extends the length of input data accordingly to overflow the buffer. Thus, it does not change the length of all parts of input data blindly to detect the vulnerabilities in the program. Second, it considers the path and vulnerability constraints when it generates new test data. Therefore, the test data traverse a specific execution path and reach the intended possible vulnerable instructions in that path. Third, our fuzzer processes the buffer overflow constraints of the loops and well-known vulnerable functions, e.g. *strcpy()*, with more priority. Since the loops and well-known copy functions are more probable to be vulnerable [9, 22], the proposed fuzzing method first fuzzes these instructions in an execution path to be more efficient.

3 New smart fuzzing method for detecting stack-based buffer overflows

Our proposed method consists of four main steps, which are illustrated in Fig. 1. In our method, the program is first executed with concrete input data. During the execution, the path and vulnerability constraints are calculated symbolically. Moreover, a table is generated and maintained in this step that records the size and address of static variables. In the second step, the vulnerability constraints are processed and combined with the calculated path

```
MySub PROC
    push  ebp            ; save base pointer
    mov   ebp,esp        ; base of stack frame
    push  ecx
    push  edx            ; save EDX
    mov   eax,[ebp+8]    ; get the stack parameter
    .
    .
    pop   edx            ; restore saved registers
    pop   ecx
    pop   ebp            ; restore base pointer
    ret                  ; clean up the stack
MySub ENDP
```

Fig. 2 *Sample ×86 assembly procedure [23]*

constraints. The combined constraints are queried from STP constraint solver in the third step. If STP returns a solution, it is used to generate new test data to detect vulnerabilities in the executed path. In the last step, the path constraints are negated one by one from the last to the first. After each negation, the path constraints are queried from STP constraint solver to generate test data that traverse new execution paths in the program. These data are used as the concrete input data to restart the fuzzing process from the first step. This process continues until the fuzzer does not generate new test data at the fourth step. The details of each step are presented in the following sections.

3.1 Executing the program with concrete data

3.1.1 Generating table of variables: To calculate the vulnerability constraints, the size and address of static variables are calculated and stored in a table during the program execution. Although the variables are not explicitly defined at the binary level, the general method of accessing local variables in the stack frame can be used to estimate the addresses and lengths of variables. The method of accessing the local variables and function parameters in ×86 assembly is described by Irvine and Das in [23]. They define stack frame as the area of the stack for storing the function arguments, subroutines return addresses and saved registers. The stack frame is created in the following steps:

- First, the passed arguments are pushed on the stack.
- When a subroutine is called, its return address is pushed on the stack.
- The *ebp* is set equal to *esp* and used as the based reference for accessing all the subroutine parameters.
- If there are local variables, *esp* is decremented to reserve space for the variables on the stack.
- If any registers need to be saved, they are pushed on the stack.

The *ebp* register is used as the base address for accessing the stack parameters and local variables. For example, consider the procedure that is shown in Fig. 2. This procedure has one stack parameter. After initialising the stack, the contents of *ebp* remain fixed through the subroutine. Fig. 3 illustrates the stack frame for this procedure. Local variables are also created on the runtime stack and are initialised at runtime. These variables are usually located below the base pointer (*ebp*). As an example, the following code declares the local variables *X* and *Y* (see Fig. 4):

Using C++ compilers, the equivalent ×86 assembly of the above code is generated as illustrated in Fig. 5. This code shows how the local variables are created, assigned values and removed from the stack. Since the stack entries default to 32 bits, the size of each variable is rounded upward to a multiply of 4 in the stack [23]. Thus, in this code, 8 bytes are reserved for the two variables. Fig. 6 illustrates the stack frame after creating the local variables in the above code.

We utilise the fact that *ebp* is mostly used as the base address for accessing static variables and estimate the addresses and lengths of

IET Softw., 2016, Vol. 10, Iss. 4, pp. 96–107

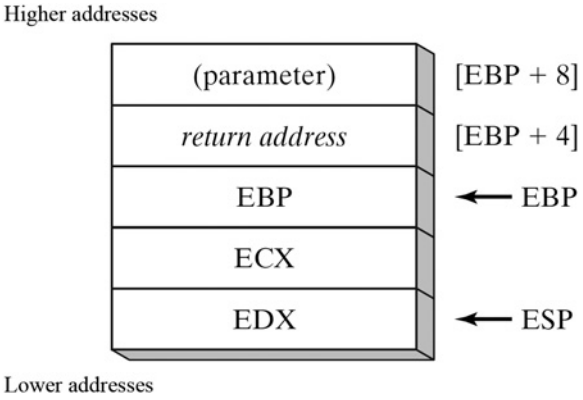

Fig. 3 *Stack frame for the procedure in Fig. 2 [23]*

```
void MySub()
{
int X = 10;
int Y= 20;
}
```

Fig. 4 *Code of the local variables X and Y*

static variables in the program. Therefore, in the proposed method the instructions that use *ebp* to access the stack are analysed for this estimation. In other words, in the instructions like $ebp - x$, $x \leq 0$, or $x \geq 0$, the values of $ebp - x$ and x are considered as the address and maximum length of a variable in the stack, respectively. The maximum length determines the distance between *ebp* and the address of the variable. For example, in the sample code of Fig. 5, two variables are located in addresses *ebp-0x4* and *ebp-0x8* with maximum lengths of 4 and 8, respectively. If a variable is assigned with data longer than its maximum length, it may overwrite *ebp* and cause a crash. Moreover, if the assigned data is longer than *maximum_length+4*, it may overwrite the return address in addition to *ebp*.

The lengths of variables can be computed according to their distance with other local variables in the same function. For example, in the previous sample C code, the length of *Y* can be computed by subtracting the address of *X* from the address of *Y*. To avoid implementation complexity, our method considers the maximum length and detects buffer overflows that overwrite *ebp* and return address values. In fact, it does not detect buffer overflows that only cause overwriting the local variables.

While the program is executed with concrete input data, a table is created for storing the estimated addresses and lengths of static local variables. This table is shown in Fig. 7, which is filled with the result of analysing the sample code of Fig. 5. For each instruction that

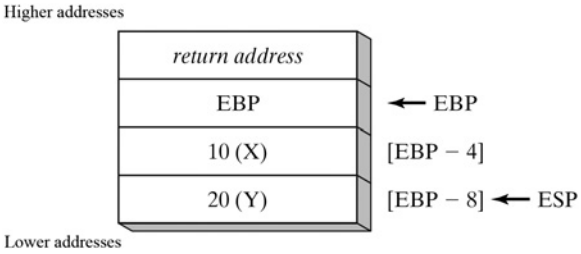

Fig. 6 *Stack content of the procedure of Fig. 5 [23]*

ID	Address	Max length	Variable function
1	ebp – 0x4	4	MySub
2	ebp – 0x8	8	MySub

Fig. 7 *Sample table of variables*

refers to the stack using *ebp* register, a new record is added to this table. This record defines the address and maximum length of a static local variable. Moreover, using the symbol table of the binary code, the name of the currently executed function is extracted and considered as the function that has defined the static variable. The extracted function name is stored as *variable function* in the new record. The table of variables is filled considering two points:

- The records must be sorted in descending order based on the address field.
- Since the stack is filled dynamically during the program execution, different functions may use the same area in the stack. In other words, when a function returns, it releases the assigned area in the stack. This area may be used by the following called functions. As the function abstraction is not presented at the binary level and it might be separated in non-contiguous parts [24], it is not possible to determine the end of a function and keep the table of variables very up-to-date. In other words, it is not possible to certainly remove the records of variables for the functions that would no longer be used. Instead, we can remove the records that overlap with a to-be-added new record and belong to other functions. Such overlaps mean that a new function is using a region of the stack that had been used by the previous functions and is now released. Thus, before adding a new record, it is checked if the new record overlaps with the previous records of other functions. If such records exist in the table, they are first removed and the new record is added then.

3.1.2 Computing the path and vulnerability constraints:

While the program is executed with concrete data, the path and vulnerability constraints are calculated for the executed path. The path constraints are computed with the same method as in EXE and KLEE for the branches in the executed path [4, 5]. Since buffer overflow might occur when the data are stored in the stack buffer, our fuzzer instruments the store instructions to calculate the vulnerability constraints. In fact, for each *STORE(Addr)=DATA*

```
MySub PROC
    push  ebp
    mov   ebp,esp
    sub   esp,8                     ; create locals
    mov   DWORD PTR [ebp-4],10      ; X
    mov   DWORD PTR [ebp-8],20      ; Y
    mov   esp,ebp                   ; remove locals from stack
    pop   ebp
    ret
MySub ENDP
```

Fig. 5 *×86 assembly equivalent of the sample code [23]*

C Code	Equivalent assembly (loop secction)	Generated constraints

`void test(char *source){`
`  int i;`
`  char dest[4];`
`  for(i=0;i<strlen(source);i++){`
`  dest[i]=source[i];`
`  }`
`  printf("%s",dest);`
`  return;`
`}`
	1. mov -0xc(%ebp),%eax
	2.mov -0xc(%ebp),%edx
	3. add 0x8(%ebp),%edx
	4. movzbl (%edx),%edx
	5.mov %dl,-0x10(%ebp,%eax,1)
	6. addl $0x1,-0xc(%ebp)
	7. mov -0xc(%ebp),%ebx
	8. mov 0x8(%ebp),%eax
	9. mov %eax,(%esp)
	10.call 8048448 <strlen@plt>
	11. cmp %eax,%ebx
		1. Input(0) is stored to a variable of length 12 that is defined in function *test* and used in function *test*
		2. Input(1) is stored to a variable of length 12 that is defined in function *test* and used in function *test*
		3. Input(2) is stored to a variable of length 12 that is defined in function *test* and used in function *test*

Fig. 8 *Creating vulnerability constraints for a sample code*

instruction, it searches for *Addr* in the table of the variables and selects the first record that includes *Addr* in its range, i.e. *address* < *Addr* < *address*+*maxlen*. To be more efficient, the vulnerability constraints are only generated for the instructions that store tainted *DATA* values in the stack. To sum up, a buffer overflow constraint is generated when some tainted data are stored in an address that exists in the table of variables. In a vulnerability constraint, the maximum length (L), the variable function (Var_func), the index of stored tainted bytes (B) and the name of current function (Cur_func) are specified as follows:

`Input(B) is stored to a variable of length L that is defined in function Var_func and used in function Cur_func.`

The name of the current function is specified as the function that might be buffer overflow vulnerable. As an example, Fig. 8 shows the vulnerability constraints that are generated for a sample code which copies a three-character tainted string into a static variable. The equivalent ×86 assembly of this code is presented in the second column. For each store instruction, a new primary vulnerability constraint is generated. The loop is repeated until the loop counter, *i*, becomes greater than the length of *source* string. Therefore, the store instruction is executed three times and three primary vulnerability constraints are generated. These constraints are presented in the constraint column of this table.

3.2 Processing the constraints

In the second step, the calculated path and vulnerability constraints are combined to generate new queries that are consistent with the syntax of the constraint solver queries. At this step, the vulnerability constraints are first pre-processed to detect the loops and well-known vulnerable functions in the executed path. The vulnerability constraints of such statements usually appear by a list of constraints with incrementing index of tainted input bytes and the same variable and Cur_func name, like the example in Fig. 8. Therefore, sequential constraints that are generated for the same destination and in the same vulnerable function are merged into a single buffer overflow constraint for the range of copied input bytes in that function.

For example, Table 1 illustrates some vulnerability constraints before and after the pre-processing step. In this table, the first four constraints are generated for copying consequent input bytes, from x to $x+3$, to the same variable in the same Cur_func. Thus, they are merged into a single constraint for storing bytes $[x, x+3]$ into the same variable in the same Cur_func. The fuzzer processes these constraints before the other vulnerability constraints that are generated for the executed path.

The vulnerability constraints are combined with the path constraints to generate queries for the constraint solver. The main concern in stack-based buffer overflow vulnerability is the length of stored data. Thus, the queries should be generated in a way to ask for input data that is longer than the estimated length of the intended destination buffer. In this way, the generated input data may cause buffer overflows in intended statements of a specific execution path. The combination of path and vulnerability constraints is performed by padding the path constraints in order to extend the length of the generated input data.

For example, when the input bytes $(x, x+5)$ are stored in a stack buffer with the length of *maxlen*, we make a padding in the path constraints for the input bytes in the range $(x, x+5)$ to extend the length of stored data. In fact, we make the length of the bytes in $(x, x+5)$ more than *maxlen* +4 to overwrite the return address. To do so, the path constraints for bytes $(0, x+4)$ remain unchanged and the rest of the constraints are shifted from byte i to the byte $i+maxlen+4$. Fig. 9 illustrates such padding process. Padding the specific bytes that are involved in possible vulnerable statements helps to have more guided fuzzing. Instead of extending the length of all the strings and fields in the input data blindly, only the sections that are used in vulnerable statements becomes longer.

Table 1 Processing the vulnerability constraints

Constraints before pre-processing	Constraints after pre-processing
Input(x) is stored to a variable of length 12 that is defined in function test and used in function test	Input([x, $x+3$]) is stored to a variable of length 12 that is defined in function test and used in function test
Input($x+1$) is stored to a variable of length 12 that is defined in function test and used in function test	
Input($x+2$) is stored to a variable of length 12 that is defined in function test and used in function test	
Input($x+3$)is stored to a variable of length 12 that is defined in function test and used in function test	
Input($x+10$) is stored to a variable of length 12 that is defined in function test and used in function test	Input([$x+10$, $x+11$]) is stored to a variable of length 12 that is defined in function test and used in function test
Input($x+11$) is stored to a variable of length 12 that is defined in function test and used in function test	
Input($x+12$) is stored to a variable of length 8 that is defined in function test and used in function test2	Input($x+12$) is stored to a variable of length 8 that is defined in function test and used in function test2

IET Softw., 2016, Vol. 10, Iss. 4, pp. 96–107

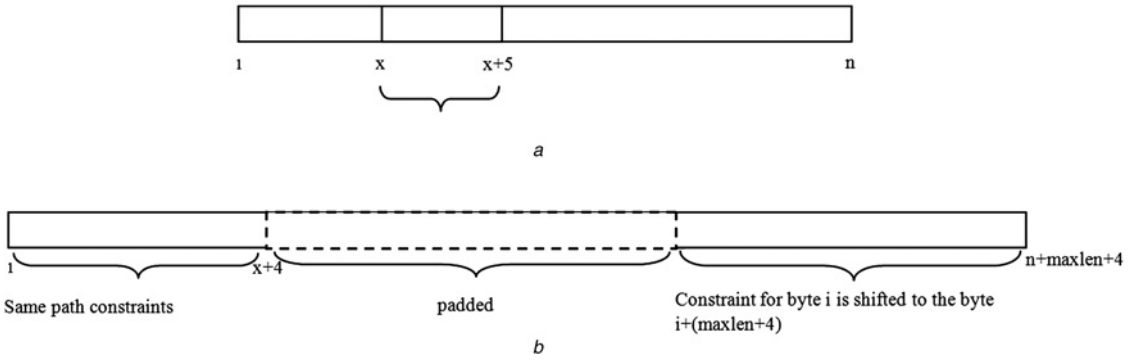

Fig. 9 *Padding of the constraints*

a Illustrates the constraints before padding
b Illustrates the constraints after padding

This makes the fuzzing process more efficient. Moreover, the new input satisfies the path constraints hence it will reach the intended possible vulnerable statements.

3.3 Testing for buffer overflow

In this step, the combined constraints are queried from STP constraint solver. If STP returns a solution, it is used to generate new test data that cause buffer overflow in the executed path. The program is then executed with the generated input data. If it causes a crash or any pre-defined unacceptable behaviour, a vulnerability is reported. Moreover, the solution would be used as an evidence for existence of the vulnerability.

3.4 Extending the execution

After testing for existence of buffer overflow in the current executed path, it is time to extend the execution of the program into new paths. In this step, we use the same method as in [6, 7, 18] to generate new test data that traverse other execution paths. To do so, the path constraints are negated one by one, from the last to the first. After each negation the resulted path constraints are queried from the constraint solver. The generated solutions are used as the concrete test data to restart the fuzzing procedure from the first step. If the constraint solver cannot find a solution for any of the negated constraints, there would be no new execution path to explore. Therefore, the fuzzing procedure is ended.

4 Evaluation

We have implemented the proposed method as a plug-in for Valgrind. Valgrind is a framework for instrumenting the binary codes and implementing dynamic analysis solutions [25]. We also used another plug-in, called Fuzzgrind, that performs concolic execution and calculates the path constraints in the binary programs [18]. In fact, we have implemented our method by extending Fuzzgrind so that it calculates buffer overflow constraints in addition to the path constraints for each execution path. Fuzzgrind has been implemented for testing 32-bit binary programs on Linux, thus our current implementation works for 32-bit Linux applications. However, the proposed fuzzing method is also usable for binary programs on other types of operating systems. It is worth mentioning that as the taint analysis solution in Fuzzgrind is limited to the inputs from files and the keyboard, our implemented fuzzer is able to detect vulnerabilities that are exploited by the use of input data from files or the keyboard. Our implemented fuzzer is tested in a Backtrack VMware with 1 GB RAM and 1.8 GHz CPU.

Currently, Fuzzgrind does not support some data types, such as V128. These data types are usually used in real-world applications. For example, many optimising compilers transform particular parts of sequential instructions into equivalent parallel ones to speed up the program execution. This transformation leads to using V128 data type in the program. Without such optimisations, executing the program would be very slow. Therefore, the implemented fuzzer is not currently testable on real-world applications. In order to demonstrate the accuracy and efficiency of the proposed method, we have used a variety of test programs and designed different test scenarios that show the abilities of our fuzzer clearly. We postpone extending Fuzzgrind to handle other data types, and detecting vulnerabilities in real-world software to our future works. All the test programs that are used in this evaluation are compiled with 'gcc-O2' and with no debugging options enabled.

4.1 Test one: detecting vulnerabilities in different statements

In the first test, we evaluated the ability of our fuzzer in identifying possible vulnerable statements and generating vulnerability constraints for those statements. We show that our fuzzer is able to recognise different vulnerable statements and generate test data that cause buffer overflow in those statements. In this test, we have used Juliet_Test_Suite_v1.2_for_C_Cpp test suit, which is provided by NIST in SARD project [26] and contains a set of vulnerable programs written in C and C++ [27]. The SARD project is aiming at collecting different test suits to help the end users to evaluate tools and tool developers to test their methods. We have used the test suits that are provided in this project since we found NIST SARD offering the most comprehensive and up-to-date test programs.

The test programs in Juliet_Test_Suite_v1.2_for_C_Cpp are classified based on the classification of faults and vulnerabilities in CWE database [28]. We have tested our fuzzer on the programs of class CWE121_Stack_Based_Buffer_Overflow. The test programs in this benchmark contain one or more good functions and a bad function. Therefore, all these test programs contain one vulnerability. Good functions avoid a vulnerability by checking the (input) data value or using a legitimate static data value in critical operations. The bad function operates on input data directly with no previous checks. It might copy the input string in a loop or using a vulnerable C function, e.g. *memcpy()*, *strcpy()*, *strncpy()*. All the test programs in this benchmark define a static fixed string and copy it into a destination. Since our method creates vulnerability constraints for the instructions that store tainted data, we changed the source string in these programs so that it is read from a file.

Table 2 presents the results of our tests on a group of these test programs. The columns in this table represent, from left- to right-hand side, the name of the test program, a description about the vulnerability and the structure of the test program, the number of generated buffer overflow constraints by our fuzzer, the number of generated path constraints by our fuzzer, the number of generated test-cases, the number of reported crashes, the number of true positives (TP), FP, true negatives (TN) and false negatives (FN) in the test result and the duration of performing the test.

Table 2 Results of the first test on a group of programs selected from [27]

Program name	Description	# bof const	# path const	# test-cases	# crash	# TP	# FP	# TN	# FN	Time, s
loop_01	1 loop	1	0	1	1	1	0	1	0	1.63
memcpy_01	1 memcpy function	1	0	1	1	1	0	1	0	1.6
cpy_01	1 strcpy function	1	6	7	1	1	0	1	0	1.9
memove_01	1 memmove function	1	0	1	1	1	0	1	0	1.64
ncat_01	1 strncat function	1	6	7	1	1	0	1	0	1.86
snprintf_01	1 snprintf function	1	6	7	1	1	0	1	0	1.73
cat_01	1 strcat function	1	6	7	1	1	0	1	0	1.7

Names of the test programs are shorten because of lack of space. The complete name of each test program is the result of appending its name in the table to 'CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare'.

As shown in Table 2, each test program contains one vulnerable statement in its bad function that stores tainted data into the stack. Our fuzzer has generated one vulnerability constraint for the vulnerable statement in the bad function of each program. The different number of path constraints in this table is because of the different behaviour of the copy instructions. When the copy operation is performed by a *strncpy*, *strncat* or *snprintf* function, the binary code checks if each copied byte is equal to zero. If the copied character is Null, the program stops the copy operation. As an example, for a tainted string that is six-character long, it checks whether each of the six characters is Null. Therefore, our fuzzer generates six path constraints. However, when the copy operation is performed in a loop, *memcpy* or *memove* function, the executable does not perform any check on the value of the copied bytes. Thus, the fuzzer does not generate any path constraints for them.

The test-cases are generated by solving the path and vulnerability constraints. When all the path and vulnerability constraints are solvable, the number of generated test-cases is equal to sum of the number of path and vulnerability constraints. The number of generated test-cases in this table shows that STP could resolve all the path and vulnerability constraints and our fuzzer has generated one test-case based on each solution. Executing the program with the test-case that was generated based on the vulnerability constraint has led to a crash, and thus the fuzzer has reported a TP alarm. As our fuzzer has not generated any alarm for the copy operations in the good function of the test programs, one TN and zero False Positive (FP) alarm is recorded for each program in this table. Moreover, as the vulnerability in each program was detected by our fuzzer, one TP and zero FN alarm is recorded in each record of this table.

4.2 Test two: detecting vulnerabilities in more complicated paths

In the second test, we designed test programs that contain vulnerabilities in more complicated execution paths. The goal of this test is to better evaluate the ability of our fuzzer in combining path and vulnerability constraints and detecting vulnerabilities in more complicated execution paths. In fact, we changed the bad function in one of the test programs in the class CWE121_Stack_Based_Buffer_Overflow of Juliet_Test_Suite_v1.2_for_C_Cpp and made it more complicated. The resulted test programs contain one good function and one bad function. The details of the bad function in these programs are presented in Fig. 10. Moreover, the results of this test are presented in Table 3.

As an example, to detect the buffer overflow in test_1, our fuzzer calculated the path constraints in lines 3 and 4 of the program. Then, it generated input data that comply with those path constraints. By executing the program with that input data, our fuzzer reached the *strcpy* function and generated a vulnerability constraint for it. The vulnerability constraint was combined with the previous path constraints to generate new test data that cause overflow in line 5 of this program.

Moreover, the bad function in test_3 securely copies the first 10 bytes of the tainted source into a static variable. Then, it copies the 21th to the last byte of input data into that variable and causes a stack-based buffer overflow. The implemented fuzzer detected the vulnerability by making a padding in the range *(21, strlen (source))* of the path constraints and generating appropriate test data that comply with the constraints in line 9. There are two secure copy operations in this program: one in the good function and another in the first loop of the bad function. The fuzzer did not generate any alarms for them, thus two TN alarms and zero FP alarm are recorded for this test in Table 3.

The bad function in test_4 copies a tainted array into the stack using a recursive function. Detecting such vulnerability is challenging and time-consuming for static analysers, since they need to generate the program dependence graph for the program and analyse the relation between the called functions. It was, however, quickly detected by our fuzzer.

4.3 Test three: detecting vulnerabilities in simplified applications

Since the test programs in the previous tests contained only one vulnerability, in the third test we have used simplified versions of real-world applications that are more complicated and contain more vulnerabilities. The goal of this test is to verify if the fuzzer can handle large amount of constraints and detect different vulnerabilities in a program. The test programs in this test are chosen from SARD.testsuite-88.2014-12-21-10-39-47 test suit of SARD project [29]. This benchmark consists of simplified version of programs, such as NSlookup and Sendmail, which contain a number of buffer overflow vulnerabilities. Table 4 presents the results of testing our fuzzer on these programs. It is worth mentioning that the test program in the first row contains execution paths that depend on values of the environment variables. Since our implemented taint analysis method only considers the data read from file and keyboard as tainted, our fuzzer was not able to calculate the constraints of these paths. Thus, to evaluate the accuracy of the implemented fuzzer, we have considered the number of vulnerabilities that exist in the paths that do not depend on environment variables. Therefore, the description column for this program presents the number of vulnerabilities that exist in the feasible paths for the implemented fuzzer. These paths depend on the values of data that are read from a file or keyboard.

The FN column shows that there are undetected vulnerabilities for the test program in the second row. These vulnerabilities occur when a fixed untainted string is copied into the stack. Since the copied data are defined statically by the program and cannot help in exploiting the program by malicious users, our fuzzer did not calculate the vulnerability constraints for these operations. There are also some secure operations that store data into the stack in each program, which are shown in the TN column. Table 4 shows that our fuzzer did not falsely report those secure copy operations as vulnerabilities.

4.4 Test four: comparison with other fuzzers

In the fourth test, we compared our fuzzer with three other fuzzers to demonstrate how calculating the path and vulnerability constraints makes our fuzzer more efficient than the other fuzzers. We have compared our fuzzer with three well-known fuzzers of different intelligence levels, i.e. a basic fuzzer, a file format fuzzer and a

Name and description	Bad function in the test program

test_1	`static void bad(char * source) {`
	`char data[100];`
	`if (source[2]=='c') {`
	`if (source[3]=='b') {`
	`strcpy(data, source);//insecure copy`
	`}`
	`printf("%s", data);`
	`}`
	`}`

test_2	`static void bad(char * source){`
	`char data[100];`
	`if(source[2]=='d'){`
	`strncpy(data,source,50);//secure copy`
	`if (source[3]=='b'){`
	`strcpy(data, source);//insecure copy`
	`}`
	`}`
	`printf("%s",data);`
	`}`

test_3
First securely copies the first 10 bytes of the tainted source into the destination. Then, copies its 21th byte and the followings to the destination and causes buffer overflow.
	`static void bad(char * source){`
	`char data[100]; int i=0, j=0;`
	`while (j < 10){`
	`data[j]=source[i];`
	`j=j+1; i=i+1;`
	`}`
	`i=20; j=10;`
	`if (data[5]=='Q' && data[4]=='A') {`
	`for (i=20;i < strlen(source);i++) {`
	`data[j]=source[i]; j=j+1;`
	`}`
	`}`
	`printf("%s",data);`
	`}`

test_4
string copy in a recursive function.
	`int copy(char * data, char* source,int i){`
	`if (i > strlen(source))`
	`return 1;`
	`else {`
	`data[i]=source[i]; i=i+1;`
	`copy(data,source,i);`
	`} }`
	`static void bad(char * source){`
	`char data[100]; int i=0;`
	`if(copy(data,source,i)==1){`
	`printf("%s",data);`
	`}`
	`}`

Fig. 10 *Details of bad functions in the designed programs of test two*

Table 3 Results of the second test on a group of designed test programs

Program name	Description	# bof const	# path const	# test-cases	# crash	# TP	# FP	# TN	# FN	Time, s
test_1	1 strcpy based on the tainted value in nested if statements (if then (if then strcpy))	1	8	9	1	1	0	1	0	1.85
test_2	nested if statements with one secure strncpy and an insecure strcpy									
(if then strncpy_secure (if then strcpy_insecure))	2	14	16	1	1	0	2	0	3.51	
test_3	described in Fig. 10	2	9	11	1	1	0	2	0	11.10
test_4	string copy in recursive functions	1	0	1	1	1	0	1	0	1.79

Table 4 Results of the third test on simplified versions of some vulnerable real-world applications

Program name	Description	# bof const	# path const	# test-cases	# crash	# TP	# FP	# TN	# FN	Time, s
realpath-2.4.2	2 strcat, 1 strcpy	3	46	49	3	3	0	1 safe strcpy	0	43.27
mapped-path	2 strcpy, 2 strcat	2	33	20	2	2	0	3 safe strcpy	1 strcpy, 1 strcat	53.47
ns-lookup	2 sprintf	2	84	80	2	2	0	2 safe strcpy	0	114.71
lquery	1 memcpy	1	14	11	1	1	0	0	0	51

```
<User name='test2015' ip='127.0.0.1' addr='city, country' Ncode='123456789'>
```

The C++ code of the first test program is as follows:

```
1. int fread_string(int fd, char* s , int len)
2. {                                  // stores at most len bytes of input file into s.
3.  char ch = 0;
4.  int index = 0;
5.  read(fd, (char*)&ch, 1);
6.  if (ch != '\'')                   // the read string should start with '
7.    return 1;
8.  while ((index < len || len == -1) && read(fd, (char*)&ch, 1))
9.   {
10.   if (ch == '\'')                 // the read string should end with '
11.   {
12.      read(fd, (char*)&ch,1);
13.      break;
14.   }
15.   s[index++] = ch;
16.  }
17.  return 0;
18. }
19. int main(int argc, char *argv[]) {
20.  char header[10] = { 0, };
21.  char header1[10] = { 0, };
22.  char header2[10] = { 0, };
23.  char header3[10] = { 0, };
21.  char buffer[256] = { 0, };
22.  char addr[128]={0,};
23.  char username[34] = { 0, };
24.  char ncode [12]={0,};
25.  char ip[16] = { 0, };
26.  if (argc != 2) {
27.    printf("Usage: %s <file>\n", argv[0]); exit(-1);
28.  }
29.  int fd = open(argv[1], O_RDONLY);
30.  if (fd == NULL)
31.   {
32.    printf("Wrong file name\n");
33.    exit(-1);
34.   }
35.  read(fd , header , 6);
36.  if (strcmp(ip, "<User ") == 0)
37.  {
38.    read(fd, header1 , 5);
39.    if (strcmp(header1 , "name=") != 0)
40.        WRONG_FORMAT_RET;
41.    fread_string(fd, username , 32);
42.    read(fd, header2 , 3);
43.    if (strcmp(header2 , "ip=") != 0)
44.        WRONG_FORMAT_RET;
45.    fread_string(fd, ip , 14);
46.    read(fd, header3 , 5);
47.    if (strcmp(header3 , "addr=") != 0)
48.        WRONG_FORMAT_RET;
49.    fread_string(fd, addr , -1);
50.    read(fd, header4 , 6);
51.    if (strcmp(header4 , "NCode=") != 0)
52.        WRONG_FORMAT_RET;
53.    fread_string(fd, ncode , 10);
54.  }
55.  else
56.    WRONG_FORMAT_RET;
57.  printf("\tUser[%s] read with addr=%s and ip=%s\n", username,addr,ip);
58.  return 0;
59. }
```

Fig. 11 *Configuration file of test programs that is filled with sample data*

concolic execution-based fuzzer that only calculates the path constraints. Unfortunately most of the fuzzers that are developed in academia are not available. From the available fuzzers, we chose Microsoft Security Development Lifecycle (SDL) MiniFuzz as a basic fuzzer [30], Peach as a file format fuzzer [31] and Fuzzgrind+Memcheck as a concolic execution-based fuzzer.

Microsoft SDL MiniFuzz is a basic fuzzer that receives an input file as a seed and generates multiple random variations of the file content. This tool is proposed by Microsoft to be used in Microsoft SDL by the development teams. MiniFuzz randomly changes the characters of input data to make the program crash. It does not analyse the target program, nor does it extend the length of input data.

IET Softw., 2016, Vol. 10, Iss. 4, pp. 96–107

Peach is a well-known file format fuzzer that allows human analysers to define the format of the input file for it. In this definition, the name, the data type and length of the data in each field can be defined. It is also possible to determine which fields should be fuzzed to detect the vulnerabilities. Based on the defined format, Peach changes the content of input file to generate new test data. Peach also generates new test data by the use of mutation fuzzing method. In the mutation method, new test data are generated by slight modifications of the good data. The modifications may be performed randomly or based on some heuristics. As an example of heuristic mutations, the fuzzer changes the length of some data into larger values to detect buffer overflow vulnerabilities.

Fuzzgrind+Memcheck is a fuzzer that we created by combining the tools Fuzzgrind and Memcheck. Memecheck is a plug-in for Valgrind that detects memory errors and vulnerabilities in the binary codes [32]. Memcheck detects the vulnerabilities that are activated in the current execution path with current test data. In other words, it does not actively generate test data that traverse different execution paths and activate specific vulnerabilities in the executed path. In Fuzzgrind+memcheck, Fuzzgrind calculates the path constraints and generates test data that traverse new execution paths. The program is executed with new test data under the supervision of Memcheck. If a vulnerability becomes active in the current execution, Memcheck detects it. This fuzzer does not calculate any vulnerability constraints. Thus, it does not know which parts of input data affect on possible vulnerable statements or what is the minimum required length of input data that causes overflow in the program. This fuzzer does not extend, even randomly, the length of input data. The goal of comparing our fuzzer with this fuzzer is to show how calculation of vulnerability constraints, in addition to the path constraints, helps to fuzz the target program more efficiently.

We designed two test programs for this test. These programs read some data from a configuration file and store them in the stack memory. The configuration file contains four fields: name, ip, address and national code (Ncode). The following shows the configuration file of these test programs that is filled with sample data. (see Fig. 11)

In line 49 of this code, the program reads the *addr* value, by the use of *fread_string* function, but does not limit the length of the read data. Thus, there is a buffer overflow vulnerability in this program. To detect this vulnerability, the fuzzers should consider the format of the configuration file and extend the length of the *addr* value. In other words, the generated test data should comply with the path constraints in lines 36, 39, 43 and 47 to reach the vulnerable statement in line 49.

Table 5 presents the results of testing the first test program with these fuzzers. In this test an input file that is not well-formatted and contains the string 'aaaaaaaaaaaaaaaaaaaaaaaa' is used as the initial input file. Thus, the fuzzers should first recognise the acceptable format of the configuration file and then extend the length of each field to detect the buffer overflow. We performed this test by Peach with two different configurations. First, we defined the format of the input file and used the not-well-formatted file as the initial input file. The result of this test is presented in the second row of Table 5. Then, we used the not-well-formatted input file but did not define the format of the input file. The result of this test is presented in the third row of Table 5.

Table 5 Results of testing the first designed test program with a not well-formatted initial input file

Fuzzer name	Detected?	# generated test-cases	Code-coverage, %	Time, s
Microsoft SDL MiniFuzz	no	>60,000	28	>3600
Peach (knowing the format)	yes	22	100	11.31
Peach (not knowing the format)	no	>15,550	28	>3600
Fuzzgrind +Memcheck	no	>3567	100	>3600
Our fuzzer	yes	152	100	138

```
42. read(fd, header2 , 3);
43. if (strcmp(header2, "ip=") != 0)
44.     WRONG_FORMAT_RET;
45. fread_string(fd, ip, 20);
46. if (ip[1]=='1' && ip[2]=='9' && ip[3]=='2') {
47.     read(fd, header3 , 5);
48.     if (strcmp(header3, "addr=") != 0)
49.         WRONG_FORMAT_RET;
50.     fread_string(fd, addr, -1);
51. }
```

Fig. 12 *C code shows the different part in the second test program*

In this test, our fuzzer and Fuzzgrind+Memcheck calculated the path constraints and generated test data that were in the acceptable format. As shown in this table, these fuzzers achieved 100% code-coverage. The value of code-coverage is calculated by dividing the maximum number of lines of code that were executed by the fuzzer to the total number of lines of code in the program. Since Fuzzgrind+Memcheck did not extend the length of the *addr* value, it could not detect the buffer overflow vulnerability. Thus, only our proposed fuzzer could detect the vulnerability. The proposed fuzzer determined the length of the destination buffer, extended the length of the *addr* field into more than 128 bytes and detected the vulnerability in acceptable time.

When the format of the input file was defined for Peach, it achieved 100% code-coverage and detected the vulnerability in the test program. Thus, when Peach has enough knowledge about the format of the input file, the use of inappropriate initial input files does not have much effect on its efficiency. However, when Peach did not know the format of the input file, it could not detect the vulnerability in an hour. It only got 28% code-coverage and did not reach the vulnerable statement. MiniFuzz did not know the correct format of input file or the logic of the program either. Thus, it got only 28% code-coverage and could not detect the vulnerability. This test demonstrated that our fuzzer is able to generate appropriate test data even without knowing the required format of the input data.

The second test program is similar to the first one, except that it checks the value of the *ip* filed before reading *addr*. If the value of *ip* begins with '192', it reads the *addr* value. The following C code shows the different part in the second test program: (see Fig. 12)

To activate the vulnerability in this program, the generated test data should be consistent with the constraint on the value of *ip* address. Thus, the fuzzers should generate test data with *ip* values that start with '192'. We tested the fuzzers with this program and by using a well-formatted input file. In this file, the value of the *ip* field is '127.0.0.1'. Moreover, we defined the format of input file to Peach in this test. Table 6 presents the results of this test. In this test, our proposed fuzzer and Fuzzgrind+Memcheck calculated the path constraints and generated test data that complied with the constraints on the *ip* value. The generated test data caused execution of line 50 in the C code of this program. Although both of these fuzzers achieved 100% code-coverage, only our fuzzer calculated the vulnerability constraint for overflowing the *addr*

Table 6 Results of testing the second designed test program with a well-formatted initial input file

Fuzzer name	Detected?	# generated test-cases	Code-coverage, %	Time, s
Microsoft SDL MiniFuzz	no	>60,000	28	>3600
Peach (knowing the format)	no	>15,550	91	>3600
Fuzzgrind +Memcheck	no	>2182	100	>3600
Our fuzzer	yes	562	100	673.68

buffer and generated test data with appropriate *ip* and *addr* values. This shows that calculating the vulnerability constraints in addition to the path constraints helps to detect the vulnerabilities in complicated paths efficiently. Since Peach and MiniFuzz did not analyse the binary code of the target program, they were not aware of the path constraints in it and did not generate test data that meet these constraints. In this test, MiniFuzz and Peach achieved 28 and 91% code-coverage in an hour, respectively. Although Peach knew the format of input file, it was not aware of the path constraint in line 46 of the program. Thus, it could not generate test data that comply with that constraints and could not execute the lines 47 to 50 of this program. Therefore, only the proposed fuzzer could reach the vulnerable statement and detect the vulnerability in it.

These tests demonstrate that our fuzzing method detects vulnerabilities efficiently even when the human analyser does not have enough information about the format of the input data or the logic of the target program. Our fuzzer analyses the binary code and determines the appropriate format of the input data. Usually software programs consist of different execution paths. Execution of each path depends on a variety of conditions that the human analysers are not usually aware of them. Our fuzzer calculates the conditions of each execution path and generates test data accordingly. The generated test data traverse intended execution paths and detect vulnerabilities in those paths.

Although due to some technical limitations we did not test the implemented fuzzer on real-world applications, we performed different tests on a variety of benchmark programs and showed that the proposed fuzzing method is able to accurately detect vulnerabilities in different vulnerable programs. Moreover, by comparing the fuzzer with three other fuzzers, we demonstrated how calculation of the path and vulnerability constraints in our method reduces the number of generated test-cases to detect a vulnerability. This makes the fuzzing process more efficient by decreasing the fuzzing time and efforts. From this evaluation, it can be concluded that as the implemented fuzzer detected vulnerabilities efficiently in these test programs, the proposed smart fuzzing method can also be applied to detect stack-based buffer overflows in real-world applications more efficiently.

5 Conclusions

This paper presented a smart fuzzing method for detecting stack-based buffer overflows in the binary codes. The designed fuzzer uses concolic execution method to calculate the path and buffer overflow constraints in the program. For each executed path, it combines the calculated path and vulnerability constraints to generate test data that cause buffer overflow in that path. In order to calculate the vulnerability constraints, a method is suggested for estimating the length of variables in the stack. Based on the estimated lengths, vulnerability constraints are generated for the instructions that store tainted data into the stack. Our method expands the executed paths, by negating the calculated path constraints and solving the new set of constraints to generate test data that traverse other execution paths.

The proposed fuzzing method has several advantages that make it more efficient in comparison with the previous methods. First, by combining the path and vulnerability constraints, it generates data for detecting vulnerabilities in a specific execution path that comply with the constraints of that path. Thus, the test data traverse the specific path and activate the vulnerabilities in that path. Next, our method considers the index of the input bytes that are used in possible vulnerable statements. It also estimates the size of the stack buffers in the program and uses this information for generating new test data. Therefore, it does not change all parts of input data blindly to detect the vulnerabilities. Moreover, our method pre-processes the buffer overflow constraints in order to fuzz the loops and well-known vulnerable functions, e.g. *strcpy()*, with more priority. These advantages make our fuzzer more efficient by reducing the fuzzing time and efforts. The evaluations demonstrated that the specified vulnerability constraints in our method are accurate and our fuzzer detects the vulnerabilities in the test programs efficiently.

Another advantage of the proposed smart fuzzing method is that it can be extended to get more accurate by adding new vulnerability constraint calculation routines into it. In other words, extending the proposed fuzzer does not require essential changes in the algorithm and can be performed only by adding new vulnerability constraints. For example, our implemented fuzzer generates vulnerability constraints for the store instructions. It can also generate the constraints for load instructions in order to detect buffer over-read and buffer under-read vulnerabilities.

In the future, we are going to extend the fuzzer to detect other types of buffer overflow in the binary codes. We only estimated the length of variables that are stored in the stack. We will extend it to estimate the length of variables in heap to detect heap-based buffer overflows.

A challenge in applying the concolic execution method in fuzzing the programs is path explosion. In fact, the number of feasible execution paths increases exponentially in the large programs. Various optimisation techniques have been proposed against the path explosion [8], which are not currently implemented in our fuzzer. In the future we are going to improve the efficiency of our fuzzer by applying appropriate optimisation techniques to tackle the path explosion challenge. Moreover, we are going to extend our fuzzer so that it supports other data types, such as V128. With these enhancements, we would make our fuzzer applicable to real-world applications. We will also extend our implemented taint analysis method so that it considers the data from other sources, such as network sockets or environment variables, as tainted.

6 Acknowledgment

This work was supported in part by APA Research Center of Amirkabir University of Technology (Tehran Polytechnic).

7 References

1 Szekeres, L., Payer, M., Wei, T., *et al.*: 'Sok: Eternal war in memory'. Proc. 2013 IEEE Symp. on Security and Privacy (SP), 2013, pp. 48–62
2 Heelan, S.: 'Vulnerability detection systems: think cyborg, not robot', *IEEE Secur. Priv.*, 2011, **9**, (3), pp. 74–77
3 Cadar, C., Godefroid, P., Khurshid, S., *et al.*: 'Symbolic execution for software testing in practice: preliminary assessment'. Proc. 33rd Int. Conf. on Software Engineering, 2011, pp. 1066–1071
4 Cadar, C., Ganesh, V., Pawlowski, P.M., *et al.*: 'Exe: automatically generating inputs of death', *ACM Trans. Inf. Syst. Secur. (TISSEC)*, 2006, **12**, (2), pp. 10–24
5 Cadar, C., Dunbar, D., Engler, D.R.: 'Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs'. Proc. Eighth USENIX Conf. on Operating Systems Design and Implementation, 2008, vol. 8, pp. 209–224
6 Godefroid, P., Levin, M.Y., Molnar, D.: 'Sage: whitebox fuzzing for security testing', *Queue*, 2012, **10**, (1), pp. 20–27
7 Molnar, D., Li, X.C., Wagner, D.: 'Dynamic test generation to find integer bugs in x86 binary linux programs'. USENIX Security, 2009, pp. 67–82
8 Chen, T., Zhang, X.S., Guo, S.Z., *et al.*: 'State of the art: dynamic symbolic execution for automated test generation', *Future Gener. Comput. Syst.*, 2013, **29**, (7), pp. 1758–1773
9 Haller, I., Slowinska, A., Neugschwandtner, M., *et al.*: 'Dowsing for overflows: a guided fuzzer to find buffer boundary violations'. Proc. Symp. USENIX Security, 2013, pp. 49–64
10 Cowan, C., Pu, C., Maier, D., *et al.*: 'Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks'. Usenix Security, 1998, vol. 98, pp. 63–78
11 Microsoft: 'A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003'. Available at http://support.microsoft.com/kb/875352/EN-US, 2013. Accessed 10-12-2015
12 Team, P.: 'Pax document for linux aslr'. Available at https://pax.grsecurity.net/docs/aslr.txt/, 2013. Accessed 10-12-2015
13 Goktas, E., Athanasopoulos, E., Bos, H., *et al.*: 'Out of control: overcoming control-flow integrity'. Proc. 2014 Symp. on Security and Privacy (SP), 2014, pp. 575–589
14 Shacham, H.: 'The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)'. Proc. 14th ACM Conf. on Computer and Communications Security, 2007, pp. 552–561
15 Strackx, R., Younan, Y., Philippaerts, P., *et al.*: 'Breaking the memory secrecy assumption'. Proc. Second European Workshop on System Security, 2009, pp. 1–8
16 Snow, K.Z., Monrose, F., Davi, L., *et al.*: 'Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization'. Proc. 2013 IEEE Symp. on Security and Privacy (SP), 2013, pp. 574–588

17 Isaev, I., Sidorov, D.: 'The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs', *Program. Comput. Softw.*, 2010, **36**, (4), pp. 225–236
18 Campana, G.: 'Fuzzgrind: an automatic fuzzing tool'. Available at http://esec-lab.sogeti.com/pages/Fuzzgrind/, 2009. Accessed 10-12-2015
19 Ganesh, V., Dill, D.L.: 'A decision procedure for bit-vectors and arrays', *Comput. Aided Verif.*, 2007, pp. 519–531
20 De Moura, L., Bjørner, N.: 'Z3: An efficient smt solver'. Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 337–340
21 Balakrishnan, G., Reps, T.: 'Wysinwyx: what you see is not what you execute', *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2010, **32**, (6), pp. 23:1–23:84
22 Rawat, S., Mounier, L.: 'Finding buffer overflow inducing loops in binary executables'. Proc. Sixth IEEE Int. Conf. on Software Security and Reliability (SERE), 2012, pp. 177–186
23 Irvine, K.R., Das, L.B.: 'Assembly language for x86 processors' (Prentice Hall, 2011), pp. 270–331
24 Brumley, D.: 'Analysis and defense of vulnerabilities in binary code'. PhD Thesis, Carnegie Mellon University, 2008
25 Nethercote, N., Seward, J.: 'Valgrind: a framework for heavyweight dynamic binary instrumentation'. ACM Sigplan Notices, 2007, vol. 42, pp. 89–100
26 NIST: 'Software Assurance Reference Dataset (SARD)'. Available at http://samate.nist.gov/SARD/, 2015. Accessed 10-05-2015
27 SAMATE: 'Juliet Test Suite for C/C++ (v1.2)'. Available at http://samate.nist.gov/SARD/testsuite.php/, 2013. Accessed 01-05-2015
28 MITRE: 'Common Weakness Enumeration'. Available at http://cwe.mitre.org/, 2015. Accessed 10-12-2015
29 Rosenberg, E.: 'Testing Exploitable Buffer Overflows From Open Source Code'. Available at http://samate.nist.gov/SARD/testsuite.php/, 2014. Accessed 01-05-2015
30 Microsoft: 'SDL MiniFuzz File Fuzzer'. Available at http://www.microsoft.com/en-us/download/details.aspx?id=21769, 2011. Accessed 10-12-2015
31 Eddington, M.: 'Peach fuzzing platform', *Peach Fuzzer* 2011
32 Nethercote, N., Seward, J.: 'Valgrind: a framework for heavyweight dynamic binary instrumentation'. ACM Sigplan Notices, 2007, vol. 42, pp. 89–100