2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing

A Smart Fuzzing Method for Detecting Heap-Based Buffer Overflow in Executable Codes

(Regular Paper)

Maryam Mouzarani
Dep. of Computer Engineering
& Information Technology,
Amirkabir University of Technology,
Tehran, Iran.
Email: mouzarani@aut.ac.ir

Babak Sadeghiyan
Dep. of Computer Engineering
& Information Technology,
Amirkabir University of Technology,
Tehran, Iran.
Email: basadegh@aut.ac.ir

Mohammad Zolfaghari
Dep. of Electronic
& Computer Engineering,
Isfahan University of Technology,
Isfahan, Iran.
Email: zolfaghari@iut.ac.ir

***Abstract*—This paper presents a new concolic execution-based smart fuzzer for detecting heap-based buffer overflow in the executable codes. The proposed fuzzer executes the target program with concrete input data and calculates the constraints of the executed path symbolically. The path constraints are used to generate test data that traverse new execution paths in the target program. For each executed path, the fuzzer also calculates heap-based buffer overflow constraints. These constraints determine what input data may cause heap-based buffer overflow in the executed path. By combining the path and vulnerability constraints new test data are generated, if possible, that traverse a specific execution path and activate specific vulnerability in that path. We implemented the proposed smart fuzzer as a plug-in for Valgrind framework. The implemented fuzzer is tested on different groups of test programs. The experiments demonstrate that the fuzzer can detect the vulnerabilities in these programs accurately.**

I. Introduction

Buffer overflow is a well-known software vulnerability. In the past twenty years, various methods have been suggested to detect this vulnerability. Smart fuzzing is one of the effective methods suggested for detecting different vulnerabilities, including buffer overflow [10], [4]. In this method, an analysis is performed on the target software to gather more information about it. Based on this information, new test data are generated that traverse deeper paths in the program and increase the chance of detecting vulnerabilities.

Some smart fuzzers apply concolic (symbolic + concrete) execution in order to increase path coverage on the target program, e. g., [3], [2], [14], [18] and [8]. In this method, the program is first executed with some concrete input data. Then, the executed path is analyzed and its constraints are calculated symbolically. The calculated constraints are negated one by one, from the last to the first, and solved by a constraint solver, like STP [7] or Z3 [6]. If the constraints are solved, some input data will be generated based on the solution that traverse new paths in the programs.

Some concolic execution-based fuzzers are proposed for detecting specific vulnerabilities, e. g., [3], [2], [11]. Such fuzzers calculate symbolic vulnerability constraints for each execution path. The vulnerability constraints determine what input data may activate a specific vulnerability in an execution path. When the program is executed with concrete data, the vulnerability constraints are calculated symbolically. For each executed path, the symbolic path and vulnerability constraints are combined together. The combined constraints are then queried from a constraints solver. If the constraint solver returns a solution, new concrete input data will be generated that activate the vulnerability in the executed path and result in security errors. Calculating the vulnerability constraints and combining them with the path constraints result in a much more intelligent fuzzing. In this method, the test data are generated purposefully to execute a specific path and activate a specific vulnerability in it. These data can be used as a proof for existence of the vulnerability in the program.

In recent years more researchers are involved in designing smart fuzzers that analyze the executable codes, instead of the source codes, to detect software vulnerability. Reflection of the exact behavior of the program, optimizations and bugs in the compilers, unavailability of the source codes and platform-specific details are some reasons that make analyzing executable codes more preferable [1]. Sage and Smartfuzz are two concolic execution-based fuzzer that are proposed for detecting integer vulnerabilities in the executable codes [8], [14]. There is also a limited number of smart fuzzers that detect buffer overflow in the executable codes, like [9], [16]. However, none of the suggested methods calculates the constraints for this vulnerability.

In this paper we present the first concolic execution-based smart fuzzing method for detecting heap-based buffer overflow in the executable codes. In the proposed method the program is first executed with concrete input data. During the execution with concrete data, the constraints of the execution path are calculated symbolically. To reduce the number of path constraints, taint analysis is performed and only the constraints that depend on the tainted data are calculated. Besides the path constraints, the vulnerability constraints are calculated for each executed path. The calculated vulnerability constraints are combined with the path constraints and are queried from a constraint solver. If the solver finds a solution, it will be used to generate some data that execute the same execution path and cause an overflow in it.

Our proposed fuzzer performs taint analysis and generates vulnerability constraints only for the instructions that are affected by the input data. Hence, the vulnerability constraints take into account which bytes of input data may be involved

978-1-4673-9376-8/15 $31.00 © 2015 IEEE
DOI 10.1109/PRDC.2015.10

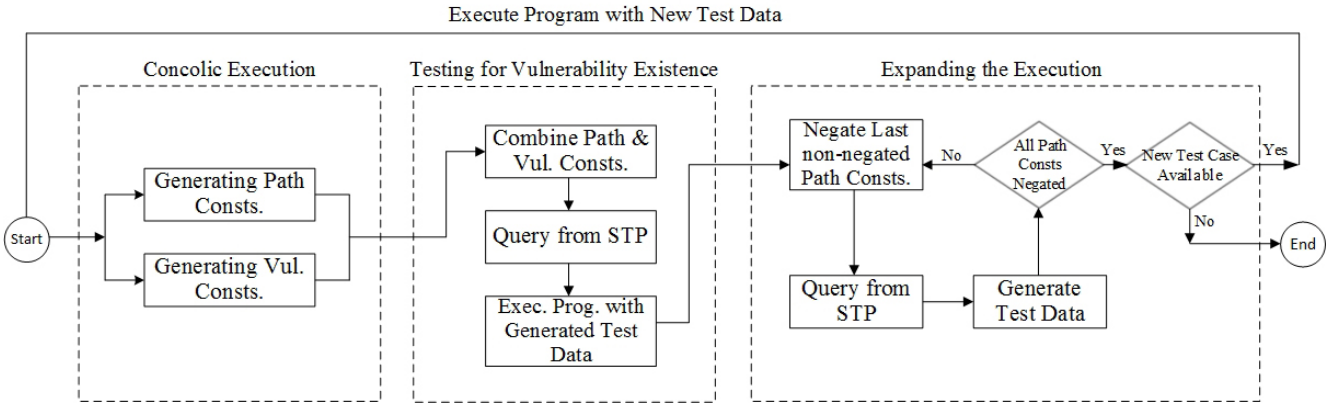

Fig. 1. Steps of the proposed smart fuzzer.

in overflowing a heap buffer. This helps in fuzzing the target program more intelligently. For example, if a specific field in a configuration file is used in a $strcpy()$ function, our method will focus on this field and does not extend the length of all the fields in the file. Also, the test data are generated by considering the constraints of the executed path. The data generated for detecting vulnerability in an execution path comply with the constraint of that path. Thus, the created test data traverse the specific execution path and reach intended instructions.

The proposed smart fuzzing method is described in section II. Section III describes the implementation details and the experiments. The paper is concluded in section IV.

II. A CONCOLIC EXECUTION-BASED SMART FUZZING METHOD

Our proposed smart fuzzer consists of three main steps: concolic execution, testing for vulnerability existence and expanding the execution. These steps are illustrated in figure 1. The details of each step are presented in the following sections.

A. Concolic execution

The proposed smart fuzzer applies concolic execution to gather more information about the target software. Concolic execution is a combination of pure symbolic execution and concrete execution. The pure symbolic execution is used in software testing to explore as many different execution paths as possible. For each path, this method generates a set of input data that results in executing that path. This data set is also used to check the existence of software vulnerability or other types of bugs in the executed paths. Symbolic execution is helpful in triggering a wide range of software errors from low-level program crashes to higher-level semantics properties [4]. An important problem with pure symbolic execution is that the constraints of complex loops and recursive functions may get very complicated and cannot be resolved in an acceptable time [19]. Using the concolic execution method, concrete input data are used to overcome the limitations of pure symbolic execution. In fact, the concrete data are used in concolic execution to simplify the constraints by changing some unknown variables into concrete data.

In our method, the executable program is instrumented in order to calculate the path and vulnerability constraints. Also, taint analysis is performed and the flow of tainted data is tracked while the program is executed with concrete data. Tainted data enter into the program from an un-trusted source, such as an input file, the keyboard or a socket. In the proposed fuzzer, the instructions that read data from input file or the keyboard are instrumented to consider the read data as tainted. However, it can be extended in the future to consider the received data from other un-trusted sources as tainted. The following sections explain how the path and vulnerability constraints are calculated for an executed path.

1) Generating the path constraints:

In the proposed method, the jump instructions are instrumented to generate path constraints. If the jump conditions depend on some tainted data, a new path constraint will be generated. For example, the following function contains two *if* statements before the *printf* statement. Each *if* statements is translated into a jump instruction in the executable codes. One of the jump instructions depends on a tainted variable. Therefore, one path constraint is generated for the execution path that contains *printf* statement.

```
void MySub()
{
char inputBuffer[10] = "";
int pathTrue=1;
fgets(inputBuffer, 10, stdin);
if (pathTrue==1)
{
if (inputBuffer[0]=='a')
printf("The input starts with a");
}
}
=> Generated path constraint: If tainted_byte(0)
       is_equal_to 0x61h
```

2) Generating the vulnerability constraints:

Besides the path constraints, symbolic vulnerability constraints are generated during the program execution. At this step, the program is executed with benign concrete input data. Such data are usually used in normal program execution by non-malicious users. Therefore, it is less probable that the vulnerabilities be activated during the concolic execution step. In fact, we calculate the vulnerability constraints to generate corrupted data that result in activating vulnerabilities in an execution path. By activating a vulnerability, we mean executing the

program with malicious input data and causing abnormal or non-privileged behavior in it.
To generate the vulnerability constraints, a table is generated during the concolic execution that is called the *memory table*. This table stores the addresses and lengths of dynamically allocated memory chunks. It is updated when a memory allocation or de-allocation occurs during the program execution. The following section describes the memory table.

The memory table

The memory table is constructed by instrumenting memory allocation and de-allocation function calls in the executable codes. For example, allocation and de-allocation of memory are performed in C and C++ programs by the use of functions such as *malloc()*, *new()*, *free()* , *delete()*, etc. Currently, our fuzzer instruments the memory management functions in the C and C++ languages. It can be extended in the future to instrument more memory management functions to detect the vulnerability in the executable codes of other programming languages.
When a memory chunk is allocated, a new record will be added to the memory table. This record contains the size and address of the new chunk. These values are obtained from the argument and return value of the called function, respectively. For example in the following C code, two memory allocations are performed using the *malloc* function.

```
void MySub()
{
char *x,*y;
x=(char*)malloc(10 * sizeof(char));
y=(char*)malloc(20 * sizeof(char));
strcpy(x, "hello");
strcpy(y, "world!");
printf("%s %s", x, y);
}
```

Table I shows contents of the memory table after these allocations. This table contains two records that indicate the size and address of the allocated memory chunks. There is also a validity column in the memory table. The value of this column is set to one by default for each newly added record. When a memory chunk is de-allocated, the validity field of the related record in memory table will be set to zero. If the value of such record is already zero, there is a double-free vulnerability in the program.
Our proposed fuzzer uses the memory table to generate vulnerability constraints and detect the vulnerabilities in the executable codes.

Heap-based buffer overflow constraints

Buffer overflow occurs when a program copies some data into a destination buffer with no previous range checking [13]. If the destination buffer exists in the heap memory, the vulnerability is called heap-based buffer overflow. Thus, this vulnerability is activated when some data are stored in a memory chunk that are longer than the size of the destination chunk. We instrument the store instructions in the executable code to generate the vulnerability constraints. The generated constraints check whether the source data can be larger than the size of the destination chunk. To be more efficient, the constraints are generated for the instructions that store tainted data. In fact, we only generate vulnerability constraints for possible vulnerable instructions that are exploitable by the use of malicious input data. Thus, for each *store(addr)=data* instruction, *data* is checked to be tainted or not. If yes, *addr* is searched through the memory table to find the valid chunk that contains *addr* in its range. In other words, the valid record in which *Address<addr<Address+Size* is chosen.
The size (S) and address (A) of the found record and the index of the tainted byte (B) are used to generate a primary vulnerability constraint as follows:

```
Heap overflows if the area around tainted_byte(B)
   in address(A) be longer than S;
```

As an example, table II presents a sample C code that copies a 3-character tainted source string into a buffer in a loop instruction. The equivalent X86 assembly of this code is presented in the second column. For each store instruction, a new primary vulnerability constraint is generated. The lines 6 to 18 show the copy operation in the loop structure. Lines 16 to 18 show that the loop is repeated until the loop counter, *i*, becomes greater than the length of *source* string. Therefore, the store instruction in line 11 is executed three times and three primary vulnerability constraints are generated. These constraints are presented in the constraint column of this table. At the end of executing the program, the primary buffer overflow constraints are processed. It is an attempt to merge the constraints that are generated for consequent store instructions in a loop or copy functions, such as *strcpy* or *memcpy*. The primary vulnerability constraints that are generated for the same chunk and are incremented one-by-one according to the tainted_byte number are merged into a single constraint. For example, table III presents some vulnerability constraints and the result of processing them. The merged constraints will be queried from the constraint solver with more priority. This is because the merged constraints usually belong to the loop or well-known copy functions and are more probable to be vulnerable.

B. Testing for vulnerability existence

Since there is a number of path constraints in the executed path, the generated test data for testing the vulnerabilities in that path should be consistent with these constraints. Hence, the test data can traverse the same execution paths and reach the possible vulnerable statements. We combine each vulnerability constraint with its previous path constraints. Combining the path and vulnerability constraints is performed by padding the path constraints and extending the length of the generated input data. For example, when the tainted bytes $(x, x + 5)$ are stored in a chunk with the length of S, 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 S to overwrite the neighbour chunks. 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 $i + S$, where $x + 4 < i < n + 1$ and n is the length of the current input data. Figure 2 illustrates such padding process.
Padding the specific bytes that are involved in possible vulnerable statements helps to have a 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. This makes the fuzzing much more efficient. Also, the new input satisfies the path constraints so it will reach the intended possible vulnerable statements.
By solving the combined constraints, new test data are generated that reach the intended possible vulnerable statements. The combined constraints are queried from a

ID	Address	Size	Validity
1	0x419a000	10	1
2	0x419a00a	20	1

TABLE I. SAMPLE MEMORY TABLE

C Code	Equivalent assembly	Generated constraints

void test(char * source)
{
int i;
char *dest;
dest=(char *) malloc(4* sizeof(char));
for(i=0; i<strlen(source); i++)
{
dest[i]=source[i];
}
printf("%s", dest);
return;
}
	1. movl $0x4,(%esp)
	2. call 804848c <malloc@plt>
	3. mov %eax,-0xc(%ebp)
	4. movl $0x0,-0x10(%ebp)
	5. jmp 80485a8 <test+0x34>
	6. mov -0x10(%ebp),%eax
	7. add -0xc(%ebp),%eax
	8. mov -0x10(%ebp),%edx
	9. add 0x8(%ebp),%edx
	10. movzbl (%edx),%edx
	11. mov %dl,(%eax)
	12. addl $0x1,-0x10(%ebp)
	13. mov -0x10(%ebp),%ebx
	14. mov 0x8(%ebp),%eax
	15. mov %eax,(%esp)
	16. call 804846c <strlen@plt>
	17. cmp %eax,%ebx
	18. jb 8048593 <test+0x1f >
	19. mov $0x80487a0,%eax
	20. mov -0xc(%ebp),%edx
	21. mov %edx,0x4(%esp)
	22. mov %eax,(%esp)
	23. call 804847c <printf@plt>
		1. Heap overflows if the area around tainted_byte(0) in address(0xbeceaf44) be longer than 4;
		2. Heap overflows if the area around tainted_byte(1) in address(0xbeceaf44) be longer than 4;
		3. Heap overflows if the area around tainted_byte(2) in address(0xbeceaf44) be longer than 4;

TABLE II. CREATING PRIMARY VULNERABILITY CONSTRAINTS FOR A SAMPLE CODE

Constraints before pre-processing	Constraints after pre-processing
Heap overflows if the area around tainted_byte(x) in address(0xbeceaf44) be longer than 4;	
Heap overflows if the area around tainted_byte(x+1) in address(0xbeceaf44) be longer than 4;	
Heap overflows if the area around tainted_byte(x+2) in address(0xbeceaf44) be longer than 4;	
Heap overflows if the area around tainted_byte(x+3) in address(0xbeceaf44) be longer than 4;	Heap overflows if the area around tainted_bytes([x,x+3]) in address(0xbeceaf44) be longer than 4;
Heap overflows if the area around tainted_byte(x+9) in address(0xbeceaf44) be longer than 4;	
Heap overflows if the area around tainted_byte(x+10) in address(0xbeceaf44) be longer than 4;	Heap overflows if the area around tainted_bytes([x+9,x+10]) in address(0xbeceaf44) be longer than 4;
Heap overflows if the area around tainted_byte(x+10) in address(0xbeceafa1) be longer than 32;	Heap overflows if the area around tainted_byte(x+10) in address(0xbeceafa1) be longer than 32

TABLE III. PROCESSING THE VULNERABILITY CONSTRAINTS.

constraint solver. If the constraint solver finds a solution, it will be used to generate test data that execute the same path and activate the specific vulnerability. The program is then executed with the test data. If it behaves abnormally or in a pre-defined manner, a vulnerability will be reported. For example, a vulnerability is reported if the program crashs, fails with a specific error number or output a specific message.

C. expanding the execution

After testing for existence of the specified vulnerabilities in the current executed path, the execution of the program is expanded into new paths. We use the same method as in [14], [5], [8] to generate test data that traverse new execution paths. To do so, the calculated 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 will be used as the 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 process will be ended.

III. IMPLEMENTATION AND EXPERIMENTS

We implemented the proposed smart fuzzing method as a plug-in for Valgrind framework. Valgrind is a framework for instrumenting the executable codes and implementing dynamic analysis solutions [15]. We used a plug-in, named Fuzzgrind, for calculating the path constraints and expanding the execution. Fuzzgrind is an automatic fuzzing tool that applies concolic execution to generate test-cases that traverse new execution paths [5]. In fact, our implemented smart fuzzer is an extension of Fuzzgrind that calculates the vulnerability constraints in addition to the path constraints for each executed path. It combines the calculated constraints to generate test data that execute a specific path and activate specific vulnerabilities in it.

There is a well-known plug-in for Valgrind, called Memcheck, that detects memory errors and vulnerabilities in the executable codes. This tool is able to detect invalid memory accesses, uses of uninitialized values, memory leaks, incorrect freeing of heap memory and overlapping of source and destination pointers in copy operation during the program execution. It monitors the status of memory during the program execution and generates error messages when an invalid operation is

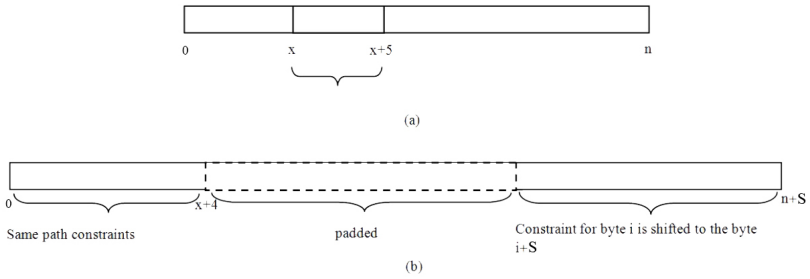

Fig. 2. Padding of the constraints. Part (a) shows the constraints before padding, and part (b) illustrates the constraints after padding.

Program Name	Description	# bof const	# path const	# test-cases	# crash	# TP	# FP	# TN	# FN	time (s)
char_ncpy_01	1 strncpy function based on source length	1	6	7	1	1	0	1 (1 good function copies untainted data)	0	1.85
char_ncpy_10	1 strncpy function based on source length (path depends on untainted data)	1	6	7	1	1	0	2 (2 good function copy untainted source into the dest)	0	1.86
char_ncpy_18	1 strncpy function (using GOTO to redirect into vulnerable path)	1	6	7	1	1	0	1 (1 good function the same as above)	0	1.66
char_ncpy_31	1 strncpy function based on source length (playing with the local vars: char *dataCopy = data; char* data = dataCopy;)	1	6	7	1	1	0	1 (1 good function same as above)	0	1.67
char_ncpy_41	1 strncpy same as above (nested functions in the vulnerable path: bad() calls badsink() good() calls goodsink()	1	6	7	1	1	0	1 (1 good nested function)	0	1.61
char_ncpy_51	1 strncpy (nested functions defined in different C files)	1	6	7	1	1	0	1 (1 good nested function defined in different files)	0	1.64

TABLE IV. RESULTS OF THE EXPERIMENTS ON THE FIRST GROUP OF TEST PROGRAMS WHICH HAVE DIFFERENT PATH COMPLEXITIES. 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 "CWE122_HEAP_BASED_BUFFER_OVERFLOW__CPP_CWE805_".

Program Name	Description	# bof const	# path const	# test-cases	# crash	# TP	# FP	# TN	# FN	time (s)
char_loop_01	1 loop based on source length	1	0	1	1	1	0	1	0	1.2
char_memcpy_01	1 memcpy function	1	0	1	1	1	0	1	0	1.4
char_ncat_01	1 strncat function	1	6	7	1	1	0	1	0	1.48
char_memove_01	1 memmove function	1	0	1	1	1	0	1	0	1.42
char_ncpy_01	1 strncpy function	1	6	7	1	1	0	1	0	1.51
char_snprintf_01	1 snprintf function	1	6	7	1	1	0	1	0	1.7

TABLE V. RESULTS OF THE EXPERIMENTS ON THE FIRST GROUP OF TEST PROGRAMS THAT USE DIFFERENT VULNERABLE FUNCTIONS. 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 "CWE122_HEAP_BASED_BUFFER_OVERFLOW__CPP_CWE805_".

Program Name	Description	# bof const	# path const	# test-cases	# crash	# TP	# FP	# TN	# FN	time (s)
test_1	1 strcpy in nested if statements (if then (if then strcpy))	1	10	10	2	3	0	1	0	2.1
test_2	nested if statements with one secure strncpy and an insecure strcpy (if then strncpy_secure)if then strcpy_insecure))	2	16	18	1	1	0	2	0	1.12
test_3	Described in VII	2s	11	13	1	1	0	2	0	11.10
test_4	string copy in recursive functions	1	1	1	1	1	0	1	0	1.0

TABLE VI. RESULTS OF THE EXPERIMENTS ON THE DESIGNED TEST PROGRAMS.

performed. However, Memcheck does not actively detect vulnerabilities in the target program. If a vulnerability be activated in the current execution path with current test data, Memcheck will detect it. In other words, it does not actively generate test data that traverse different execution paths and activate specific vulnerabilities in the executed path. On the

other hand, our proposed smart fuzzer generates test data that traverse as many execution paths as possible to detect the vulnerabilities in them.

We tested our algorithm on two groups of benchmarks. The first group is chosen from The Juliet_Test_Suite _v1.2_for_C _Cpp test suit which contains a set of vulnerable programs written in C and C++ [17]. These vulnerable programs are classified based on the classification of faults and vulnerabilities in CWE database [12]. 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 changing it to a fixed value before using in critical operations. The bad function operates on input data directly with no previous checks. For example, it might copy the input string in a loop or by the use of a vulnerable C function, e.g., $memcpy()$, $strcpy()$, $strncpy()$.

Tables IV and V present the results of our test on a group of test programs that are chosen from this benchmark and contain heap-based buffer overflow. The columns in these tables present, from left to right, the name of test program, a description about the vulnerability and the complexity of the test program, the number of vulnerability constraints, the number of path constraints, the number of generated input test-cases (test data created by solving the path and vulnerability constraints), the number of reported crashes, the number of TP (True Positives), FP (False Positives), TN (True Negatives) and FN (False Negatives) in the test result and the duration of performing the test in seconds. The name of these test programs are shorten in these tables because of the lack of space. The name of each test program should be appended to "CWE121_Stack_Based_Buffer _Overflow__CWE805 _char_declare" in order to have its complete name.

Table IV presents the results for testing our tool on the programs with the same vulnerable statement but different path complexities. The vulnerable statement in each program in this table is in a different execution path. The description column in table IV presents a short description about the complexity of each test program. The results in this table show that our fuzzer could detect the vulnerability in these test programs.

The path constraints for these test programs are generated during the copy operation in a copy function, such as *strcpy* or *memcpy*. For each copied character, the executable code checks whether the character is Null or not. If the copied character is Null, the program stops the copy operation. As an example, for a tainted string that is 6-character long, it checks whether each of the 6 characters is Null. Therefore, our fuzzer generates 6 path constraints. Since the path constraints in the programs of table IV depend on fixed and un-tainted data, the number of generated path and vulnerability constraints are similar in these programs.

The test programs of table V contain different vulnerable statements such as *strcpy*, *memcpy*, *strncpy* and *snprintf*. Our fuzzer instruments these statements and generates the vulnerability constraints for the arguments of these functions. In order to evaluate the implemented fuzzer more accurately, we created more complex test programs. The vulnerable statements in these programs are located in more complex execution paths. These programs are designed to test the ability of combining path and vulnerability constraint in our fuzzer. In fact, we changed the bad function in one of the test programs in the class CWE122_Heap_Based_Buffer_Overflow__cpp_CWE805_ and made more complexity in it. The resulted test programs contain one good function and one bad function. The details of the bad function in these programs are presented in Table VII. For example, to detect the vulnerability in test_2, the fuzzer should generate some test data that meet the path conditions in lines 4 and 6. Then, by executing the program with such test data, it calculates the vulnerability constraints and combines it with the path constraints. The solution for the resulted set of constraints should traverse the same path and causes an overflow. The implemented fuzzer detects this vulnerability accurately. It takes more time, in comparison with the results in tables IV and V, to test these test programs. It is because the fuzzer needs some time for creating input data that reach the vulnerable statements. Also, the bad function in test_3 securely copies the first 5 bytes of the tainted source into the dynamic variable. Then, it copies the input bytes, from the 6th to the last byte, into the dynamic variable and causes a heap-based buffer overflow. The implemented fuzzer made a padding in the range *(5, strlen(source))* and detected the vulnerability.

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 analyzers, since they need to generate the program dependence graph for the program and analyze the relation between the called functions. It is, however, quickly detected by our fuzzer.

IV. Conclusion

This paper presented the first concolic execution-based smart fuzzing method for detecting heap-based buffer overflow in the executable codes. The proposed fuzzer executes the binary program and calculates the path and vulnerability constraints symbolically for the executed path. It solves the combined constraints to generate test data that traverse the executed path and detect the vulnerabilities in that path. The fuzzer negates the path constraints one by one and solves the resulted constraints to generate test data that traverse new execution paths.

The advantage of the proposed fuzzing method is that it combines the buffer overflow constraints with the path constraints. Thus, the generated data for testing a vulnerability in a specific execution path comply with the constraints of the path. In this way, the test data traverse the specific path and activate the vulnerabilities in it. Another advantage of our fuzzing method is that it considers the index of the input bytes that are effective in vulnerable statements. Therefore, the fuzzing process is not performed on all parts of input data. This makes the fuzzing more efficient by reducing the fuzzing duration.

In the future, we are going to enhance the taint analysis step of the implemented fuzzer so that the fuzzer considers data from other un-trusted sources as tainted. Currently, we consider data from keyboard and input files as tainted. It can be extended to other un-trusted sources, such as the network sockets. Also, we will improve the calculation algorithm of taint propagation. This algorithm now propagates the tainted data in the direct assignment and arithmetic operations. There are, however, more complex taint propagation scenarios. For example, tainted data might be used in the conditional instructions and change the flow of the program.

Name and description	Bad function in the test-case

test_1
The input string ”source” is tainted and 8 characters long.
	static void bad(char * source) {
	data = (char *)malloc(100*sizeof(char));
	if (source[2]==’c’) {
	if (source[3]==’b’) {
	strcpy(data, source);//insecure copy
	free(data); }
	free(data);
	printf(”%s”, data);
	} }

test_2	static void bad(char * source) {
	data = (char *)malloc(100*sizeof(char));
	data2 = (char *)malloc(5*sizeof(char));
	memset(&data, 0, 100);
	memset(&data2, 0, 5);
	if (source[2]==’d’) {
	strncpy(data2,source,strlen(data2));//secure_copy
	if (source[3]==’b’){
	strcpy(data, source);//insecure copy
	} } //first if
	printf(”%s”,data);
	}

test_3
First securely copies the first 5 bytes of the tainted source in the destination. (It is to check false positive) Then it copies the 6th byte and the followings to the destination (in order to detect heap-based buffer overflow the padding is performed after the 5th bytes of the input string).
	static void bad(char * source){
	data = (char *)malloc(100*sizeof(char));
	int j, i;
	j=0; i=0;
	while (j <5)
	{
	data[j]=source[i];
	j=j+1; i=i+1;
	}
	j=5;
	if (data[5]==’Q’ && data[4]==’A’)
	{
	for (i=5;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){
	data = (char *)malloc(100*sizeof(char));
	int i; i=0;
	if(copy(data,source,i)==1){
	printf(”%s”,data);
	} }

TABLE VII. THE DETAILS OF BAD FUNCTIONS IN THE TEST PROGRAMS OF TABLE VI.

Also, there can be other heuristics for padding the constraints, like padding the middle part or starting part of the range $(x, x + 5)$ in figure 2. Since the experiments demonstrate the suitability of the current heuristic, we did not examine the others in our work. In the future, we are going to present new methods for padding the constraints. The efficiency of each method should be analyzed and the most efficient padding strategy should be selected.

ACKNOWLEDGMENT

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

REFERENCES

[1] G. Balakrishnan and T. Reps, “Wysinwyx: What you see is not what you execute,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 23:1–23:84, 2010.

[2] C. Cadar, D. Dunbar, and D. R. Engler, “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*, vol. 8, 2008, pp. 209–224.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “Exe: automatically generating inputs of death,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 10–24, 2006.

[4] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, “Symbolic execution for software testing in practice: preliminary assessment,” in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 1066–1071.

[5] G. Campana, “Fuzzgrind: an automatic fuzzing tool,” http://esec-lab.sogeti.com/pages/Fuzzgrind/, 2009, [Online; accessed 2015-01-05].

[6] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[7] V. Ganesh and D. L. Dill, “A decision procedure for bit-vectors and arrays,” in *Computer Aided Verification*, 2007, pp. 519–531.

[8] P. Godefroid, M. Y. Levin, and D. Molnar, “Sage: whitebox fuzzing for security testing,” *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[9] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, “Dowsing

for overflows: A guided fuzzer to find buffer boundary violations." in *USENIX Security*, 2013, pp. 49–64.

[10] S. Heelan, "Vulnerability detection systems: Think cyborg, not robot," *IEEE Security and Privacy*, vol. 9, no. 3, pp. 74–77, 2011.

[11] I. Isaev and D. Sidorov, "The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs," *Programming and Computer Software*, vol. 36, no. 4, pp. 225–236, 2010.

[12] MITRE, "Common Weakness Enumeration," http://cwe.mitre.org/, 2015, [Online; accessed 2015-06-05].

[13] MITRE-CWE, "CWE-122: Heap-based Buffer Overflow," https://cwe.mitre.org/data/definitions/122.html/, 2015, [Online; accessed 2015-06-05].

[14] D. Molnar, X. C. Li, and D. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs." in *USENIX Security Symposium*, 2009, pp. 67–82.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[16] S. Rawat and L. Mounier, "Finding buffer overflow inducing loops in binary executables," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 177–186.

[17] SAMATE, "Juliet Test Suite for C/C++ (v1.2)," http://samate.nist.gov/SARD/testsuite.php/, 2013, [Online; accessed 2015-01-05].

[18] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," *SIGSOFT Software Engineering Notes*, vol. 30, no. 5, 2005.

[19] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *Computer Security–ESORICS 2011*, 2011, pp. 210–226.