



A New Fuzzing Technique for Software Vulnerability Mining

Zhiyong Wu¹

J. William Atwood²

Xueyong Zhu³

^{1,3}Network Information Center University of Science and Technology of China
Hefei, Anhui, China

wuzhiyong0127@gmail.com, zhuxy@ustc.edu.cn

²Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada
bill@cse.concordia.ca

Abstract - Test case mutation and generation (m&g) based on data samples is an effective way to generate test cases for Knowledge-based fuzzing, but present m&g technique is only capable of one-dimensional m&g at a time, based on a data sample, and thus it is impossible to find a vulnerability that can only be detected by multi-dimensional m&g. This paper proposes a mathematical model FTSG that formally describes Fuzzing Test Suite Generation based on m&g, and can process multi-dimension input elements m&g, which is done by a Genetic Algorithm Mutation operator (GAMutator). By execution-oriented input-output (I/O) analysis, the influence relationships between input elements and insecure functions in target application were collected. Based on these relationships, GAMutator can directly mutate corresponding input elements to trigger the suspected vulnerability in a target insecure function, which could never have been found by one-dimension m&g fuzzing. Importantly, GAMutator does not bring the input combination explosion, and the number of test cases it generates is linear with the number of insecure functions. Finally, an experiment on Libpng has proved that FTSG could effectively enrich the ability of knowledge-based fuzzing technique to find vulnerabilities.

I. INTRODUCTION

Fuzzing [12] is a kind of software vulnerability mining technique, which combines random testing and boundary testing, symbolic execution, protocol knowledge and attack knowledge, concrete execution and probing attack method. Present advanced fuzzing techniques can be divided into two categories. One is whitebox fuzzing [4][5], which combines with static analysis, symbolic execution, concrete execution and other whitebox testing techniques to produce high code coverage fuzzing. The other is knowledge-based fuzzing [1], which generates test cases based on format

analysis. Outstanding tools such as SPIKE [1], Peach¹, Sulley² and AutoDafe³ belong to this class. Test cases that are generated by mutation based on a well-formed data sample, according to the file format or network protocol that is being examined, could effectively pass some program checks in applications such as static field, program checksum, and count or length calculation, which greatly improves the test data's validity.

However, it is hard for whitebox fuzzing to automatically pass strong program checks in real-world application such as hard-coded hash, encryption and decryption, program checksum operation. The shortcoming of present knowledge-based fuzzing is that it only implements one-dimension mutation and generation (m&g), which means only mutating one input element at a time to form a new test case. It does not have the multi-dimensional m&g, such as combinatorial testing in black-box testing. However, certain vulnerabilities can only be triggered by some special combination of multi-dimensional input, and thus some vulnerabilities will be missed.

The vulnerable code in Figure 1 provides an example. It is hard for automatic constraint solving to pass the program checksum verification (see line 6) to trigger the vulnerability in the insecure function in line 12. Although popular knowledge-based fuzzing can pass program checksum verification, the one-dimension m&g strategy cannot simultaneously trigger the vulnerability in line 12 when, for example, $length(head_str) = 16$ and $length(data_str) = 20$, i.e.,

¹ Peach, <http://www.peachFuzzer.com>

² Sulley, <http://www.fuzzing.org>

³ AutoDafe, <http://autodafe.sourceforge.net>

the length of normal initial values of *head_str* and *data_str* do not reach their bounds. Only two-dimension m&g on both of them can generate the combination of test cases to trigger this vulnerability.

```

1.  int process_chunk(char* head_str, char* data_str,
    char* program_checksum){
2.  charbuf[60];
3.  char buf1[32];
4.  char buf2[32];
5.  memset(buf, 0,60);
6.  if ( true == strong_check(head_str,data_str,program
    checksum)){
7.      If (strlen(head_str) > 32 || strlen(data_str) >32)
8.          return -1;
9.      strcpy(buf1, head_str);
10.     strcpy(buf2, data_str);
11.     strcat(buf, head_str);
12.     strcat(buf, data_str);//error
13.     return 1;
14. }
15. else
16.     Return -1;
17. }

```

Fig 1: C code of a vulnerable procedure

We propose a new fuzzing model named FTSG to overcome the shortages above.

II. METHODOLOGY

2.1 FTSG

To understand the m&g principle of knowledge-based fuzzing, we introduce a universal mathematical system FTSG, i.e., Fuzzing Test Suite Generation. In FTSG, we express all input elements in the form of a tree and all operations are based on this tree structure.

The tree structure can correctly express the sequential or nested relationships among elements in network protocols or file structures. Thus, a data m&g strategy based on tree structure can not only mutate one single element as a leaf node, but also can effectively mutate a set of input elements as a nonleaf node, and is able to mutate the structure of some protocol or file. In Section IV, we will use the functions of the Portable Network Graphics library defined in RFC 2083 [13] as our test subjects.

FTSG is as follows:

$$FTSG = (s, L, N, C, F, OP, Result),$$

$$OP = \{M, Slv\},$$

$$Result = \{sampletree, mediumtree, newtree, testcase, testsuite\}.$$

In which, s is a data sample.

L , leaf node set, $L = \{l_1, l_2, \dots, l_i, \dots, l_p\}$, l denotes an indivisible semantic unit in network protocols or file structures, such as *signature*, *Width*, or *Height* in RFC 2083.

N , NonLeaf node set, $N = \{n_1, n_2, \dots, n_j, \dots, n_q\}$, n denotes a divisible semantic unit in network protocols or file structures, for example, *IHDR_chunk* in RFC 2083 is made up of a few of leaf nodes, such as *Width* and *Height*.

C is a set of constraints, which represents the constraints of nodes' attributes.

F , insecure function set, $F = \{f_1, f_2, \dots, f_e, \dots, f_v\}$, f denotes insecure functions in target application that are influenced by l or n .

sampletree is a tree whose framework and relationships between nodes come from L , N and C , and the nodes' initial values of *sampletree* are set by the data sample s .

M is a set of mutation operators, Slv is a constraint solver.

OP denotes relative operations set, see Figure 2.

```

M = {m1, ..., mi, ..., mk, GAMutator}
F = {f1, f2, ..., fe, ..., fv}
for (each mi in M except GAMutator)
{
    while (!(mediumtree = mi (sampletree)))
    {
        newtree = Slv(mediumtree, C)
    }
}
for (each fe in F)
{
    while (!(mediumtree = GAMutator (sampletree, fe)))
    {
        newtree = Slv(mediumtree, C)
    }
}

```

Fig 2: Procedure of Generating test cases by Mutation Operators and Slv

A *mediumtree* comes from a *sampletree* whose attributions are changed by a mutation operator m_i .

Some of *mediumtree*'s nodes attributes may not meet

the constraints such as program checksum in C , Slv can transform them to change a *mediumtree* into a *newtree*.

All leaf nodes of a *newtree* constitute a *testcase*, and all the *testcases* make up a set, i.e., *testsuite*.

Let $|m_i(\text{sampletree})|$ denote the number of test cases from a *sampletree* that is manipulated by mutation operator m_i , then the total number of test cases would be :

$$T = |\text{testsuite}| = \sum_{i=1}^k |m_i(\text{sampletree})| \quad (1)$$

Some currently available mutation operators m_i can mutate the l or n to form a *mediumtree*, and then change other corresponding nodes by Slv to meet the constraints among nodes in C , and form a half-valid test case. But the m_i can only mutate one single node at a time, whether l or n , it cannot process multi-dimensional mutation operations, which will miss some vulnerabilities. However, the extended mutation operator *GAMutator* in M can process multi-dimension mutation operations.

2.2 Static analysis, dynamic binary instrument and dynamic trace

Static analysis technique is used to identify insecure functions including standard C library functions such as *strcpy()* and other insecure functions written in target executables. We developed a script based on **IDA PRO**⁴ to search for these insecure functions.

Dynamic binary instrument technique helps to get insecure functions' dynamic input arguments values to calculate fitness value for every *testcase* generated in *GAMutator*. This technique could be achieved by a dynamic binary instrument tool **Pin** [11].

Dynamic trace technique could be used to monitor buffer coverage by setting memory breakpoints. This technique is used to calculate fitness value sometimes when dynamic binary instrument technique does not work well. We developed a debugger based on **Pydbg**⁵.

2.3 I/O analysis

⁴ <http://www.datarescore.com/idabase>

⁵ <http://www.pedram.redhive.com/PaiMei/docs>

If we take input nodes l or n as inputs and insecure functions' arguments as outputs, by analyzing the execution records relating to them, we could construct influence relationships between them. This technique is called I/O analysis [10]. I/O analysis could both be implemented by static analysis on source code or execution-oriented analysis on binary code.

The former would produce many redundant results, e.g., some insecure functions could not be covered based on s , and more influence relationships between input elements and insecure functions could be generated due to inherent false alarm of symbolic execution, while the latter one is simpler and more precise.

Let O be the set of all outputs, and $O = \{o_1, o_2, \dots, o_k, \dots, o_n\}$. For each element of L and N , there is a set of test data for each of the inputs: $D(l_1), D(l_2), \dots, D(l_p), D(n_1), D(n_2), \dots, D(n_q)$. Let $X = \{x_1, x_2, x_3, \dots, x_s, \dots, x_{p+q}\}$, in which, $x_1=l_1, x_2=l_2, \dots, x_p=l_p, x_{p+1}=n_1, x_{p+2}=n_2, \dots, x_{p+q}=n_q$. An input x_s influences output o_k if and only if there are three test data inputs: $t_1 = (a_1, a_2, \dots, a_s, \dots, a_n)$ and $t_2 = (a_1, a_2, \dots, a_s, \dots, a_n)$ and $t_3 = (a_1, a_2, \dots, a_s', \dots, a_n)$ where $a_i \in D(x_i), a_s' \in D(x_i), a_s \neq a_s'$ such that when the program under test is executed on test data t_1 , the output value of o_k is the same as the output value of o_k on execution of test data t_2 and different from t_3 . We set t_1 equal to t_2 because some output changed themselves without influence of inputs.

Different from I/O analysis in black-box testing, sometimes output function arguments do not have values, because these functions were not executed with some input. Only results with effective values are considered here.

2.4 GAMutator

GAMutator is a special mutation operator we designed and implemented to mutate relative l or n in *sampletree* to trigger suspected vulnerability in f_e . l or n are the inputs that influence f_e . There are some differences between *GAMutator* and other single-dimension mutation operators m_j in M :

- *GAMutator* is a multi-dimension mutation operator that can mutate several input elements at the same time so that it could generate the *testcases* to trigger one or more vulnerabilities. However, these *testcases* would never be generated by m_j in M .
- *GAMutator* is a demand-oriented operator while m_j in M is node-type-specific operator. The nodes that *GAMutator* operates on are not decided by itself

but by the f_e in F , which largely reduces the number of combinations of the nodes.

- The number of test cases GAMutator generates may be different even with the same f_e in F while m_j in M generates a fixed number of *testcases* with the same *samplertree*, because the genetic algorithm is an intelligent advanced random algorithm.
- GAMutator directly and dynamically mutates l or n with some feedback information from the last generations' result, but m_j in M just runs alone without any communication with outside system.
- The genetic algorithm is used in GAMutator to generate test cases to trigger vulnerability in f_e in F but not to cover some specific path or statements [7][8][9].
- Let p denote individual number of every generation of GAMutator, g denote the maximum generations, and h denote the number of insecure functions collected by I/O analysis. So the maximum number of test cases generated by GAMutator is pgh . Since p, g are constants set by users, the number of test cases generated by GAMutator is $O(h)$.

According to different f_e in F , GAMutator mutates corresponding input nodes, which could be identified by I/O analysis. GAMutator is directed by some heuristics that are obtained by dynamic binary instrumentation and dynamic tracing to an insecure function.

2.5 Heuristics and fitness function

Heuristics are used to generate test cases more likely to trigger vulnerability in f_e in F . For example, standard library function *strcpy(dst, src)* is an insecure function that may trigger buffer overflow vulnerability, If the length of *src* (denoted as $len(s)$) is longer than the space size of *dst* buffer (denoted as $size(d)$), *dst* buffer will be overflowed. When the length of *src* is 0, *dst* buffer will never be overflowed, and its fitness function is designed as:

$$f(x) = \begin{cases} \frac{size(d)}{len(s)} & \text{if } len(s) \neq 0, \\ MAX_DEFAULT_FITNESS, & \text{if } len(s) = 0. \end{cases} \quad (2)$$

$MAX_DEFAULT_FITNESS$ is a global, manually-set constant. It is set to be larger than the largest value that

can be generated by the genetic algorithm in use. In the *strcpy()* example, it is used to represent the "badness" of the test case where $len(s) = 0$, which is the worst *testcase*, because it cannot possibly trigger the vulnerability.

Let us take the insecure function *malloc(a)* as another example that could trigger integer overflow vulnerability. Its fitness function is designed as:

$$f(x) = \begin{cases} A - a, & \text{when } a < A, \\ 0, & \text{when } a \geq A \text{ and } (a\%A) < B, \\ a\%A - B, & \text{when } a \geq A \text{ and } (a\%A) > B. \end{cases} \quad (3)$$

In which, $A = 0x\text{fffffff} + 1$, B is set by users, which means the memory space *malloc(a)* could successfully allocate and it relates to the usage of the whole memory.

III. PROTOTYPE SYSTEM

Based on the research and analysis to **FTSG**, we present prototype system DXFuzzing using the architecture of **FTSG**.

3.1 DXFuzzing

DXFuzzing (Directed Extended Fuzzing) is a new fuzzer tool developed by our team. It can fuzz both file formats and network protocol applications. DXFuzzing fuzz tests a target application two times, the first time using m_j in M , and the second time, based on some analysis, where it intelligently and directly fuzzes f_e in F with GAMutator.

DXFuzzing has five main components: Scheduling Engine, XFuzzing, Program Analyzer, Data Mapper, and GAMutator.

- The Scheduling Engine is responsible for scheduling a complete fuzzing flow. It starts one-dimension fuzzing with m_j in M and then organizes multi-dimension fuzzing with GAMutator.
- XFuzzing tool is a Fuzzer tool we developed that can fuzz both file formats and network protocol applications. It inherits many characteristics from Peach, such as using xml script to describe data elements in the form of a tree, the constraints between them, and designing data mutation operators according to their types and so on; it also inherits some techniques from Sulley, which can collect and analyze test results.
- Program Analyzer could statically locate the address of insecure functions in an executable

binary file and dynamically instrument them to collect runtime information from their arguments. Program Analyzer is implemented based on IDA Pro and Pin.

- Data Mapper is used to construct influence relationships between x_s in X in *sampletree* and f_e in F . Influence relationships are expressed by a table. The runtime information is obtained when a target is fuzzed the first time by m_j in M .
- GAMutator is a new mutation operator in DXFuzzing, it generates new *testcases* according to genetic algorithm. It gets the influence relationships from Data Mapper and mutates several l_i or n_j relative to f_e in F , which are being monitored by Program Analyzer.

The overall work procedure of DXFuzzing is shown in Figure 3.

1. Locate insecure functions positions in target binary code by Program Analyzer. Record their information into database.
2. Analyze corresponding network protocols or file format in target application according to related knowledge, choose a sample file s and write a primitive xml test script manually, which contains a *sampletree*;
3. The Scheduling Engine calls XFuzzing to fuzz target application with m_i and record runtime information with Program Analyzer when it is necessary.
4. The Data Mapper constructs relationships between X and F based on collected runtime information.
5. The Scheduling Engine call XFuzzing to fuzz target application with GAMutator.

Fig 3: Procedure for our prototype system DXFuzzing

3.2 Validity and Efficiency Analysis

Based on application-specific knowledge, DXFuzzing can generate test cases that easily pass strong program checks in the program, such as program checksum, to cover f_e in F , which is hard for automatic testing using intelligent algorithms such as genetic algorithm, because some of these strong program checks are mathematically hard to pass automatically [2].

With application-specific knowledge of strong program checks and the right data sample, the complexity to cover f_e in F is largely reduced. The problem of finding new combinations to trigger possible vulnerability in f_e in F is especially suitable for genetic algorithm to solve. The fitness we have given directly relates to the vulnerability, so it is direct, and is not code coverage or nesting depth [8] which is not direct at all. Experiments in Section IV

will show that.

GAMutator does not only care about the relationships between l_i and f_e , but also cares about n_j and f_e . Because some f_e in F are influenced by the n_j , however, the n_i is neglected in general.

Different from combinatorial test in black-box testing, the combination of l_i or n_j in DXFuzzing is decided by the I/O analysis; the values of l_i or n_j in some combination are refined by every generation. DXFuzzing only selects effective, controllable combinations of l_i and n_j , and it can search the value close to most optimal solution quickly. If we set values of them randomly, the vulnerability maybe easily missed; and if we test all possible values, as in brute force testing, then combinatorial explosion will occur and the time cost is not tolerable. So, DXFuzzing is an easy, efficient and effective way, which mines more vulnerability than other tools that are only capable of one-dimensional fuzzing.

Execution-oriented I/O analysis in DXFuzzing is preferred here, because it gives more precise and effective results than static analysis. Another reason is that all relationships established by execution-oriented I/O analysis can be reached by some existing *testcases*. The third reason is that I/O analysis could be done piggyback when DXFuzzing fuzzes the target applications with m_i in M , without costing additional time.

DXFuzzing using FSTG is multi-dimension vulnerability fuzzing, compared to single-dimension fuzzing. It can mine some vulnerabilities that have never been mined by single-dimension fuzzing, because these vulnerabilities have only been triggered by special combinatorial input elements.

IV. EXPERIMENTS

For validating the effectiveness of DXFuzzing, we use the official LibPng library as the target application. LibPng is a library widely used to deal with .png (Portable Network Graphics) format file in many popular systems and applications, and its safety is a research focus. Our experimental target is libpng v 1.0.6 and the usePng.exe program that we developed to call LibPng. Insecure functions, some input nodes and insecure functions that are influenced by input nodes are shown separately in Table I, Table II and Table III.

TABLE I: INSECURE FUNCTIONS IN TARGET APPLICATION

Function name	usePng.exe	LibPng.dll v1.0.6
strcpy	1	6
memcpy	0	77
sprintf	0	16
malloc	18	113

TABLE II: INPUT NODES

ID	INPUT ELEMENTS
101	PngFile..IHDA_CHUNK_DATA.BitDepth
102	PngFile..IHDA_CHUNK_DATA.ColorType
109	PngFile..IHDA_CHUNK_DATA.Height
111	PngFile..IHDA_CHUNK_DATA.Width

TABLE III: INSECURE FUNCTIONS INFLUENCED BY INPUT NODES

ID	INSECURE FUNCTIONS
72	pngutil.c(2939):png_ptr->row_buf=(png_bytep)png_malloc(png_ptr,row_bytes)
73	pngutil.c(2945):png_ptr->prev_row=(png_bytep)png_malloc(png_ptr, png_uint_32)(png_ptr->rowbytes + 1))
89	pngread.c(1301):info_ptr->row_pointers=(png_bytepp)png_malloc(png_ptr,info_ptr->height * sizeof(png_bytep))

By static analysis on the source code, the relationships between input elements and insecure functions could be established as shown in Figure 4. By execution-oriented I/O analysis, more precise relationships were established in Figure 5.

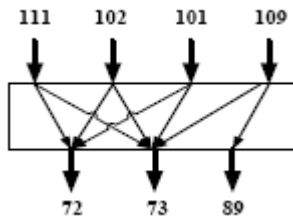


Fig 4: Relationships between inputs and insecure functions by static analysis

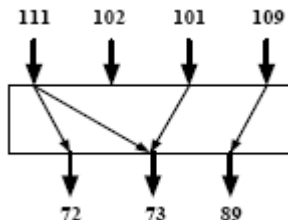


Fig 5: Relationships between inputs and outputs by dynamic execution

Take (111,101;73) for an example, function 73 is influenced by both input node 111 and input node 101. Let w denote input Width whose id is 111, d denote

BitDepth whose id is 101, z denote the actual space of `png_malloc` allocated whose id is 73. Initial values of x and y could be seen Figure 6 and $w = 0x20$, $d = 0x01$.

It is impossible find this vulnerability in function 73 if we just process one-dimension mutation on w or d based on sample file in Figure 6 even if all the test cases were generated.

```

0000h: 39 5 Width: 47 0D 0A 1A 0A BitDepth 00 0D 49 48 44 52
0010h: 00 00 00 20 00 00 00 20 00 00 00 00 00 5B 01 47
0020h: 59 00 00 00 04 67 41 4D 41 00 01 86 A0 31 E8 96
0030h: 5F 00 00 00 5B 49 44 41 54 78 9C 2D CC B1 09 03

```

Fig 6: png sample file

By multi-dimension mutation on w and d with DXFuzzing, on average this vulnerability was triggered with the cost of 400 generations and there are 60 individuals in every generation.

From RFC2083, we knew $w \in [0,0xffffffff]$, $d \in [0,0xff]$. Further analyzing, we got $d \in \{1,2,4\}$, w and d will generate $3 \times 0x100000000 = 12884901888$ combination test cases, but there are only 262148 of them that could trigger this vulnerability if we set $B=100000$ which `png_malloc` could successfully allocate memory. So the possibility is $262148/12884901888 = 0.00002$.

The distribution of $\{w,d,z\}$ that could successfully trigger this vulnerability could be seen in Figure 7.

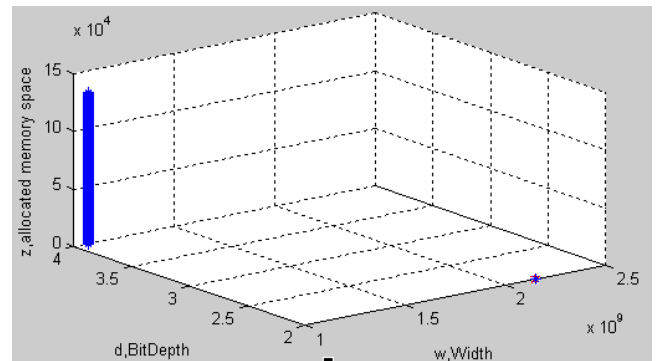


Fig 7: Width, BitDepth distribution when they trigger this vulnerability

By analyzing the source code, we could get the distribution of (w, d) that could trigger the vulnerability as follows:

$$((wd \geq (A-7)) \text{ and } (wd < A)) \text{ or } ((wd \geq A) \text{ and } (wd \% A \leq A)).$$

Its formal fitness function is:

$$f(w, d) = \begin{cases} A - wd - 7, & \text{when } wd < A - 7, \\ 0, & \text{when } wd \geq A - 7 \text{ and } wd < A, \\ 0, & \text{when } wd \geq A \text{ and } wd \% A \leq B, \\ wd \% A - B, & \text{when } wd \geq A \text{ and } wd \% A > B. \end{cases} \quad (4)$$

in which, $A = 0x\text{ffffff} + 1$, $B = 100000$ means the biggest space `png_malloc` could successfully allocate memory which is set manually. The numbers of bugs found by different tools are listed in Table IV while Peach 2.3 and DXFuzzing based on the sample file shows in Figure 6. GAMutator found 3 additional bugs and generated 3600 test cases. Smart fuzzer from paper [3] and GAFuzzing from paper [6] do not perform well in this experiment.

TABLE IV: VULNERABILITIES FOUND BY DIFFERENT FUZZING TOOLS

Tools	Number of vulnerability checked	Number of test cases
Smart Fuzzer	0	1000000
GAFuzzing	0	1000000
Peach 2.3	4	31026
DXFuzzing	7	34222

V. CONCLUSION

Whitebox fuzzing is complex and costly in time. There are still some problems such as path explosion, and is hard to pass strong program checks fully automatically. Peach is a knowledge-based fuzzing tool that is highly regarded because of its outstanding performance and ease of use and its understandability. DXFuzzing enriches current mutation methodology with multi-dimension input nodes mutation strategy without combination explosion, so DXFuzzing could find more vulnerability that never will be found by one-dimension mutation fuzzing. Paper [6] also combined fuzzing with genetic algorithm, but the fuzzing it used is simply randomly testing technique and the way it used genetic algorithm is to cover suspected vulnerable point, however this is hard to apply in practical large applications because there are many program strong program checks in them. Different from it, DXFuzzing used genetic algorithm to find interesting combination. Paper [3] also located vulnerable code like DXFuzzing, but its constraint solver cannot solve program strong program checks automatically.

VI. ACKNOWLEDGMENT

We would like to thank Hongchuan Wang and Jianjun Xia for their work and suggestions on this paper. J.W. Atwood acknowledges the support of the Natural Sciences and Engineering Research Council of Canada, through its Discovery Grants program.

REFERENCES

- [1] D. Aitel, "The advantages of block-based protocol analysis for security testing," *Immunity Inc.*, February, 2002.
- [2] P. Oehlert, "Violating assumptions with fuzzing," *IEEE Security and Privacy*, March/April 2005, pp.58-62.
- [3] L. Andrea, M. Lorenzo, M. Mattia and P. Roberto, "A smart fuzzer for x86 executables," in *Proceedings of the Third International Workshop on Software Engineering for Secure Systems: IEEE Computer Society*, 2007.
- [4] P. Godefroid, M. Levin, and D. Molnar, "Automated whitebox fuzz testing," In *NDSS*, 2008.
- [5] P. Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, and Nikolai Tillmann, "Automating Software Testing Using Program Analysis," *IEEE Software*, September/October 2008, pp. 30-37.
- [6] Liu Guang-Hong, Wu Gang, Zheng Tao, Shuai Jian-Mei, Tang Zhuo-Chun, "Vulnerability analysis for x86 executables using genetic algorithm and fuzzing," in *Third 2008 International Conference on Convergence Hybrid Information Technology (ICCIIT)*, 2008.9.
- [7] Christoph C. Michael, Gary McGraw, Michael A. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, December 2001, pp.1085-2001.
- [8] Phil McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verif. Reliab.* 2004; 14:105-156.
- [9] Concettina Del Grosso, Giuliano Antoniol, Massimiliano Di Penta, Philippe Galinier, Ettore Merlo, "Improving network applications security: a new heuristic to generate stress testing data," in *GECCO'05*, June 2005.
- [10] P.J. Schroeder, B. Korel, "Black-box test reduction using I/O analysis," in *Proceedings of the International Symposium on Software and Analysis (ISSTA '00)*, Portland, Oregon, August 2000.
- [11] C. Luk, R. Cohn, R. Muth, et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI2005)*, pages 190-200, 2007.
- [12] B. P. Miller, L. Fredrikson, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Comm. of the ACM*, vol. 33, no. 12, p. 32, December 1990.



- [13] T. Boutell, et al., “PNG (Portable Network Graphics) Specification”, Version 1.0, IETF Request for Comments 2083.