Test Case Generation for Vulnerability Detection Using Genetic Algorithm

Bo Shuai, Haifeng Li, Jian Wang, Quan Zhang and Chaojing Tang

School of Electronic Science and Engineering, National University of Defense Technology, Changsha, Hunan, P. R. China, 410073

shuaibo85@163.com

Keywords: genetic algorithm; fuzz; path coverage; test cost

Abstract. In order to elevate efficiency of traditional Fuzzing technique, a novel method using genetic algorithm is proposed based on path coverage and test cost. There are evidences that GA has been already successful in generating test cases. Considering path coverage as the test adequacy criterion, we have designed a GA-based test data generator that is able to synthesize multiple test data to cover multiple target paths. Meanwhile, in order to reduce the test cost in Fuzzing process, test cost is analyzed respectively from running time and loop structure in the method. Experimental results show that proposed approach could obtain higher vulnerability detection accuracy and efficiency.

Introduction

Detection of software vulnerability is very important for software quality and security. Among software testing techniques, Fuzzing [1] has proven successful in finding security vulnerabilities in large software. A number of severe software vulnerabilities have been revealed by Fuzzing techniques [2]. However, traditional Fuzzing testing fails to satisfy syntactic or semantic constraints and therefore cannot exercise deeper code [3]. In order to improve the traditional Fuzzing vulnerability detection techniques, a feasible method is proposed using genetic algorithm based on path coverage and test cost, which could find more potential security vulnerabilities for binary executable software.

Due to ignoring the logical context relationship of software, the random Fuzzing vulnerability detection technology has some unique advantages and characteristics such as high degree of automation, low computational cost and low false alarm rate. However, test case generation process is completely random, which results that many test cases often point to the same one execution path. The lack of guidance directly leads the lower execution path coverage and low detection efficiency [4].

Genetic algorithm first proposed by Holland [5], is usually used to calculate the ex-act solution or the approximate solution for the optimization and search problems. Genetic algorithm has unique advantages in solving large space, highly complex non-linear problems [6]. Introducing genetic algorithm into the software test case generation process could result a faster convergence to the target solution, the fitness function can also make the process of generating test cases more oriented and efficiency. Therefore, the research based on path coverage and test cost is studied in order to generate the effective test case to cover the potential vulnerabilities execution path and reduce the actual test cost to enhance the efficiency of test case.

Framework

The framework of the method is shown in Figure. 1. Firstly, the binary executable software is tested through static analysis. Secondly, according to the static analysis results, specific information of path coverage is calculated. Where, test path diversity is achieved based on the sequence alignment algorithm. Meanwhile, test cost is also disposed respectively from running time and loop structure. Where, the loop structure is simplified using finite expansion manner. Thirdly, the genetic algorithm fitness function based on the code coverage and test cost is constructed to guide the test case generation. Genetic algorithm continues until the test cases triggering potential security vulnerabilities in binary executable software.

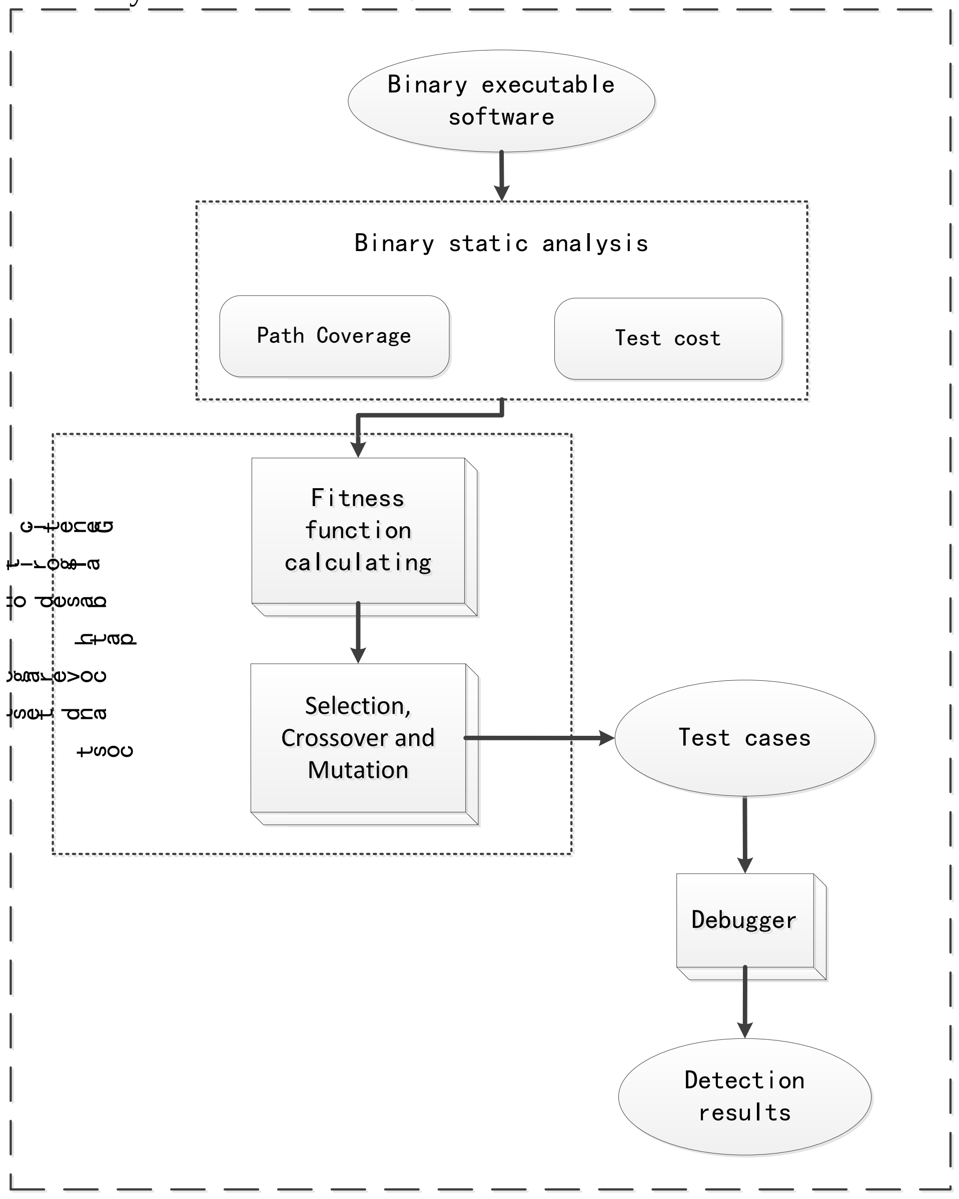
Binary executable software
Binary static analysis
Path Coverage
Test cost
Fitness function calculating
Selection, Crossover and Mutation
Test cases
Debugger
Detection results

Figure. 1 Framework of the proposed method

Issue Analysis of Path Coverage and Test Cost

Path Coverage. As has been demonstrated, many GA-based test data generators adopt statement or branch coverage as their objectives. However, by nature, path coverage criterion covers statement and branch coverage criteria; thus, a more effective software structural testing should have path coverage as its objective [7, 8].

As the previous section concluded, none of the works on satisfying path coverage focuses satisfying multiple target paths at a time, i.e. covering a set of target paths in a single run of GA. Clearly, covering multiple paths in one run would require incorporating these paths within the fitness calculation. The rationale behind considering multiple paths in one run is based on the observation that while trying to cover a single path, some of the individuals generated cover other paths as a by-product. Accordingly, trying to satisfy multiple paths at a time is expected to greatly increase the effectiveness and efficiency of the test data generator, i.e. attaining more coverage with fewer resources than a single-path test generator that would need multiple runs to cover the same number of paths.

Test cost. Currently, the dynamic test generation techniques are based on two basic assumptions:

a) Test case has a certain degree of test coverage and the test coverage remained basically unchanged during each running of the program.

b) Assume that each test case has the same test cost such as running time, and the test cost remained basically unchanged during each running of the program either.

However, in the actual testing process, the test cost of each test case are different, especially the running time can't be exactly the same. In order to find a set of test cases with lowest test cost rather than the least number, the test running time must be taken into consideration, which is an important way to truly improve the efficiency of test cases.

Loop is a widely used program structure in programming language. Loop structure associates closely with the vulnerability in two aspects: the input variable validation and the continuous access and assignment of memory address. Incorrect configuration of the loop termination condition is most likely to cause an abnormal memory operation, which is the main reason of buffer overflow and array cross-border access vulnerability. One effective way to dispose the path explosion problem is to expand the loop structure in finite times. This paper replaces the loop structure with a certain number expansion conditional branch instruments. The expanded depth equals to the maximum execution number of loop structure.

Genetic Algorithm Using Path Coverage and Test Cost

Basic Theory of GA. To avoid the blindness of random Fuzzing technology in test case generation, the genetic algorithm is introduced to enhance the guidance. A novel fitness function based on critical path information is constructed in order to hit the vulnerabilities in the binary program more efficiently. The genetic algorithm is proposed in 1975 by the Holland J, which simulates biological evolution and natural selection process through selection, crossover and mutation mechanism to improve the fitness of the individual. Genetic algorithm is a global optimization algorithm, its main features are:

a) Utilizing probabilistic optimization method to automate access to optimized search space, without the use of any operating rules established, it can adaptively adjust the search direction;

b) Operating directly on the structure of objects, there is no limited condition in continuity and derivative function;

c) With the ability of global optimization and implicit parallelism. These character-istics of the genetic algorithm has been has been widely used in adaptive control, signal processing, machine learning, artificial life and portfolio optimization and other fields [9].

Design of Fitness Function. Fitness function plays the role of the only interface between genetic algorithm and specific applications, which determines the evolutionary direction. A good fitness function could guide the whole evolutionary process faster to optimal convergence of the solution space. The purpose of vulnerability detection is to trigger the vulnerabilities as many as possible, thus a feasible fitness function for vulnerability detection should be able to guide the genetic algorithm to generate test cases that are most likely to trigger the vulnerability.

A building block is a constituent of the fitness function of consideration of path coverage. The constituents of a fitness function affect its effectiveness/efficiency in directing the search toward the desired goal. The basic building blocks of our pro-posed fitness function candidates are based on comparing traversed paths to target paths in terms of distance D and violation V. D is calculated as the difference between the traversed path and the target path, in term of predicate values for the "unmatched node-branches". For example, consider the predicate "A < B", and assume that this predicate should be "false" within a target path. D will be calculated as ABS(B − A) if A is actually less than B in a given traversed path. V tells how many unmatched nodes exist between the target path and the traversed one. It is worth noting here that $D \geqslant 0$ and $V \geqslant 0$. A node is a branching predicate, i.e. a statement where the program is heading to different branches logically. For example, an IF-THEN-ELSE statement is a node that has two logical branches, i.e. THEN branch and ELSE branch. The objective is to minimize the distance D and violation V. The effectiveness of building blocks of path coverage is shown in (1).

Where, i is index of target path; j is index of individual; k is index of node in both target path i and traversed path j; and IF is intermediate fitness that looks at the fitness of an individual with respect to one target path. IF is considered as a building block for the overall individual fitness where the fitness with respect to all target paths is considered.
Where, C(n) is the running time ratio, which means the percentage of test running time of the current called test case. Based on the above analysis, the fitness function problem is transformed into search for the test cases that covers the highest path coverage with the least test cost.

Construction of GA Operators. There exist three GA Operators, namely selection operator, crossover operator and mutation operator. Firstly, selection operator is constructed. In traditional genetic algorithm, the use of roulette method is used as selection operator. In order to improve the probability of high fitness individual being selected, the accelerated selection method is introduced, which directly copies the maximum fitness individual into the next generation. Secondly, crossover operator is constructed. The proposed method is for stand-alone binary executable software. First get the length of the input file denoted as file length. Secondly randomly generate two integers between 1 and filelength-1. Make the two integers as the cross intersection relative to the input file header and interchange the input file byte fragment between the two cross intersection. Thirdly, mutation operator is constructed. First randomly generate another two integers between 1 and filelength-1 as the mutation point. Then generate two random hexadecimal digits between 0 to 0xFF as the mutation values. Replace the value of the mutation point with the mutation values.

Experiments and Results

Experiment Environment. The experiment condition contained a test server which was HP DL980G7, with 4 x Intel E7-4870 CPU, single CPU @ 2.5Ghz, 10 cores per CPU, 128G memory. The IDAPro software [10] was used to complete the static analysis of the binary executable software. The experiment of genetic algorithm was based on C + + Genetic Algorithm Development Kit [11] in Visual Studio 2012.

Results and Comparisons. Three known software applications were selected for testing, including multimedia player and format conversion application, such as Winamp.exe, MP3 CD Ripper.exe and Storm.exe. Because the prototype system was built on the Windows platform, the selected binary executable software was all the Windows applications. After about two days of vulnerability detection, three exceptions were triggered in the test software, which have been verified as the public high-risk vulnerabilities. The specific detection results are shown in Table 1.

Table 1. Specific detection results of testing softwares

No.	Binary software	Version	Serial number of public vulnerability	Number of exceptions
1	Winamp.exe	5.5.7.2830	CVE-2010-4371	1
2	MP3 CD Ripper.exe	2.6	EDB-17727	1
3	Storm.exe	2012 3.10.4.21	CNVD-2010-0752	1

In order to verify the efficiency of this method, the proposed method is compared to the open source tool zzuf [12] based on random Fuzzing technology. In the condition of triggering one exception, the comparison results are shown in Table 2.

Table 2. Comparison results between the proposed method and random fuzzing

Method		Winamp.exe	MP3 CD Ripper.exe	Storm.exe
Running time（s）	Proposed approach	79,192	162,283	89,784
	zzuf	1,036,296	>1,728,000	838,056

As can be seen, from the point of view of running time, the average detection efficiency of the proposed method is more effective that of the random Fuzzing technology. Thus, compared with the random Fuzzing method, a more efficient method for binary executable software vulnerability detection was presented.

Conclusion

A novel vulnerability detection method for binary software based on path coverage and test cost is proposed. As can be seen from the experimental results, the proposed method could achieve much more efficient binary executable software vulnerability detection results compared with the random Fuzzing technology. Path coverage especially test path coverage and test cost such as running time and loop structure are both very significant in the vulnerability detection process. Meanwhile, the genetic algorithm is considerable effective in improving guidance of test case generation. The future work, on the one hand is to refine the integrality conditions of test path to better address the path explosion problem. On the other hand, the network application vulnerability detection is more and more important, how to adapt the method to the network application will be our future task.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China Grant No. 61101073.

References

[1] Sutton M, Greene A, Amini P. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional (2007)

[2] Oehlert P. Violating assumptions with fuzzing. IEEE Security and Privacy, vol. 3, pp. 58-62 (2005)

[3] Clarke T. Fuzzing for software vulnerability discovery. Department of Mathematic, Royal Holloway, University of London (2009)

[4] Greg B, Marco C, Viktoria F. SNOOZE: toward a stateful network protocol fUzzer. In proceedings of the 9th information security conference, LNCS, vol.1, pp. 343-358, 2006.

[5] John H. Adaptation in Natural and Artificial System. The MIT Press (1992)

[6] David E G. Genetic algorithms in search, optimization and machine learning. Addison-Wesley Publishing Company (1989)

[7] Sthamer HH. The automatic generation of software test data using genetic algorithms. PhD dissertation, University of Glamorgan; November (1995)

[8] Sthamer HH, Wegener J, Baresel A. Using evolutionary testing to improve efficiency and quality in software testing. In: Proceedings of the second Asia-Pacific conference on software testing analysis and review, Melbourne, Australia; 22–24th (2002)

[9] Abdelhamid B. An Immune Genetic Algorithm for Software Test Data Generation. In proceedings of the 7th international conference on hybrid intelligent systems, vol. 3, pp. 84-89 (2007)

[10]Chris E. The IDA Pro Book. USA: No Starch Press (2011)

[11]Ohsumi T K, Borowsky M L. MolBioLib: a C++11 framework for rapid development and deployment of bioinformatics tasks. Bioinformatics, vol.28(19), pp. 2412-2416 (2012)

[12]David M, LI Xue-cong, David A W. Dynamic test generation to find integer bugs in x86 binary linux programs. In proceedings of the 18th conference on USENIX security symposium, vol. 1, pp. 68-72 (2009)

[13]Kim H C. Efficient file fuzz testing using automated analysis of binary file format. Journal of Systems Architecture, vol. 57(3), pp. 259-268 (2011)

[14]Abdallah N. Performance impact of fuzz testing windows embedded handheld applications. Proceedings of the 14th International Conference on Enterprise Information Systems, vol. 2, pp. 371-376 (2012)

[15]Tsankov P, Mohammad T, Basin D. Semi-valid input coverage for fuzz testing. International Symposium on Software Testing and Analysis, vol. 3, pp. 55-56 (2013)

[16]Schneider M, Grossmann J. Online model-based behavioral fuzzing. IEEE 6th International Conference on Software Testing, Verification and Validation Workshops, vol. 2, pp. 469-475 (2013)

[17]QI Lan-lan, WEN Jiang-tao, HUANG Hui, The research on the fuzzing. Lecture Notes in Electrical Engineering, vol. 255, pp. 85-91 (2013)