H-Fuzzing: A New Heuristic Method for Fuzzing Data Generation

Jinjing Zhao[1,2], Yan Wen[1,2], Gang Zhao[1,2]

[1] Beijing Institute of System Engineerring,
Beijing, China

[2] National Key Laboratory of Science and Technology on Information System Security,
Beijing, China

misszhaojinjing@sina.com.cn, celestialwy@gmail.com, zg@public.bise.ac.cn

Abstract. How to efficiently reduce the fuzzing data scale while assuring high fuzzing veracity and vulnerability coverage is a pivotal issue in program fuzz test. This paper proposes a new heuristic method for fuzzing data generation named with H-Fuzzing. H-Fuzzing achieves a high program execution path coverage by retrieving the static information and dynamic property from the program. Our experiments evaluate H-Fuzzing, Java Path Finder (JPF) and random fuzzing method. The evaluation results demonstrate that H-Fuzzing can use fewer iterations and testing time to reach more test path coverage compared with the other two methods.

Keywords: Fuzzing test, static analysis, dynamic analysis, program slicing, control flow graph, program security testing.

1 Introduction

Fuzzing, according to its basic definition, might be characterized as a blind fishing expedition that aims at uncovering completely unsuspected problems in the software. If the program contains a code slice that may lead to exceptions, crash or errors, it can be determined that a vulnerability has been discovered. Generally, fuzzers are good at finding buffer overflow, DoS, SQL Injection, XSS, and Format String bugs, but always suffer the difficulty of finding vulnerabilities that does not cause program to crash, such as information disclosure, encryption flaws and so on.

Because of its random nature, the fuzzing data space must be huge enough to achieve high veracity and vulnerability coverage. For many applications, injecting random bits is almost infeasible. Consequently, completely random fuzzing is a comparatively ineffective way to uncover problems in an application.

To address this limitation, this paper presents a new heuristic fuzzing data generation method, namely H-Fuzzing. H-Fuzzing collects the information of key branch predictions and builds its relations with the program input variables. Besides,

H-Fuzzing supervises how the fuzzing data space shrinks with the branch predictions and input variables. By abstracting these static information and dynamic property from the analyzed program, it accomplishes high program execution path coverage.

The remainder of this paper is organized as follows. Section 2 is a brief introduction of the related work. Before the description of our method, the program model is built in section 3. The details of H-Fuzzing is described in section 4 and an experimental evaluation is presented in section 5. Finally, we conclude our work with a brief summary and discussion of open problems in section 6.

2 Related Work

The term fuzzing is derived from the fuzz utility [1], which is a random character generator for testing applications by injecting random data at their interfaces [2]. In this narrow sense, fuzzing just means injecting noise at program interfaces. For example, one might intercept system calls made by the application while reading a file and make it appear as though the file containing random bytes. The idea is to look for interesting program behavior that results from noise injection. Such behavior may indicate the presence of vulnerability or other software fault.

A simple technique for automated test generations is random testing [3-8].In random testing, the tested program is simply executed with randomly-generated inputs. A key advantage of random testing is that it scales well in the sense that generating random test input takes negligible time. However, random testing is extremely unlikely to expose all possible behaviors of a program. For instance, the "then" branch of the conditional statement "if (x= =10) then" has only one in 2^{32} chances of being executed if x is a randomly chosen 32-bit input variable. This intuitively explains why random testing usually provides low code coverage.

Several symbolic techniques for automated test generation [9-13] have been proposed to ameliorate the limitations of manual and random testing. Grammar-based techniques [14, 15] have recently been presented to generate complex inputs for software systems. However, these techniques require a grammar to be given for generating the tested program's input, which may not always be feasible.

There are several other works including some more intelligent techniques. For example, fuzzing tools are aware of commonly used Internet protocols, so that testers can selectively choose which parts of the data will be fuzzed. These tools also generally let testers specify the format of test data. This is very useful for applications that do not use the standard protocols. These features overcome the limitations discussed in the previous paragraph. In addition, fuzzing tools often let the tester systematically explore the input space. Such tester might be able to specify a range of input variables instead of having to rely on randomly generated inputs.

To sum up, existing traditional fuzzing ways suffer the following deficiencies:

1) They lack general fuzzers because they have to put focus on special objects to reduce the complexity and scale of fuzzing data generation.

2) The random methods suffer poor test efficiency.

3) They only have a low execution path coverage rate, which denotes the execution path number divided by the total number of branches in the program.

3 Program Model

A program P in a simple imperative programming language consists of a set of functions $F = \{f_1, f_2, ..., f_n\}$, one of which is distinguished as *main*, i.e., the program execution entry function. Each function f_i is denoted as { $Entry_i$, $Input_i$, $Exit_i$ }, wherein $Entry_i$ is the function executing entrance, $Input_i = \{I_{i1}, I_{i2}, ..., I_{im}\}$ is the function input set, and $Exit_i = \{E_{i1}, E_{i2}, ..., E_{it}\}$ is the set of function return points. The function f_i is executed as a call, namely *call*(f_i). In its body, $m := e$ means assigning the value e to the memory location m, an expression free of side effects and a conditional *if p then goto l*, wherein l is the label of another statement in the same function and p is a predicate free of side effects.

The execution of program P with inputs $Input_P$ which the customers give proceeds through a sequence of labeled program statements $p_0; ...; p_k$, with $p_0 = l_{main;0} : Entry_{main}$, the first statement of the main function.

The program P consisting of functions f_1, ..., and f_n. Its combined control flow and static call graph $CFCG_P$ is a directed graph whose vertices are the statements of P. The edges of $CFCG_P$ begin from each statement $l_{i;j} : s_{i;j}$ and end at its immediate successors.

$$\begin{cases} none & if\ s_{i,j} = Exit_{f_i} \\ l' \text{ and } l_{i,j+1} & if\ \ s_{i,j} = if\ \ p\ goto\ l' \\ Entry_{f_i} \text{ and } l_{i,j+1} & if\ \ \ s_{i,j} = call(f_i) \\ l_{i,j+1} & \text{otherwise} \end{cases} \quad (1)$$

To describe the rest of this paper more clearly, the following definitions are proposed.

Definition 1 (Control Flow Graph, CFG) The control flow graph G denotes a directed graph. Each of its nodes is a basic module with only one entry and exit, symbolized with *Entry* and *Exit* correspondingly. If the control flow can reach the basic module B from the basic module A directly, there is a directed edge from node A to node B. Formally, the control flow graph of program P can be represented with a quadruple *G (IV, E, Entry, Exit)*. *IV* is the node set which symbolizes the basic modules. E denotes the edge set. Each edge is symbolized by an ordered couple $<n_i, n_j>$ which represents a possible control transition from n_i to n_j (n_j may be executed just after n_i has been executed).

Definition 2 (Branch Path Tree, BPT) The branch path tree of program P, namely T_f, is the set of all the possible execution paths of P. Each node T_f is the set of a branch node along with all the non-branch nodes between this branch node and previous branch node, or the *Exit* of P. The root of T_f denotes the initial state of P.

Definition 3 With the denotation that l represents any node of a control flow graph *CFG*, we can reach the following definitions.

a) ***The Definitions Set***, *Def(l)* = { x| x is a variable changed in the sentence l }.

b) ***The References Set***, *Ref(l)* = { x| x is a variable referred in the sentence l }.

Definition 4 (Data Dependence) If node n and m satisfy the following two conditions, n is data-dependent on m.

a) There is a variable v, and v belongs to $Def(m) \cap Ref(n)$.
b) There is a path p in G from m to n, and for any node $m' \in p - \{ m , n\} , v | def(m')$.

Definition* 5 *(Control Dependence) If node n and m meet the following two conditions, n is control-dependent on m.

a) There is a path p in G from m to n, and for any node $m' \in p - \{ m , n\}$, n is the successive dominator of m'.
b) n is not the successive dominator of m.

Definition* 6 *(Program Slicing) A slice S of a program is an executable sub-program. For a variable v located in some interest point l (l and v is termed as the Slicing Rules), S is composed of all the sentences which may influence v in l. In the functionality point of view, the slice S is equivalent with P. The so-called influencing v refers to having data-dependence or control-dependence on v.

Slicing a program is supposed to follow some slicing rules. While slicing the same program, the slices worked out will differ with the selected slicing rules. A dynamic slicing rule is represented with an ordered triple $<Input_p, S_k, v>$, wherein $Input_p$ denotes the input set of a program. S is a sentence of the program, and then S_k indicates that the sentence S is executed at step k. It can also be symbolized with $S:|k$. v denotes a variable. It can represent a single variable or a subset of the program's variable set.

4 Heuristic Program Fuzzing Data Generation Method

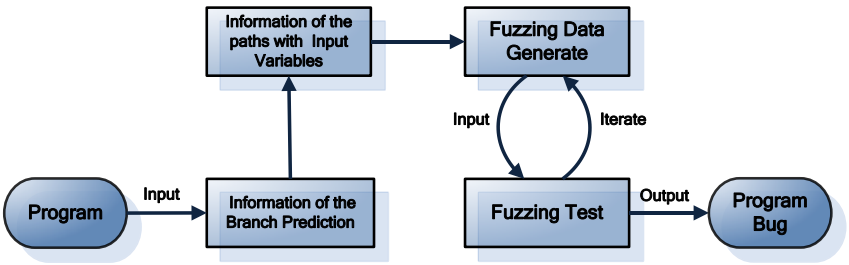


Fig. 1. The Work Flow Chart in H-Fuzzing Method.

H-Fuzzing is composed of two processes: static analysis process and fuzzing process. Fig. 1 illustrates the working flow of H-Fuzzing. Firstly, the information of all branch predictions and possible execution paths is collected, especially their relations with the input variables. Secondly, an initial input is generated, and then a new path will be chosen to be analyzed next time according to the execution path selecting rules. With the information supervised in the static analysis process, a new input variable set will be generated to run the program continually. If it works, the fuzzing process will get the next path until all the paths in the program are covered. Otherwise, the process will iterate the input variable again.

4.1 The Program Static Analysis Procedure

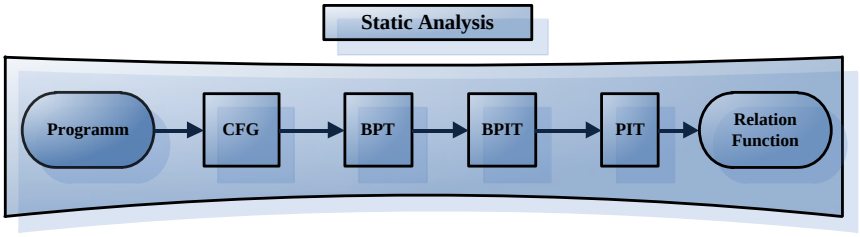


Fig. 2. The Output Information of The Static Analysis.

The static analysis is responsible for analyzing the relations between the possible execution paths and the input variables. The program tested here can be source code, inter-procedure language or even Java byte code. The working steps of the static analysis are listed as follows, as shown in Fig.2.

1) Construct the CFG of the fuzzed program;
2) Build the branch path tree (BPT) of the program with the CFG information;
3) Slice the program with every branch prediction variable in branch lines, and record its data & control dependence information, especially with the input variables.
4) Deduce the relation functions between execution paths and input variables. All such information is organized with a table, called Branch Prediction Information Table (BPIT).
5) According to the BPT and BPIT, deduce the information of every possible path in the fuzzed program, and work out a Path Information Table (PIT) which records the relations between execution paths and input variables.

One entry in BPIT includes four elements: { *Line Number, Branch Predication, Corresponding Input, Relation Function*}. *Line Number* is line no. of the branch sentence. *Branch Predication* is the variables which make the branch be executed. *Corresponding Input* is the input variable set which has effect on the branch prediction. *Relation Function* defines the relation function between the branch prediction and the input variable set.

One entry in PIT is composed of five elements: *{ID, Path, Branch Point, Corresponding Input, Relation Function}*. *ID* is the exclusive number of path. *Path* is the set of line numbers of the path. *Branch Point* contains all the line numbers of branch sentences. *Corresponding Input* is the input variable set which has effect on this path. *Relation Function* defines the relation function between the path branches and the input variable set.

Because the generation algorithms of the program CFG and the BPT have been fully studied and lots of related tools are available, they would not be discussed here.

The process of calculating the prediction slicing is listed as follows: Firstly, H-Fuzzing finds all the nodes which the prediction variable *v* in the branch sentence *S* has data or control dependence on directly. Secondly, it keeps finding the nodes which new nodes have data or control dependence on directly, until no new nodes is added. Finally, it parallels these nodes according to the sequence in the program sentences. Then, the slice of the variable v in the branch sentence S is generated.

```
Algorithm "H-Fuzzing Static Analysis Procedure"
Input: P
Output: PIT
Begin
    Begin at the Entry_main in P, add every object to Object[] in P
    While Object[] ≠ ϕ {
        For o_i ∈Object[]{
10:         add every function to Function_i [] in o_i;
            While Function_i [] ≠ ϕ {
20:             For f_j ∈Function_i []{
                    Build CFG_fj ;
                    Build BPT_fi;
                    For each branch node Bn_t ∈BPT_fj {
                Get the information of its { Line Number, Branch Predication, Corresponding Input, Relation
                Function};  }
                    Build BPIT_fi;
                    For each path Pt_l ∈BPT_fj {
                Get the information of its { ID, Path, Branch Point, Corresponding Input, Relation Function }; }
                Build PIT_fi;  }
                    Find the related function f_m to f_j;
                    Get their exchanged information;
                    f_j = f_m ;
                Delete f_m from Function_i [];
                Goto 20;
            }
            Find the related object o_n to o_i;
            Get their exchanged information;
            o_i =o_n ;
            Delete o_n from Object[];
            Goto 10;
        }
    }
End
```

Fig. 3. H-Fuzzing Static Analysis Procedure.

In the concrete analyzing process, H-Fuzzing introduces the hierarchical decomposing method. H-Fuzzing need to separate the program into data layer, function layer, object layer and class layer, abstract the hierarchical slicing models, and figure out all kinds of dependence relations between them. Following that, H-Fuzzing builds the CFGs and BPTs in different layers, and then utilize the prediction slicing algorithm to calculate different granularity slices in every layer from the top level to the bottom. When the slicing process is working among layers, the escalating algorithm is adopted. This concrete static analysis algorithm is described in Fig. 3.

4.2 The Fuzzing Data Generation Algorithm

The fuzzing data generation process of H-Fuzzing is listed as follows. H-Fuzzing firstly constructs an initial input in a random or manual way. Then, it records the execution pathes and adjusts one or more input variables following the relation functions. In this way, a new input will be generated and make the tested function execute the specified path iteratively. H-Fuzzing will repeat the above process until the whole paths have been executed.

The branch prediction includes the following symbols.

Relation symbols: “>”, “<”, “==”, “>=”, “<=”, “≠”.

Operator symbols: “+”, “—”, “ *”, “ / ”.

Conjoint symbols: “&”, “||”.

The branch predictions can be categorized into several types, such as atom predictions, twice-dimension predictions, triple-dimension predictions, and so on.

The input variable generation algorithm of the atom predictions is listed as follows.

(1) If “nl+n2+…+ni>m” , then nl =N, n2=N+1, ...,ni =N+i-1, m=-1+i(2N+i-1)/2.

(2) If “nl+n2+…+ni <m”, then nl =N, n2=N+1, ...,ni =N+i-1, m=1+i(2N+i-1)/2.

(3) If “nl+n2+…+ni =m”, then nl =N, n2=N+1, ...,ni =N+i-1, m= i(2N+i-1)/2.

(4) If “nl+n2+…+ni≠m”, then nl =N, n2=N+1, ...,ni =N+i-1, m ≠ i(2N+i-1)/2.

The input variable generation algorithm of twice-dimension predictions is illustarted as follows.

(1) “&” conjunction

① If “a>(<, =, >=, <=)N1 & b>(<, =, >=, <=)N2 & a+(-, *,/)b>(<, =, >=, <=)N3”, then H-Fuzzing takes a, b for the axis and choose the points surrounded by three lines to establish dimensional rectangular coordinate system.

② If “a≠ N & b≠ M”, then H-Fuzzing takes a, b for the axis and choose the points outside the two lines to establish dimensional rectangular coordinate system.

(2) “||” conjunction

① If “a>(<、=、>=、<=)b || c>(<、=、>=、<=)d”, then generate three set values, i.e., (a=N+l, b=N, c=M, d=M+l), (a=N, b=N+1, c=M+1, d=M), and (a=N+1, b=N, c=M+1, d=M).

② If “a≠b || c≠ d”, then generate three set values, i.e., (a≠N, b=N, c=M, d=M), (a=N, b=N, c≠M, d=M), and (a≠N, b=N, c≠M, d=M).

The search method for more complex three-dimensional predicated coverage test is shown as follows.

(1) “&” conjunction (“≠”conjunction is not considered).

① Reduce inequalities to linearly independent inequalities.

② Figure out the critical values for the one-dimensional expression variables, in accordance with the direction of the critical value inequality +1 or -1 (eg, “a> N” will take “a = N + l").

③ Adjust another variable conditioning variables included in the two-dimensional variables inequalities, which have been identified, to satisfy the two-dimensional variable inequality.

④ Adjust the value of third variable to meet the three-dimensional variable inequality, regulate the value of the fourth variable to meet the four-dimensional variable inequality ..., and finally adjust the value of i-variables to satisfy the N-dimensional variable inequality.

(2) “||” conjunction (“≠”conjunction is not considered).

The values are that satisfying the i-th expression accordance with the “||” conjunction input variable generation method of the two-dimensional complex predicate coverage

(3) Conjunctions contain the relation symbol “≠”

Following the above two-step search algorithm, H-Fuzzing can puzzle out any value, which satisfies the expression, and the other variable values.

In order to improve the efficiency of fuzzing data generation algorithm, H-Fuzzing defines the following rules.

Rule 1. Maximum Iteration Times Rule: The search will terminate into failure if it runs out of branches to select, exhausts its budget of test iterations, or uncovers no new branches after some set number of iterations.

Rule 2. The Minimum Variable Changing Rule: If one branch prediction is influenced by several input variables, H-Fuzzing will change the numbers of inputs as least as possible during the fuzzing data reducing process.

Rule 3. DFS and Nearest Rule: While choosing the next execution path, H-Fuzzing will follow the depth first order and the path nearest to the current execution path because they have the most same code.

Algorithm "**H-Fuzzing Fuzzing Procedure"**
Input: P,$\{x_{init},y_{init}\}$,PIT,Al_{max}
Output: Bug
Begin
Execute the program P with input $\{x_{init},y_{init}\}$;
According to the Rule 3, find the next $path_i$ in PIT;
10: **for (***j***=** Al_{max};0;*j*--) {
Find the Branch Point set $BP_i[]$ in $path_i$;
Find Corresponding Input fset $In_i[]$ in $path_i$;
Generate $\{x_i,y_i\}$ according to the Relation Function $Rf()$ in $path_i$;
Execute the program P with input $\{x_i,y_i\}$;
if (the path follows $path_l$){
if ((Bug found) or ($path_l$= ϕ)) return (Bug) ; **else** Break; } }
Printf("Could find the inputs to execute s%", $path_i$.path);
find the next $path_l$ in PIT;
if (($path_l$= ϕ)) return (0) ;
else{ $path_{i=}$ $path_l$; goto 10; }
End

Fig. 4. H-Fuzzing Fuzzing Procedure.

5 Experimental Evaluation

Table 1. The Key Attributes of The Tested Programs.

	Jar File Size	Code Line	Class Count	Function Count
JLex	50	7874	21	146
SAT4J	428	19641	120	1056
JCL2	56984	3586	35	179
JLine	91183	5636	36	324
checkstyle	627333	49029	328	1982

We have implemented H-Fuzzing based on the official Java platform, viz., OpenJDK. In this section, we will evaluate H-Fuzzing and compare it with JPF and random fuzzing method to demonstrate its effectiveness. The testbed is equipped with an Intel i7 920 processor, 4G RAM, Windows XP SP3 operating system, and OpenJDK

(version b71). We test five open source Java projects, exploring JLex, SAT4J, Java Class Loader 2 (JCL2), JLine and checkstyle. Table 1 lists the key attributes of their source code.

In our experiment, the time for each test is limited within one hour. The evaluation makes statistic on the input file numbers and the paths covered by H-Fuzzing, JPF and random fuzzing method. The evaluation results demonstrate that H-Fuzzing can use fewer fuzzing iterations and testing time to reach higher test path coverage than the other two methods.

Due to the page limitation, the static analysis results are only presented for JLex. For the other programs, just the evaluation results (the path coverage comparisons) are illustrated.

- **JLex**

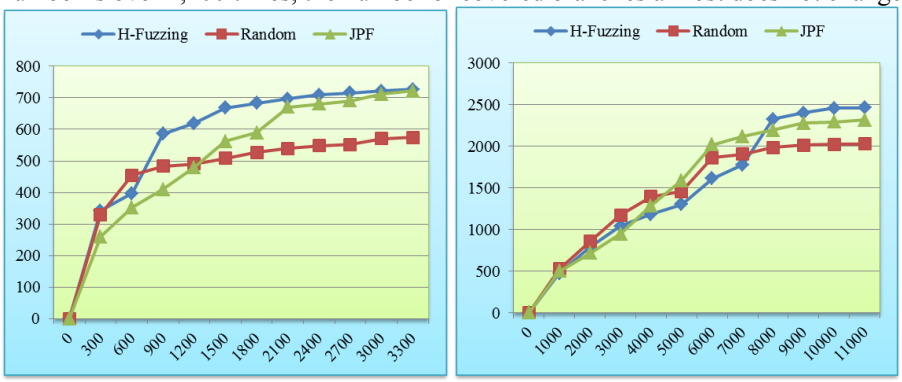
JLex is a Lex file parser generator. Its input is just the path of a Lex file. The static analysis process of H-Fuzzing analyzes 162 functions of JLex, however there are only 146 methods in its source code. This is because that some functions will be added by the Java compiler during the compiling procedure, such as class's default constructor, destructor and so on. The static analysis process also generates the control flow graph and pollution spreading graph, which contains the Java intermediate language expressions of the branch nodes dependent on the input variables. Fig. 5 is the static decompiled results of the main function of JLex. As shown in this figure, the statement, tagged with [3], is the branch statement dependent on the input variables.

```
public class JLex.Main extends java.lang.Object
{
    ......
    public static void main(java.lang.String[]) throws java.io.IOException
    {
        java.lang.String[] r0;          JLex.CLexGen r1, $r4;
        java.lang.Error r2, $r6;        int $i0;
        java.io.PrintStream $r3, $r7; java.lang.String $r5, $r8;
        [1] r0 := @parameter0;
        [2] $i0 = lengthof r0;
        [3] if $i0 >= 1 goto label0;
        [4] $r3 = java.lang.System.out;
        [5] $r3.println("Usage: JLex.Main <filename>");
        [6] return;
        [7]     label0:         $r4 = new JLex.CLexGen;
        [8] $r5 = r0[0];
        [9] specialinvoke $r4.<init>($r5);
        [10] r1 = $r4;
        [11] r1.generate();
        [12]    label1:         goto label3;
        [13]    label2:         $r6 := @caughtexception;
        [14] r2 = $r6;
        [15] $r7 = java.lang.System.out;
        [16] $r8 = r2.getMessage();
        [17] $r7.println($r8);
        [18]    label3:         return;
        [19] catch java.lang.Error from label0 to label1 with label2;
    }
}
```

Fig. 5. Static Decompiled Results of The Main Function of JLex.

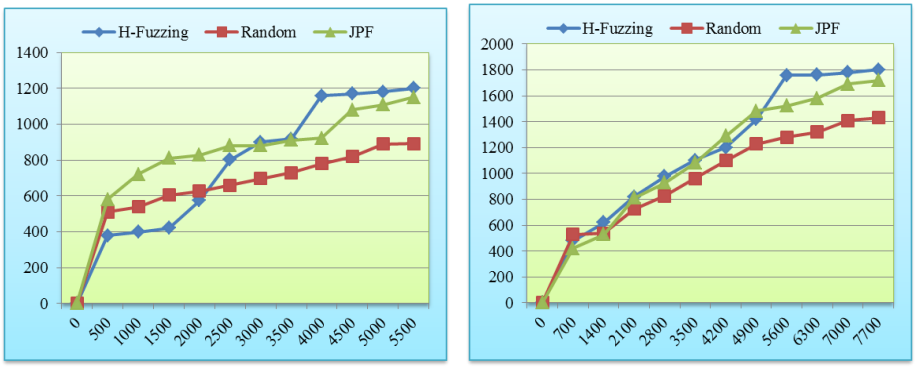
H-Fuzzing constructs the heuristic fuzz data referring to both the control flow graph and the static decompiled results. The static analysis shows there are 775

branch statements in its source code. Fig. 6 (a) is the evaluation results of H-Fuzzing, JPF and random fuzzing method. The horizontal axis is the number of fuzzing iterations. The vertical axis is the number of covered branches. When the iteration number is over 2,100 times, the number of covered branches almost does not change.

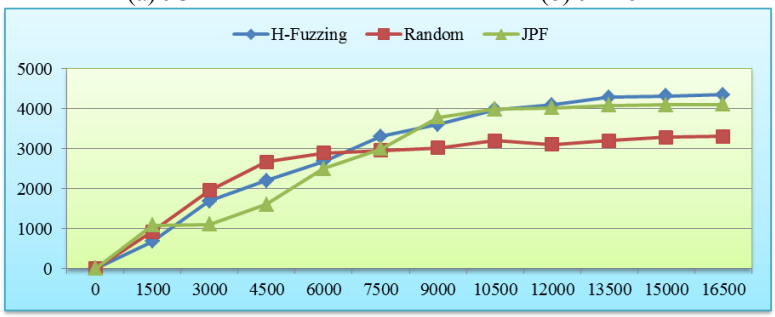


(a) JLex (b) SAT4J

Fig. 6. Trend Graph of JLex & SAT4J Branch Coverage.



(a) JCL2 (b) JLine



(c) checkstyle

Fig. 7. Trend Graph of JCL2, JLine & checkstyle Branch Coverage

- **SAT4J**

SAT4J is a Java language Boolean SAT tool. Its input parameters is a set of real numbers. 2610 functions of SAT4J are analyzed during the static analysis process. There are 3074 branch statements in its source code. Fig. 6 (b) shows the comparison results of H-Fuzzing, JPF and random fuzzing method. As shown in this figure, when the iteration number exceeds 8,000 times, the number of covered branches almost does not increase.

- **JCL2**

JCL2 is a configurable, dynamic and extensible custom class loader that loads java classes directly from Jar files and other sources. There are 1356 branch statements in JCL2 program. The comparison results are shown in Fig. 7 (a). As illustrated in this figure, when the iteration number is more than 4,100 times, the number of covered branches reaches a relatively stable state.

- **JLine**

JLine is a Java library for handling console input. It is similar in functionality to BSD editline and GNU readline. There are 1987 branch statements in JLine program. As shown in Fig. 7 (b), after the iteration number run overs 5,900 times, the number of covered branches begins to keep about 1800.

- **checkstyle**

Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring task. There are 4729 branch statements in checkstyle program. Fig. 7 (c) illustrates the comparison results of H-Fuzzing, JPF and random fuzzing method. As demonstrated in Fig. 7 (c), after the iteration number is over 141,000 times, the number of covered branches begins to keep about 4300.

5 Conclusion and Discussion

In this paper, we present a new method named H-Fuzzing for program fuzzing data generation. H-Fuzzing achieves high program execution path coverage by virtue of the the static analysis information and the program dynamic property.

In order to effectively reduce the fuzzing data set, H-Fuzzing figures out the key branch predictions information and builds its relations with the program input variables.

During the iterative input variable generating procedure, H-Fuzzing abstracts the dynamic property from the tested program. Besides, H-Fuzzing introduces a series of fuzzing data reduction rules to improve the efficiency of the fuzzing data generation algorithm and reach a high execution path coverage rate.

H-Fuzzing has high practical value for the program security testing. In the future study, more efforts will be involved to perfect our fuzzing method, for example, recording and recovering the variable information if the next chosen path has the same execution part with the previous one. In addition, we will further improve its performance to apply it in large-scale program security testing.

References

1. Fuzz utility. ftp://grilled.cs.wisc.edu/fuzz.
2. Miller, Barton P.; Fredriksen, Lars; & So, Bryan. “An empirical study of the reliability of UNIX utilities.” Communications of the ACM 33, 12: 32-44. (1990)
3. D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. IBM Systems Journal, 22(3):229–245. (1983)
4. J. Offut and J. Hayes. A Semantic Model of Program Faults. In Proceedings of International Symposium on Software Testing and Analysis (ISSTA’96), pages 195–200. (1996)
5. J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In Proceedings of the 4th USENIX Windows System Symposium. (2000)
6. C. Csallner and Y. Smaragdakis. JCrasher: An Automatic Robustness Tester for Java. Software: Practice and Experience, 34:1025–1050. (2004)
7. C. Pacheco and M. D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In Proceedings of 19th European Conference Object-Oriented Programing. (2005)
8. J. C. King. Symbolic Execution and Program Testing. Communications of the ACM, 19(7):385–394. (1976)
9. L. Clarke. A System to Generate Test Data and Symbolically Execute Programs. IEEE Transaction on Software Engineering, 2:215–222. (1976)
10. S. Visvanathan and N. Gupta. Generating Test Data for Functions with Pointer Inputs. In Proceedings of 17th IEEE International Conference on Automated Software Engineering (ICASE’02). (2002)
11. W. Visser, C. S. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In Proceedings of 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA’04), pages 97–107. (2004)
12. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In Proceedings of the 26th International Conference on Software Engineering (ICSE’04), pages 326–335. (2004)
13. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’05). (2005)
14. R. Majumdar and R. Xu. Directed test generation using symbolic grammars. Foundations of Software Engineering, pages 553–556. (2007)
15. P. Godefroid, A. Kiezun, and M. Levin. Grammar-based Whitebox Fuzzing. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI’08). (2008)
16. R. Majumdar and K. Sen. Hybrid concolic testing. In Proceedings of 29th International Conference on Software Engineering (ICSE’07), pages 416–426. IEEE. (2007)