Hardiness sensing for susceptibility using American Fuzzy Lop

Krishna Prakash R[1]*, *Vasudevan* I[2], *Indhuja* I[3], *Thangarasan* T[4] and *Krishnan* C[5]

[1] Kongunadu College of Engineering and Technology, Trichy, India
[2] KSR College of Engineering, Tiruchengode, India
[3] Mahendra College of Engineering, Salem, India
[4] KSR College of Engineering, Tiruchengode, India
[5] Mahendra Institute of Technology, Namakkal, India

Abstract. American Fuzzy Lop is an automated software testing method to give unexceeded values, else random data's as input to computer programs for testing the hardiness process. Existing fuzzing methods will come under carried based type. Pointing to test a certain input that covers a certain region of units to fit under retain part. Still, a susceptibility over a program space may not arrive paraded in all execution that occurs to see the certain program parts; just some program executions will go the location could disclose the susceptibility. In this paper, we introduced a unified fitness metric known as direct board, which can be used for American Fuzzy lop, and that is explicitly pointed toward exploring for test input that can disclose susceptibilities. To improvise the AFL, we have enhanced its methodology to produce a more effective quality fuzzing tool. This causes our method to notice buffer invade as well as the whole number invade susceptibilities. Enhanced AFL is tested with similar benchmark programs to compare it and adding an extension to it called AFLBLEND. By our method, many uncover susceptibility could be found with a given amount of time and faster than the AFL approach by 8%.

Keywords: AFL, American Fuzzy Lop, Fuzzing.

1 Introduction

The unusual behavior of the software with a bug in the code is named software susceptibility [20]. It is done by hostile attackers and they can potentially manip-ulate to attain control over the software during its execution. To uncover such susceptibilities, it automatically pointing out test inputs in the prevailing soft-ware that helps the developers to bring in the software more protected. In the target program, the runtime errors are revealed by using the search-based soft-ware testing (SBST) [1, 2, 3] during the time of execution. Search Based Software Testing (SBST) is a class of methods with a huge number of inputs produced au-tomatically. To uncovering an error in these methods, rely on a fitness metric for test inputs to lead the search towards the test inputs that fulfill a testing objective. Compare to other test inputs with a greater fitness cadent and is closer to ful-filling the testing objective and consequently produces the newer inputs. In this way, the test inputs possible to attain the test objective to get elicited eventually.

Some various practical tools and techniques are imposed with a grey box fuzz-ing [4]. It is a certain kind of SBST approach. On each test inputs, we intend to use the lightweight instrumentation to obtain a profile from the run of the pro-gram and to utilize this profile to evaluate the fitness cadent of the interrelated test inputs. Coverage based fuzzers is a method having the object of uncovering test inputs as promptly as possible and that cause execution to attain as several parts of the program as probable. The other class of grey box fuzzing tool is known as directed fuzzers [11], use a fitness cadent that brings the program exe-cution closer to a defined susceptibility location and fitness cadent mainly gives priority to the inputs. The inconsistent existing is incremented at the location which has high enough value when its location is attained, an integer overflow [19] susceptibility in a program location will be demonstrated only in the test runs. To generate an input we need to be fuzzed further to reach the susceptibility locale that is induced by the test inputs. And it not only attains the susceptibility area but also uncovers the susceptibility. However, the intention of both cover-age-based fuzzers and directed fuzzers is being met when a test run attains the susceptibility area and it would not fuzz the interrelated test input further. This process stimulates our key hypothesis, which leads the search towards the test inputs, which one wants vulnerability specific fitness metric that not only attains the susceptibility areas, but it obtains them along with

* Corresponding author: krishcommon@gmail.com

the execution paths that the program states result cause the susceptibility to be uncovered. We nurture this concept in the rest of the section.

1.1 Causative Example

The illustration flowchart showed in figure 1 performs as a running and motivating representation. And this flowchart initially reads the input from a buffer input (step 2) and character 'b' copies all occurrences in the input from another buffer outC (step 7). The program units are tagged with letters A to F at (step 7) the buffer overflow [12] error remark that this program is susceptible. The buffer outC size is 250 if the input comprises more than 250 characters 'b' of happenings then the outC overflow at this statement. The well-known coverage based grey box fuzzing tool is AFL [6] here we discuss pertain why coverage-based fuzzing [5] may not be enough to find susceptibilities. The branch of pair is outlined as it's a pair of sequentially executable branches of the program e.g.: in (figure 1) AB, AC, BD, BE, CE, and EF, are instances of branch pairs. To each branch pair during an enactment, the branch pair contour counts the number of visits of each branch pair. And it further buzzing off for at least one branch pair bp, AFL evaluates the newly produced test input t, to be capable and maintains. the visit count of the same branch pair by each execution on each other maintained test input is significantly distinct from the visit count of bp by the enactment on t so far. AFL uses the higher bar significant difference and many test inputs dramatically slow down the persuasiveness and efficiency of the search as maintaining too. Ap-proximately speaking if their logarithms are the base 2 are distinct that two counts are significantly varied.

The running example is shown in figure 2(partially) the prospect tree of devel-oped test inputs that may be produced by the AFL [11]. It is used because due to randomization in the transformation operations. A similar tree may not be produced in every run of AFL. the relationships between parent-child are characterized by the edges in the test inputs and the base node is a given seed input “a “. From the middle, tree runs vertically down we may concentrate on a specific path. At the right side of the input, the blue color indicates the branch pair profile and its test inputs. The number of 'b's in the input is illustrated by the number suggested within the round parenthesis for each test input on this path. There is no symbol other than 'b' in the input. For an instance, we can ignore all the redcolored numbers within the boxes, as they are relayed to AFL [11]. It contains six inputs, the first four inputs will be maintained in this path, but the fifth input will not be retained because the contour of the parent and the branch pair contour is not relatively varied from its parent profile (i.e., 131 'b's). Therefore, the fifth input would not be produced the sixth input (I.e.,261'b's). Remark that at statement 7 the sixth input would have been the first one to inflict the buffer to over-flow in this path.

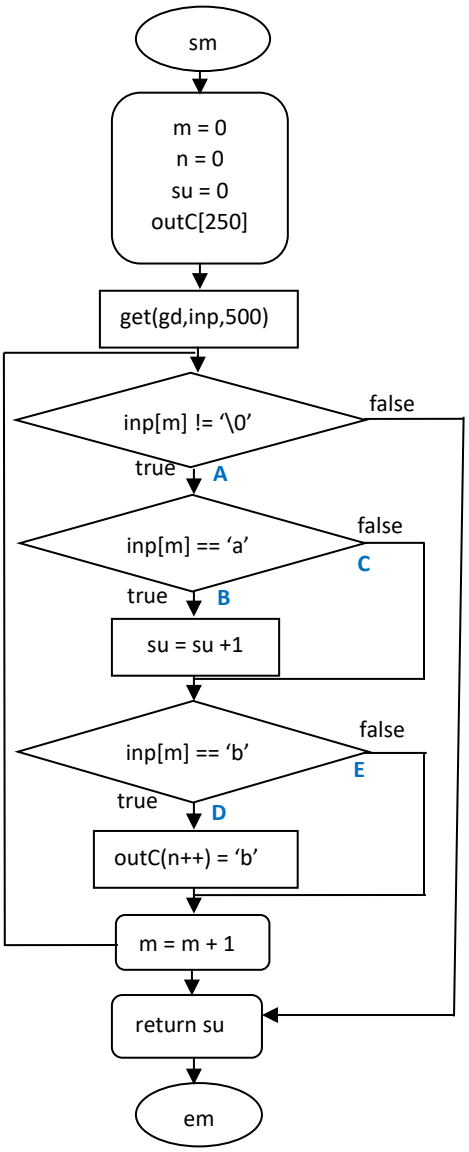

Fig. 1: Example program for buffer overrun susceptibility

In supplementary phrases, by mutating the fourth test input the AFL would have to produce an input comprising more than 250'b's immediately(i.e.,131'b's) and this process can transpire only via a mutation operation that puts in 129 or more 'b's in a single operation. The process is set up in a way that the randomization in AFL is the ratio of inserting a bigger number of identities in a single mutation operation are instantly proportional to the number of mutant test inputs produced by AFL so far. Thus, such "large" mutations strive for a lot of time could elapse earlier.

In mutation operation, the run of AFL ahead in a simple implication here would be to improve the proportion of putting a larger number of bytes. Regard-less, this straightforward idea would originate extensively input files before on, thus each test input duration of running is improving, and holding back the number of mutants is produced per unit time. And these several large mutations may not beneficial also, e.g., ones that insert

an extra number of 'a's than 'b's. there-fore, the uncovering susceptibilities of the possibility may not be obtained before.

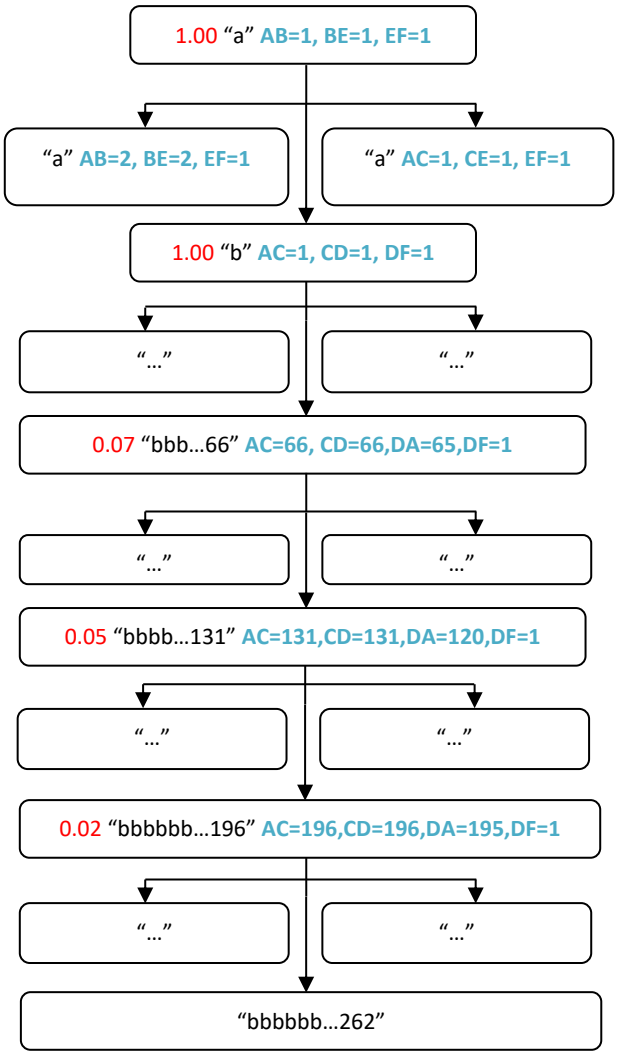

Fig. 2: Production of test input for fuzzing

This issue is not inevitably significant in addressed by even directed fuzzing. In our instance, the first-time (step 7) is breached a directed fuzzer would con-template its matter-of-fact as have been met the first moment. That this would transpire with modest input 'b' itself, and which does not uncover the susceptibility.

2 Susceptibility Spotting Method

In this section, by extending our method to detect susceptibility over buffer run and the methodology used for it.

2.1 Methodology

Susceptibility is a glitch at a susceptibility location s_1 in a program P such way on certain carrying out paths will reach s1 as an error occurs, in that way the malicious attackers will lead to having unexpected action and overtake the control to serve their purposes. Susceptibility is of various types, e.g., buffer overflow, integer overruns, reuse after completion, etc. Let S_L be the set of altogether susceptibility locations in the given program P. (This methodology for various susceptibility locations to display several types of susceptibilities.) A chiefboard c ∈ [1,0] is an evaluation of how near a susceptibility is found in susceptibility location s1. The least value represents how close is susceptibility found, as c=0 gives a higher probability of susceptibility.

To explain the functioning of ISNEARER that takes a susceptibility profile N_h from a given input, compares it with N_l and gives an answer containing a Boolean value, and updates the N_l value.

bool ISNEARER(Nh)

{

$scp = \exists S_l \in S_L.\ N_h(S_l) \prec N_L(S_l)$

$N_l = \min(n_L, N_h)$

return scp

}

The function ≺ is a comparison of log2 plate; i.e., its second value is above 0.50 then it returns true if the first value is below 0.50; again, if the second value is betwixt 0.25 and 0.50, it delivers true if the first argument is lower than 0.25 and below.

2.1.1 Our Methodology

Our methodology is a combination of the AFL model to produce AFLBLEND[7], where more than two values are tested and involved at the same time using their functions. To accomplish two sets of given test inputs. Set 1: Until getting of dissimilar values compared to previous test values. This belief is acquired from a report conducted from fuzzing technique [13], and is meant to assure that it gives various points near to susceptibility locations are beget. Set 2: Given test input doesn't reach new values, but gets lower chiefboard at susceptibility locations greater than previously acquired test inputs.

Argument 16-23 of algorithm 1 add up the freshly generate test values v_g if both v_g achieves a various branch couple profile than some inputs in V_G, as tried by AFLBLEND's reinforced function CHECKIF, else it achieves less chiefboard than older generated test input for some susceptibility location, as tried through the function ISFINAL, that has already shown in part 2.1.

Later algorithm 1 is concluded, the susceptibility disclose test inputs are obtained by selecting the test inputs in V_G where susceptibility visibility has a zero-entry for any susceptibility location.

3 Initialize chiefboard to Particular Susceptibility Type

In this part, by establishing the idea of the impression of chiefboard to initialize it to a real-time susceptibility.

3.1 Susceptibilities in Overflow Buffer

A buffer overflow comes when a buffer or array is scripted to above its limits. It is dominant susceptibility in the real-world program; for example, it ranks third in the top 10 harmful program errors in the CVE database [8].

Seeing a buffer approaching location s1 of the form "*loc = ….", where loc is a pointer, and consider a particular visit to this location during a run. Let L_c be the value of loc during this call, and let L_h and i be the initial address and size of the allotted buffer within that loc is alleged to choose to during this call. Then define the chiefboard c1 as follow below:

$$C1 = \begin{cases} 0, \text{ if } L_c \geq L_h + i \\ \frac{Lh+i-Lc}{i}, \text{ if } L_h \leq L_c < L_h + i \\ 1, \text{ otherwise} \end{cases}$$

At last, the chiefboard for all the run at s1, i.e $N_h(s_1)$, is outlined as the low value of i_l as explained above among every call to s_1 while running.

Algorithm 1: Detecting Susceptibility by Testing
Essential: Main Program MP, and a set of input v
Assure: A tree of various input to test V_G for MP.
Create an abandon tree V_G of various inputs.
$[f_v, i_v]$ = EXECUTE(MP, v.data)
n_r.data = v n_r is a newly tree set
n_r.fit = EVALUATEFIT (f_v)
n_r.reasonToAcuqire = "original"
V_G.setRoot[n_r]
$N_L = \lambda s_l \in S_L$. 1
repeat
get n = CHOOSENEXTI(V_G)
get T = LETFUZZPOSSIBLEI(n)
get V_n = ACQUIREOFFSPRING(n,T)
while all v_g in V_n do
Get $[f_g, i_g]$ = EXECUTE(MP, V_g.value)
Get N_g = EVALUATEFIT(f_g)
Get N_h = SetVPROFILE(i_g)
if CHECKIS(N_g) V (c = ISFINAL(n_h))
then
v_g.check = N_g
v_g.causeToAccquire = "original"
if c then
v_g.nearest = N_h
v_g.causeToAccquire = "chieboard"
endif
appendnew(V_G, V, v_g)
endif
end while
until manual termination
return V_G

4 Executions

In this part, we discuss the execution of our method, on finding over the buffer overflow and value overrun susceptibility as mentioned in part 3. In this method the extension is AFL. Where AFL has quite famous for researchers to execute and measure extension to fuzzing [11, 9, 10]. The AFL is a method in C program fuzzers. Additional extension AFL changes to have two sets of test inputs, from set 2 to fuzzing, and evaluate the fuzzing potential for set 2 input if got importantly than susceptibility data than another set 2 test inputs in the tree V_G. The changes were discussed in part 2.

```
void *glb_obuf;

int gld_siz_obuf;

void main() {

        char in[100], out[50], c;

        int i = 0;

        glb_obuf=(void *) obuf;

        glb_size_obuf = 50;

        fgets(in, 100, stdin);

        while((c = in[i++]) != null) {

                _hdr_calc_bov(1,glb_obuf,
(void *) out, glb_size_obuf);

                *out++ = c;          }

}
```

Fig. 3: Program for buffer overflow

5 Valuation Using Benchmark Programs

To evaluate our Buffer overflow susceptibilities (BOS) and Value overrun susceptibilities (VOS) a set of eight benchmark programs recognized by "MIT Benchmark" compounded by "SV-COMP benchmark" [15]. The bug size taken for valuation is mentioned in the x-axis up to 25 in figure 4 likewise 160 in figure 5. This method is compared with AFL and AFLBLEND is our tool extension added to it.

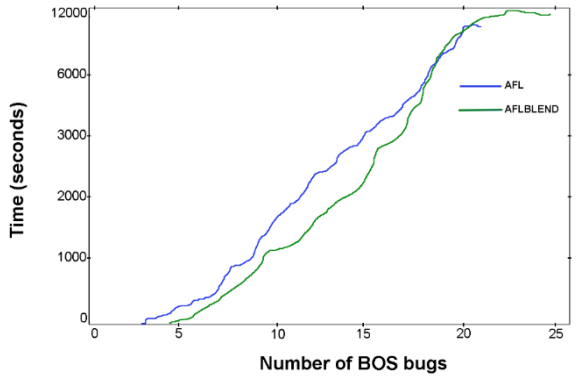

Fig. 4: Results for buffer overflow susceptibilities

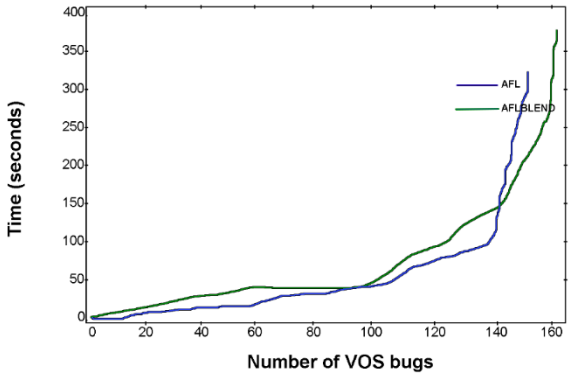

Fig. 5: Results for value overrun susceptibilities

Table 1. Mean value of buffer overrun violations perceived by every tool and results of statistical important tests.

Benchmark		v1	v3	v4	v5	c1	c3	c4	g1
Number of susceptible locations		28	3	4	3	1	1	2	4
Suscep. Location detected	AFL (A)	8	2	3	0	1	0	1	1
	statistically important?	Y	Y	Y	Y	N	Y	N	N
	AFLBLEND (B)	17	3	4	1	1	1	1	1
	statistically important?	Y	Y	Y	Y	N	N	N	N

5.1 Susceptibilities in Overflow Buffer

MIT benchmarks we used are initialized as v1, v3, v4, v5, c1, c3, c4, and g1. These eight MIT benchmarks are injected with 46 susceptible buffer overflow locations. It took nearly 3 hours for each benchmark test and each benchmark has tested 5 times depending on the time and randomness susceptible given to each program. Since it is a strict evaluating method competed with SV_COMP and Test_COMP [16, 17], evaluation of each tool depends on each benchmark. Figure 5 and Table 1 shows the overall result of benchmark results.

Figure 5 and table 1 shows the overall result of benchmark results.

5.1.1 Value Overrun Susceptibilities

These primary benchmarks test for SV_COMP will use over 177 values to check the susceptibilities. Since the susceptibilities are smaller in size each tool will test over 900 seconds for each benchmark. The values given are likely one negative number, one zero number, and one large positive number. Each set will run over 5 times on every benchmark and figure 6 shows the overall result in the decided format and found that AFLBLEND is on top compared to the traditional AFL method.

6 Conclusions

By addressing the problem to expose the susceptibilities that can be found while testing software programs and to detect susceptibilities location in the coding part we implement the AFLBLEND method by deriving core part from AFL. We proposed a method that fits right from the first-time susceptibility specific fitness metric to call and continue test input that comes near to exposing susceptibilities. A valuation of our AFLBLEND tool on normally used benchmarks program generates that our method finds multiple times many susceptibilities or detects them in a short period compared to AFL also with recently added directed fuzzer.

The main restriction over this concept that while running on benchmark program we have run with limited values of inputs, not in large scale values [18] to the program. To address this issue, we might need to migrate our method to have Whitebox Fuzzing [21] on Low Level Virtual Machine (LLVM) so that it will enhance memory availability for tracking chiefboard during program execution. To keep track of chiefboard values a memory address checker method such as Address Sanitizer (ASAN) [14] can be used to accurately keep track of it. These works could consistently improve to move for a large set of database tests and challenging type of susceptibilities.

7 References

1. Gordon Fraser & Andrea Arcuri (2012) in Whole test suite generation IEEE Transactions on Software Engineering 39, 2 (2012), 276–291.
2. Phil McMinn. (2004) in Search-based software test data generation: a survey. Software Testing, Verification and reliability 14, 2 (2004), 105–156.
3. Joachim Wegener, André Baresel, & Harmen Sthamer. (2001) in Evolutionary test environment for automatic structural testing. Information and Software Technology 43, 14 (2001), 841–854.
4. Jared DeMott, Richard Enbody, & William F Punch. (2007) in Revolutionizing thefield of grey-box attack surface testing with evolutionary fuzzing. BlackHat and Defcon (2007).
5. Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, & Herbert Bos. (2017) in VUzzer: Application-aware Evolutionary Fuzzing. In USENIX security. 1–14.
6. Michael Zalewski. [n.d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/
7. Mark Harman, S Afshin Mansouri, & Yuanyuan Zhang. (2012) in Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys (CSUR) 45, 1 (2012), 11.
8. CVE database (2017) in CVE Details - Vulnerabilities by type. Technical Report.

https://www.cvedetails.com/vulnerabilities-by-types.php

9. Caroline Lemieux, Rohan Padhye, Koushik Sen, & Dawn Song. (2018) in Perffuzz: Automatically generating pathological inputs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 254– 265.
10. Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, & Giovanni Vigna. (2016) in Driller: Augmenting fuzzing through selective symbolic execution. In Proceedings Network & Distributed System Security Symposium(NDSS). 1–16.
11. Raveendra Kumar Medicherla, Raghavan Komondoor, & Abhik Roychoudhury (2020) in International Conference on Software Engineering (ICSEW'20), 513-520.
12. Misha Zitser, Richard Lippmann, & Tim Leek. (2004) in Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code. In Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (FSE) (Newport Beach, CA, USA). New York, NY, USA, 97–106.
13. George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, & Michael Hicks. (2018) in Evaluating fuzz testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2123–2138.
14. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, & Dmitriy Vyukov. (2012) in Address Sanitizer: A Fast Address Sanity Checker. In USENIX Annual Technical Conference. 309–318.
15. Dirk Beyer. (2019) in Automatic verification of C and Java programs: SV-COMP 2019. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 133–155.
16. Dirk Beyer. (2019) in International competition on software testing (Test-Comp). In International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 167–175.
17. Dirk Beyer & M Erkan Keremoglu. (2011) in CPA checker: A tool for configurable software verification. In International Conference on Computer Aided Verification. Springer, 184–190.
18. Cristian Cadar, Daniel Dunbar, Dawson R Engler & et al. (2008) in KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In OSDI, Vol. 8. 209–224.
19. Will Dietz, Peng Li, John Regehr, & Vikram Adve. (2015) in Understanding integer overflow in C/C++. ACM Transactions on Software Engineering and Methodology (TOSEM) 25, 1 (2015), 2.
20. Gordon Fraser & Andrea Arcuri. (2015) in 1600 faults in 100 projects: automatically finding faults while achieving high coverage with EvoSuite. Empirical software engineering 20, 3 (2015), 611–639.
21. Patrice Godefroid, Michael Y Levin, David A Molnar & et al. (2008) in Automated Whitebox Fuzz Testing. In NDSS, Vol. 8. 151–166.
22. Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and R Venkatesh. 2019. VeriFuzz: Program aware fuzzing. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 244--249.
23. Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. SAVIOR: Towards Bug-Driven Hybrid Testing. In IEEE Symposium on Security and Privacy (SP), 2020.