Research on Fuzzing Technology Based on Dynamic Fitness Functions Postprint

Authors: Deng Yijie, Liu Kesheng, Zhu Kailong, Chang Chao

Date: 2018-07-23T00:00:00+00:00

Abstract

Fuzzing is an effective automated vulnerability discovery technique. Mainstream fuzzing techniques employ genetic algorithms to generate test cases, which suffer from premature convergence, leading to insufficient path coverage. To address this problem, we propose a fuzzing method based on dynamic fitness functions. Comprehensively considering factors of seed novelty and path depth, we design a fitness function that varies dynamically according to different testing phases, and implement a fuzzing tool named DynFuzzer based on dynamic fitness functions. Experiments conducted on test sets provided by BegBunch and CGC demonstrate that, compared with existing fuzzing tools, DynFuzzer improves path coverage by 40% and discovers 10% more bugs. The fuzzing method based on dynamic fitness functions can effectively overcome the premature convergence problem, enhance path coverage, and discover more bugs.

Full Text

Preamble

Research on Fuzzing Techniques Based on Dynamic Fitness Functions

Deng Yijie, Liu Kesheng, Zhu Kailong, Chang Chao
(National University of Defense Technology, Electronic Countermeasures Institute, Hefei 230031, China)

Abstract: Fuzzing is an effective automated vulnerability mining technique. Mainstream fuzzing approaches employ genetic algorithms to generate test cases, but suffer from premature convergence, resulting in insufficient path coverage. To address this issue, this paper proposes a fuzzing method based on dynamic fitness functions. By comprehensively considering seed novelty and path depth factors, we design a fitness function that dynamically changes according to different testing phases, and implement a fuzzing tool called DynFuzzer based on this dynamic fitness function. Experiments conducted on test sets provided by

BegBunch and CGC demonstrate that compared with existing fuzzing tools, DynFuzzer improves path coverage by 40% and discovers 10% more bugs. The dynamic fitness function-based fuzzing method can effectively overcome the premature convergence problem, improve path coverage, and discover more vulnerabilities.

Keywords: fuzzing; genetic algorithm; dynamic fitness function; DynFuzzer; path coverage

0 Introduction

Fuzzing is an effective vulnerability mining technique characterized by high automation, strong scalability, and applicability to large-scale programs, making it widely used in software testing. Statistics show that 20%-25% of security vulnerabilities in Microsoft products are discovered through fuzzing before release. Therefore, designing an efficient fuzzing method holds significant importance in software security.

Based on test case generation methods, fuzzing can be categorized into: (1) approaches combining format analysis with program understanding, and (2) approaches combining static analysis with dynamic testing. Representative tools of the first category include SPIKE and Peach, which offer high execution efficiency, broad applicability, and strong generality, but still require substantial manual effort for knowledge acquisition. Representative tools of the second category include BuzzFuzz, which provides high code coverage but cannot overcome path explosion or handle intensive program checking.

In 2016, AFL (American Fuzzy Lop) became one of the most commonly used industrial-strength fuzzing tools due to its simplicity and efficiency, combining heuristic algorithms with fuzzing. AFL employs a path coverage-guided genetic algorithm for fuzzing, which improves path coverage compared to other methods. However, AFL' s genetic algorithm uses a fixed fitness function that fails to fully utilize dynamically obtained program path information during testing, leading to premature population gene convergence. This makes it difficult to discover new paths in later testing stages, resulting in insufficient path coverage.

To address these issues, this paper proposes a fuzzing method based on dynamic fitness functions. The fitness function design comprehensively considers both path novelty and depth factors, assigning different weights to these factors at different fuzzing stages to achieve higher path coverage.

1 Problem Analysis

1.1 AFL' s Test Case Generation Method

AFL uses a genetic algorithm to generate test cases. Initially, users provide a set of seed test cases. Then, guided by path coverage, AFL employs mutation operations to generate the next generation of test cases, retaining those that

produce new paths as seeds. This process iterates continuously to discover bugs by progressively covering program execution paths. In AFL, seeds are selected from test cases, which are in turn generated through seed mutation. If a test case executes a new path during execution, it is added to the seed set to generate the next population. This establishes a one-to-one correspondence between seeds and paths, meaning AFL' s population size only increases.

During seed execution, AFL employs a unique "favorite" strategy. For each path segment, the seed that reaches that segment fastest or with the smallest input size is marked as "favorite." These favorite seeds have a higher probability of being selected. After seed selection, each seed mutates to generate a certain number of test cases. The number of test cases generated is determined by the seed' s execution time, path segment coverage, and time spent generating new inputs. Favorite seeds generate thousands of test cases each time they are selected, while other seeds produce only dozens. While AFL' s special seed selection and energy allocation strategy improves fuzzing efficiency, it also leads to premature population gene convergence and insufficient code coverage.

1.2 AFL Case Study

We conducted a fuzzing test on the b64_{encode}1 challenge from CGC using AFL for 2 hours. AFL performed approximately 2,000 cycles. The distribution of seed selection times is shown in [Figure 1: see original paper], where the x-axis represents seed IDs and the y-axis represents the number of times each seed was selected. The results show that seeds with IDs 1, 7, 8, 13, 15, and 18 were selected far more frequently than others—these are the seeds marked as favorite.

In the same experimental environment, we also counted the number of test cases generated by each seed and calculated the average number of test cases generated when each seed was selected, as shown in [Figure 2: see original paper]. The x-axis represents seed IDs sorted in descending order by the number of test cases generated, while the y-axis shows the number of test cases generated per seed in one fuzzing cycle (i.e., the number of times the path was executed). The analysis reveals several characteristics of AFL' s test case generation: (a) 38% of seeds generated extremely few test cases; (b) 28% of seeds generated the vast majority of test cases.

It is evident that AFL spends most of its time executing favorite seeds, with non-favorite seeds accounting for less than 1% of execution time. However, the favorite designation is based on seed execution speed and size, not on the potential to generate new paths. These factors cause AFL' s population genes to converge prematurely, resulting in insufficient path coverage.

2 Dynamic Fitness Function Based on Path State

To effectively concentrate fuzzing resources on defect-prone areas and achieve higher path coverage, this paper proposes a fuzzing method based on dynamic

fitness functions. The approach combines two fitness functions: a novelty-prioritized fitness function and a depth-prioritized fitness function, employing a fitness function that dynamically changes according to the testing phase.

2.1 Novelty Fitness Function

The novelty fitness function aims to make the program' s control flow graph as complete as possible and discover shallow bugs. Based on experiments and vulnerability mining experience, we propose two heuristic rules for the early stages of fuzzing:

Heuristic Rule 1: The newer the path fragments discovered by a seed, the greater the probability that its offspring will discover new paths in the next round of testing.

Heuristic Rule 2: The more path fragments on a seed' s execution path, the greater the probability that its offspring will discover new paths in the next round of testing.

Based on these heuristic rules, we define a seed' s novelty fitness as the sum of the novelty values of all path fragments on its execution path:

$$\text{Fitness}_{nfs}(\text{seed}) = \sum_{i \in \text{path(seed)}} \text{value}_{nfs}(i)$$

where i is the ID of a path fragment on the seed' s execution path, $\text{cyc}(i)$ is the cycle number when path fragment i was first executed, and $\text{value}_{nfs}(i)$ represents the novelty of path fragment i. The impact of time on fitness is defined at an exponential level, increasing the selection priority of seeds that discover new path fragments.

[Figure 3: see original paper] illustrates the calculation of test case novelty fitness through an example. The figure shows a partial control flow graph of a program, where numbers on path fragments indicate the cycle when each fragment was discovered. Assuming three seeds c_1, c_2, and c_3 with execution paths *a*-*b*-*d*, *a*-*c*-*e*, and *a*-*c*-*f* respectively, we calculate their novelty fitness values as:

$$\text{Fitness}_{nfs}(c_1) = 2^{-1} + 2^{-2} + 2^{-2} = 6$$

$$\text{Fitness}_{nfs}(c_2) = 2^{-1} + 2^{-2} + 2^{-10} = 10$$

$$\text{Fitness}_{nfs}(c_3) = 2^{-1} + 2^{-2} + 2^{-2} = 14$$

Thus obtaining the novelty fitness values for the three seeds.

2.2 Depth Fitness Function

The depth fitness function aims to select more dangerous paths from the program' s path set and conduct targeted fuzzing on these paths to discover deep bugs. Based on experiments and vulnerability mining experience, we propose a heuristic rule for the later stages of fuzzing:

Heuristic Rule 3: The deeper the path corresponding to a seed, the more likely it is that bugs exist on that path.

Based on this heuristic rule, we prioritize path depth as the dominant factor in the fitness function during later fuzzing stages to increase the probability of discovering deep bugs.

2.2.1 Mapping Control Flow to Markov Chain A Markov chain is a discrete-time stochastic process in probability theory where state transitions satisfy the property that past states have no influence on future predictions—future states depend only on the current state. The research targets binary programs written in structured languages like C. For structured programs, each basic block can be viewed as a state. Previous basic blocks have no influence on which basic block may be reached next; the next possible basic block depends only on the current one. Therefore, program execution flow satisfies the Markov property and can be abstracted as a Markov chain.

Treating program basic blocks as states, paths can be viewed as state transition processes. The probability of each path is the state transition probability, which is obtained through statistics. AFL inserts lightweight instrumentation at each program branch point to obtain path information. Our method uses this instrumentation to record the number of test cases taking different branches at branch points. The proportion of branch cases to total cases defines the probability of that branch, as illustrated in [Figure 4: see original paper].

Assuming after one fuzzing process, m test cases pass through branch AB and n test cases pass through branch AC, then:

$$p(AB) = \frac{m}{m+n}$$

$$p(AC) = \frac{n}{m+n}$$

2.2.2 Path Weight Calculation This method calculates a seed' s depth fitness by computing the weight of each path. A seed' s depth fitness is obtained by summing the weights of basic blocks along its corresponding path. Basic block probabilities are derived from path state transition probabilities using the following formula:

$$p(S) = \sum_{D \in u(S)} p(D) \cdot p(D \to S)$$

where S is a basic block, $u(S)$ is the set of basic blocks that can reach S in one state transition, $D \to S$ represents the path from basic block D to S, and $p(D)$ and $p(\text{edge})$ denote the probabilities of basic block D and path fragment respectively.

The weight of each basic block is defined as the reciprocal of its state probability:

$$w(S) = \frac{1}{p(S)}$$

If a seed corresponds to a path passing through the set of basic blocks block(seed), then the depth fitness of that path is:

$$\text{Fitness}_{dfs}(\text{seed}) = \sum_{\text{Dot} \in \text{block(seed)}} w(\text{Dot})$$

It should be noted that new paths are continuously generated during fuzzing, so the program' s control flow graph changes dynamically. The control flow graph is updated after each fuzzing cycle, and new seed fitness values are calculated based on the updated graph for the next round of fuzzing. [Figure 5: see original paper] illustrates a complete fuzzing process for path weight calculation.

[Figure 5: see original paper] shows the program's control flow graph, with values (a, b) next to each basic block, where a represents the basic block probability and b represents its weight. Numbers between basic blocks indicate transition probabilities. Suppose seed c_4 corresponds to path Main $\to B \to C \to E \to F \to$ Bug $\to$ Return, and seed c_5 corresponds to path Main $\to B \to C \to D \to G \to$ Return. We calculate their depth fitness values based on the current control flow:

First, calculate each basic block' s probability and weight from the current control flow graph:

$$p(B) = 1 \times 0.7 = 0.7$$

$$w(B) = \frac{1}{p(B)} = 1.62$$

$$p(G) = 0.35 \times 0.8 + 0.35 \times 0.6 + 0.14 \times 0.9 = 0.616$$

Then sum the weights of basic blocks along each path (excluding start and end blocks) to obtain the corresponding seed' s depth fitness:

$$\text{Fitness}_{dfs}(c_4) = w(B) + w(C) + w(E) + w(F) + w(\text{Bug}) = 84.29$$

$$\text{Fitness}_{dfs}(c_5) = w(B) + w(C) + w(D) + w(G) = 7.34$$

2.3 Dynamic Fitness Function

Based on the two fitness function calculation methods described above, this approach implements a dynamic fitness function design by assigning different weights to them in different periods. To prevent the novelty fitness and depth fitness from interfering with each other, they are first normalized:

$$\text{Fitness}_{nfs}(\text{seed}) = \frac{\text{Fitness}_{nfs}(\text{seed})}{\sum_{j\in\text{set}} \text{Fitness}_{nfs}(j)}$$

$$\text{Fitness}_{dfs}(\text{seed}) = \frac{\text{Fitness}_{dfs}(\text{seed})}{\sum_{j\in\text{set}} \text{Fitness}_{dfs}(j)}$$

where set represents the seed set for that cycle.

The normalized fitness functions are then combined with different weights:

$$\text{Fitness}(\text{seed}) = \begin{cases} (1-\log(a)) \cdot \text{Fitness}_{nfs}(\text{seed}) + \log(a) \cdot \text{Fitness}_{dfs}(\text{seed}) & \text{cyc} \le a \\ (1-\log(b)) \cdot \text{Fitness}_{nfs}(\text{seed}) + \log(b) \cdot \text{Fitness}_{dfs}(\text{seed}) & \text{cyc} > a \end{cases}$$

To ensure continuity of the fitness function, parameters a and b satisfy $2 \cdot \text{rush} = a = b$. The rush value can be customized based on program scale, with a default of 20. Rush is a fitness threshold indicating that in cycles fewer than rush, novelty fitness has greater weight than depth fitness, while in cycles beyond rush, novelty fitness has less weight than depth fitness. This design dynamically changes the genetic algorithm's fitness function to avoid population convergence and effectively improve path coverage and the probability of discovering deep bugs.

After seed selection, the energy allocation phase begins. Seeds with higher fitness receive more energy and generate more test cases. Before fuzzing begins, a maximum number of test cases is defined for each seed:

$$\text{MAXcase}(i) = 2^{\text{split}(i)}$$

where $\text{split}(i)$ is the number of path fragments in the path corresponding to seed i. The final energy allocated to seed i is:

$$\text{power}(i) = \begin{cases} 10 & \text{if } \text{Pro}(i) \cdot \text{MAXcase}(i) < 10 \\ \text{Pro}(i) \cdot \text{MAXcase}(i) & \text{if } 10 \leq \text{Pro}(i) \cdot \text{MAXcase}(i) \leq 10^5 \\ 10^5 & \text{if } \text{Pro}(i) \cdot \text{MAXcase}(i) > 10^5 \end{cases}$$

where $\text{Pro}(i) = \frac{\text{Fitness}(i)}{\sum_{n=1}^{|\text{set}|} \text{Fitness}(n)}$ is the probability of selecting seed i.

This dynamic fitness function-based fuzzing method can effectively discover complete program paths and allocate fuzzing time and computational resources to areas more likely to contain bugs, facilitating the discovery of deeper code defects. The drawback is a slight decrease in the rate of discovering crashes during early fuzzing stages.

3 DynFuzzer System Design and Implementation

Based on the dynamic fitness function-based fuzzing method proposed above, we designed and implemented a fuzzing tool called DynFuzzer. This section introduces the genetic algorithm and system architecture used by DynFuzzer.

3.1 Seed Selection and Energy Allocation Algorithm

DynFuzzer selects seeds for fuzzing and allocates energy based on their dynamic fitness values. In this context, energy is defined as the number of test cases generated through seed mutation. After calculating the dynamic fitness of seeds in the seed set, DynFuzzer uses the roulette wheel algorithm to select seeds for fuzzing. Each seed has a certain probability of being selected, with the probability of selecting seed i given by:

$$\text{Pro}(i) = \frac{\text{Fitness}(i)}{\sum_{n=1}^{|\text{set}|} \text{Fitness}(n)}$$

This formula indicates that the energy allocated to seeds is bounded: a minimum of 100 test cases and a maximum of 100,000 test cases are generated.

3.2 System Architecture

The DynFuzzer system consists of two modules: an agent module and a monitoring module, as shown in [Figure 6: see original paper].

Agent Module: The main fuzzing module responsible for selecting seeds from the seed set for fuzzing, mutating them to generate test cases, executing the target program with these test cases, and selecting test cases that execute new paths to add to the next generation seed set.

Monitoring Module: Responsible for monitoring whether crashes occur during test case execution and recording crash details such as register values and stack information.

The system operation flow is as follows: 1. Users input the initial test case set for execution. 2. Record program execution path information. If a crash occurs, invoke the monitoring module to record detailed execution information, then select test cases that executed new paths to add to the next generation seed set. 3. Calculate seed fitness based on path information from step 2, then perform seed selection and energy allocation according to fitness values. 4. Mutate seeds selected in step 3 to generate test case sets and execute the target program, then return to step 2.

The fuzzing cycle continues until manually terminated or until testing stops after a certain period without discovering new paths.

4 Experiments

4.1 BegBunch

We used BegBunch as the experimental dataset for DynFuzzer. BegBunch is an artificially constructed bug test suite containing 67 crashes. The experimental environment was 64-bit Ubuntu 14.04 with 8 CPU cores and 12 GB of memory. The running time was 6 hours. Statistics on the number of paths and crashes discovered by AFL and DynFuzzer at different time points are shown in [Figure 7: see original paper] and [Figure 8: see original paper].

The x-axis in both figures represents fuzzing time. The y-axes represent the number of discovered paths and crashes respectively. Experimental results show that DynFuzzer' s ability to discover new paths is slightly weaker than AFL in the early stage. However, after a certain number of cycles, DynFuzzer ultimately discovered 13,058 paths, while AFL discovered only 7,613 paths—DynFuzzer found 40% more paths than AFL. Regarding crash discovery, DynFuzzer' s capability remains relatively stable across different time periods, with late-stage fuzzing performance similar to early-stage performance, ultimately discovering 66 crashes. In contrast, AFL' s fuzzing capability clearly degrades over time, demonstrating its insufficient late-stage fuzzing capability, and ultimately discovered only 60 crashes. DynFuzzer with dynamic fitness functions discovered 10% more crashes than AFL.

4.2 CGC Challenges

We collected challenges from the Cyber Grand Challenge (CGC) as the test set for this experiment. Due to varying difficulty levels in CGC challenges, both AFL and DynFuzzer can quickly discover crashes in simple challenges, while neither can discover crashes in extremely difficult ones, making performance differences indistinguishable. Therefore, we only 统计了中等难度的、Fuzz 时间较长的测试用例测试情况. This experiment used the same environment as Section 4.1,

but both tools were run for 12 hours before crash statistics were collected, with results shown in [Figure 9: see original paper].

The x-axis represents CGC challenge IDs, and the y-axis represents the number of crashes discovered after fuzzing completion. The experiment revealed that DynFuzzer discovered 438 crashes in total, while AFL discovered 348 crashes—DynFuzzer found 26% more crashes than AFL. Considering that the 统计的 CGC 赛题都是中等难度的, and the real test set also contains 10 simple challenges (for which both AFL and DynFuzzer discovered all crashes) and 5 extremely difficult CTF challenges (for which neither AFL nor DynFuzzer could discover crashes), the comprehensive impact of these challenges on the comparative experiment was calculated. The result shows that DynFuzzer discovered on average 10.4% more crashes than AFL.

5 Conclusion

To address the problems of premature population gene convergence and insufficient path coverage in current fuzzing techniques, this paper proposes a dynamic fitness function-based fuzzing method and implements the fuzzing tool DynFuzzer based on this method. Experiments demonstrate that DynFuzzer discovers 40% more paths than AFL in terms of path coverage, and discovers 10% more crashes on average. DynFuzzer has two limitations: (1) its ability to generate new paths and crashes is relatively weak in the early fuzzing stage; (2) due to the impact of module coupling on the conversion to Markov chain models, this method still cannot achieve ideal results for complex unstructured programs and large-scale real-world programs. Addressing these two issues requires further research in future work.

References

[1] Lian Mei, Zhou Yanyan, Huo Wei, et al. Parallel fuzzy testing framework for dynamic resource perception[J]. Application Research of Computers, 2017, 34(1): 52-57.

[2] Li Honghui, Qi Jia, Liu Feng, et al. Research on fuzzy testing technology[J]. Chinese Science: Information Science, 2014, 44(10): 1305-1322.

[3] Wu Zhiyong, Wang Hongchuan, Sun Lechang, et al. Review of fuzzing technology[J]. Application Research of Computers, 2010, 27(3): 829-832.

[4] Peach[EB/OL]. (2014-02-23)[2015-11-28]. http://community.peachfuzzer.com/.

[5] Luo Yongbiao, Ye Jun, Ma Xiaowen. Multicriteria fuzzy decision-making method based on weighted correlation coefficients under interval-valued intuitionistic fuzzy environment[J]. European Journal of Operational Research, 2010, 205(1): 202-204.

[6] Vuori J. Student engagement: buzzword of fuzzword?[J]. Journal of Higher Education Policy & Management, 2014, 36(5): 509-519.

[7] Caroline Lemieux, Koushik Sen. FairFuzz: targeting rare branches to rapidly increase greybox fuzz testing coverage[J]. 2017.

[8] Serebryany K. Continuous fuzzing with Libfuzzer and address sanitizer[C]//Cybersecurity Development. 2017: 157-157.

[9] Cha S, Woo M, Brumley D. Program-adaptive mutational fuzzing[C]//Proc of IEEE Symposium on Security and Privacy. 2015: 725-741.

[10] Rawat S, Jain V, Kumar A, et al. VUzzer: application-aware evolutionary fuzzing[C]//Proc of Network and Distributed System Security Symposium. 2017.

[11] Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain[C]//Proc of ACM SIGSAC Conference on Computer and Communications Security. 2016: 1032-1043.

[12] Wang Junjie, Chen Bihuan, Wei Lei, et al. Skyfire: data-driven seed generation for fuzzing[C]//Security and Privacy. 2017: 579-594.

[13] Yoshida Y. Markov chains with a transition possibility measure and fuzzy dynamic programming[J]. Fuzzy Sets and Systems, 1994, 66(1): 39-57.

[14] Mohamed M A, Gader P. Generalized hidden Markov models I: theoretical frameworks[M]. [S.l.]: IEEE Press, 2000.

[15] Yan S, Wang Ruoyu, Christopher S, et al. SOK: (State of) the art of war: offensive techniques in binary analysis[C]//Security and Privacy. 2016.

[16] Cifuentes C, Hoermann C, Keynes N, et al. BegBunch: benchmarking for C bug detection tools[C]//Proc of International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM Press, 2009.

[17] Wang Liwei, Hong Yong. Study on convergence of genetic algorithm[J]. Chinese Journal of Computers, 1996, 19(10): 794-797.

Note: Figure translations are in progress. See original paper for figures.