Hindawi
Security and Communication Networks
Volume 2022, Article ID 1505842, 17 pages
https://doi.org/10.1155/2022/1505842

Research Article

GSA-Fuzz: Optimize Seed Mutation with Gravitational Search Algorithm

Mingmin Lin,[1] Yingpei Zeng,[1,2] Ting Wu,[1,3] Qiuhua Wang,[1] Linan Fang,[1] and Shanqing Guo[4]

[1]*School of Cyberspace, Hangzhou Dianzi University, Hangzhou 310000, China*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210000, China*
[3]*Hangzhou Innovation Institute, Beihang University, Hangzhou 310051, China*
[4]*School of Cyber Science and Technology, Shandong University, Jinan 250000, China*

Correspondence should be addressed to Yingpei Zeng; yzeng@hdu.edu.cn

Received 28 September 2021; Revised 17 February 2022; Accepted 26 June 2022; Published 15 July 2022

Academic Editor: Zhiyuan Tan

Mutation-based fuzzing is currently one of the most effective techniques to discover software vulnerabilities. It relies on mutation strategies to generate interesting seeds. As a state-of-the-art mutation-based fuzzer, AFL follows a mutation strategy with high randomization, which uses randomly selected mutation operators to mutate seeds at random offsets. Its strategy may ignore some efficient mutation operators and mutation positions. Therefore, in this paper, we propose a solution named GSA-Fuzz to improve the efficiency of seed mutation strategy with the gravitational search algorithm (GSA). GSA-Fuzz uses GSA to learn the optimal selection probability distributions of operators and mutation positions and designs a position-sensitive strategy to guide seed mutation with learned distributions. Besides, GSA-Fuzz also provides a *flip* mode to calculate the efficiencies of the deterministic stage and indeterministic stage and implements switching between the two stages to further improve the efficiency of seed mutation. We compare GSA-Fuzz with the state-of-the-art fuzzers AFL, MOPT-AFL, and EcoFuzz on 10 open-source programs. GSA-Fuzz finds 145% more paths than AFL, 66% more paths than EcoFuzz, and 43% more paths than MOPT-AFL. In addition, GSA-Fuzz also outperforms other fuzzers in bug detection and line coverage.

1. Introduction

Mutation-based fuzzing is an efficient way to discover software vulnerabilities. Its efficiency highly depends on seed mutation strategies. Improving the efficiency is important; since the more interesting seeds it generates, the more likely it triggers bugs. In recent years, a lot of effort has been done to promote the efficiency of fuzzing, including improving code coverage [1–4], strengthening seed selection strategy [5–8], and combining it with other techniques [9–12].

High randomization is one of the factors that affects the efficiency of fuzzing. There are many ways to reduce randomization, and a good way is to use optimization algorithms to solve the problem spontaneously. Optimization algorithms [13, 14] can be used in various research fields, and some recent researches have made breakthroughs [15–18]. What they have in common is that they all use the knowledge of optimization algorithms or machine learning to optimize their problems. Thus, optimization theories have good implications for reducing the randomization of fuzzing.

Popular mutation-based fuzzer AFL [19] follows such a mutation strategy to generate seeds, which is to use *randomly selected* operators to mutate seeds at their *random positions*. It always ignores efficient operators and mutation positions in seed mutation. Therefore, some solutions are proposed. In order to improve the selection of mutation operators, MOPT [20] utilizes the PSO (particle swarm optimization) to learn the optimal probability distribution of operators and uses learned distribution to guide the selection of operators. MOPT's particle swarm model uses multiple particles in each swarm to learn operator efficiencies, and

each particle represents the selection probability of an operator. Therefore, MOPT can be viewed as a process of searching for the optimal solution in a high-dimensional space. However, it has been proven that PSO is inefficient in exploring the optimal solution in a high-dimensional search space, and its accuracy and efficiency are lower than some similar algorithms such as GSA (gravitational search algorithm) [21]. Therefore, it may help AFL find efficient operators more quickly by using a more suitable optimization algorithm. In order to reduce the effect of randomness on mutation positions, there are also some methods to improve the efficiency of seed mutation by studying mutation positions. For instance, FairFuzz [4] uses a small number of mutation experiments to find the key mutation positions of seeds and keeps them from being mutated, which reduces the fuzzing of high-frequency paths. But the identified mutation positions are only a small part of each seed, and the fuzzer may ignore the effects of other mutation positions. Godefroid et al. [22] consider PDF documents as input sequences and use the recurrent neural network (RNN) to train a large number of input sequences to get a model that can generate new PDF documents with high coverage. However, the RNN scheme is just a specific solution for PDF documents that may not apply to other kinds of inputs.

In this paper, we propose a comprehensive solution GSA-Fuzz to improve the random scheduling scheme of mutation operators and mutation positions. It regards the selection of operators and mutation positions as GSA optimization problems. Since GSA has fast optimization speed and outstanding effect of convergence, it can quickly find suitable selection probability distributions of mutation operators and mutation positions, and continuously optimize them in fuzzing. Besides, GSA-Fuzz also applies a position-sensitive strategy to systematically guide the seed mutation and thus improves the fuzzing performance significantly.

When scheduling the selection of operators, supposing there is an n-dimensional space, GSA-Fuzz uses some particles to represent the probability distributions of operators. Each particle has a coordinate $(x_1, x_2, x_3, \ldots, x_d)$, which represents the particle's position in search space. Its value of each dimension x_i $(0 \le i \le d)$ represents the probability of the i-th operator. Particles keep moving to find the optimal position in space, and GSA-Fuzz also provides a *flip* mode to accelerate this exploration process. Also, in scheduling the selection of mutation positions, GSA-Fuzz divides each seed into multiple segments and every segment has a mutation probability. The fuzzer regards mutation probabilities of segments as particles to learn segment efficiencies. When GSA-Fuzz finishes optimization, the more efficient particle will get a greater mutation probability, and the fuzzer would mutate the corresponding segment more frequently in later fuzzing.

In order to efficiently apply the learned distribution to the seed mutation, GSA-Fuzz provides a position-sensitive strategy. When executing the strategy, GSA-Fuzz first chooses a mutation position according to the learned mutation probabilities of segments and then uses suitable selection probability distribution of operators to mutate the seed.

GSA-Fuzz is implemented on top of AFL. In order to test its performance, we evaluate it on ten open-source programs and compare it with three state-of-the-art fuzzers AFL [19], MOPT-AFL [20], and EcoFuzz [23]. The results show that GSA-Fuzz triggers 145% more paths than AFL, 66% more paths than EcoFuzz, and 43% more paths than MOPT-AFL. Also, GSA-Fuzz finds 275% more crashes than AFL, 150% more crashes than EcoFuzz, and 114% more crashes than MOPT-AFL. In addition, the line coverage of GSA-Fuzz is almost 20% higher than the other three fuzzers in the best case.

In summary, we have made the following contributions to this paper:

(i) We applied GSA to learn the selection probability distributions of mutation operators and mutation positions and designed a new position-sensitive strategy to guide seed mutation.

(ii) We designed a *flip* mode to calculate the efficiencies of the deterministic stage and indeterministic stage and used them to guide the switching between these two stages.

(iii) We implemented a prototype of GSA-Fuzz and used it to fuzz 10 open-source programs and compared it with three state-of-the-art fuzzers. The results showed that GSA-Fuzz outperformed other fuzzers. We published our code at https://github.com/lmm-1997/GSAFuzz.

2. Background

2.1. American Fuzzy Lop. AFL [19] is a popular mutation-based fuzzer that uses a new type of instrumentation and genetic algorithm to trigger program vulnerabilities. It is the ancestor of many mutation-based fuzzers such as LibFuzzer [24], VUzzer [25], and BFF [26]. As shown in Table 1, it has a set of mutation operators that define how to mutate a seed. When AFL works, it applies these operators to mutate the seed and uses the mutated seed as an input to execute the program under test. In terms of mutating seeds, it has two mutation stages, and their transformation is shown in Figure 1.

2.1.1. Deterministic. In this stage, mutation operators are selected to mutate the seed in a fixed order. AFL determines the mutation times of the seed according to its length, and this stage usually needs much execution time. Besides, the deterministic stage is only enabled once for a seed.

2.1.2. Indeterministic. This stage consists of the havoc step and splice step. When the deterministic stage is skipped or is over, AFL will run the havoc step and start to select mutation operators to mutate the seed at random offsets until it cannot generate a new interesting seed or the assigned mutation times of the seed are over. Once there is no interesting seed generated in the havoc step, AFL will start the splice step, where it randomly breaks two seeds from the seed pool and splices them into a new seed. Then the fuzzer returns to the havoc step and mutates the spliced seed. For determining the

Table 1: Defined mutation operators in AFL.

Name	Type	Function
BITFLIP	bitflip1/1 bitflip2/1 bitflip4/1 bitflip8/8 bitflip16/8 bitflip32/8	Randomly flip some bits of a seed
ARITHMETIC INC/ DEC	arith8/8 arith16/8 arith32/8	Perform integer addition or subtraction mutation on some bytes of the seed
INTERESTING VALUES	interest8/8 interest16/8 interest32/8	Replace some bytes in the seed with interesting values
DICTIONARY STUFF	user_extras(over), user_extras(insert) auto extras(over)	Replace tokens provided by the user or detected by the system into the seed
RANDOMLY BYTE	add to byte subtract from byte	Randomly select a byte and add or subtract a value
DELETE BYTES	delete bytes	Randomly delete some bytes
CLONE BYTES	clone bytes	Randomly clone some bytes and insert them into the seed randomly clone some bytes
OVERWRITE BYTES	overwrite bytes	And overwrite them into the seed

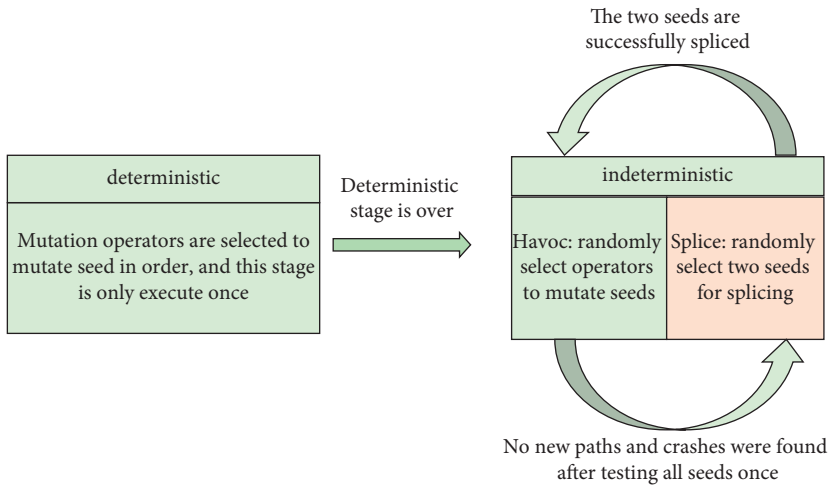

Figure 1: Two main mutation stages in AFL.

mutation times of the seed, it gives a score to the seed according to its execution speed, execution depth, consumed time of generating seeds, and so on. The mutation times of the seed are determined by its score. The higher score a seed gets, the more mutation times it gains.

2.2. Mutation Positions of a Seed Affect Operator Efficiencies. In general, a seed usually has its key part that can generate a large number of new interesting seeds. For example, the head of a protocol seed is one of its core parts and correct mutation at this part may trigger lots of paths for programs such as tcpdump because modifying the head of a protocol can easily change it into another protocol. We use AFL to count the discovered paths in different segments of seeds and evaluate the efficiencies of some operators in different segments. In our testing, we divide each seed into 5 segments to distinguish different mutation positions. As shown in Figure 2, AFL discovers a different number of paths in segments when it fuzzes sass. For example, segment 1 finds more than 2,500 paths, while segment 5 finds less than 1,500 paths, and the other 3 segments each find nearly 2,000 paths. It indicates that mutation positions of seeds affect the efficiency of fuzzing. Besides, we also study the efficiencies of operators in different segments. Figure 3 shows the discrepancy among operators' contributions in different segments. Without considering operator *extra overwrite*, operator *clone bytes*

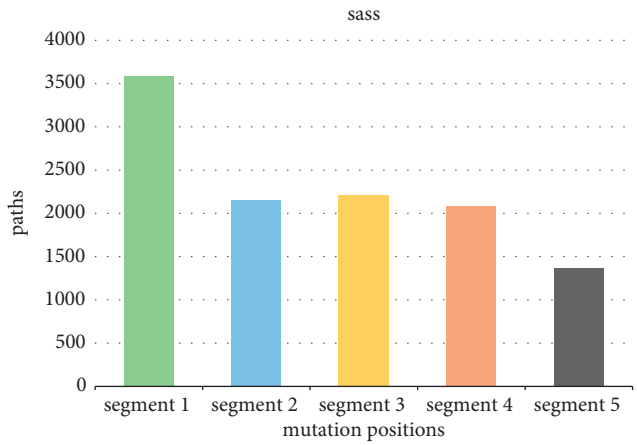

Figure 2: The discovered paths of each segment when AFL fuzzes sass.

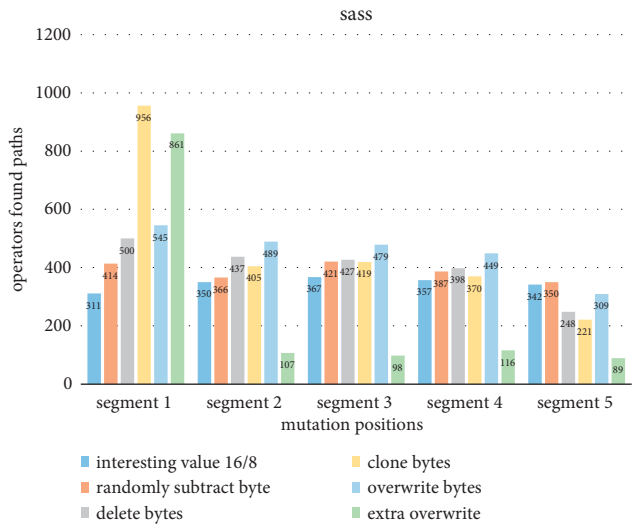

Figure 3: The efficiency of mutation operators in different segments.

find the most 956 paths in segment 1, which are twice as many as other operators. In segments 2–4, its found paths are similar to other operators. But, in segment 5, its found paths are less than other operators. Therefore, we believe that mutation positions may also have different sensitivities to operators.

After the above analysis, we find that mutation operators and mutation positions both have impacts on the performance of fuzzing. However, AFL only uses a random scheduling scheme to treat them, which is inefficient in generating interesting seeds.

3. Overview of GSA-Fuzz

In order to solve the existing problems, we present GSA-Fuzz to make AFL work more efficiently. To be exact, GSA-Fuzz improves the random scheduling scheme of AFL from two aspects. First, it regards a seed as several segments and uses GSA to learn the selection probability distribution of operators and the selection probability distribution of segments. Second, it applies a position-sensitive strategy to guide seed mutation.

In this section, we detailedly introduce GSA in Section 3.1. GSA-Fuzz models the selection of mutation operators and mutation positions as optimization problems of particles in GSA space. It means that we establish two models in GSA-Fuzz. Since the two models have the same framework, we only take the optimization of operators as an example to introduce some important concepts of GSA-Fuzz in Sections 3.2 and 3.3. After that, we simply introduce how GSA-Fuzz works to optimize fuzzing with GSA in Sections 3.4 and 3.5.

3.1. Gravitational Search Algorithm (GSA). GSA is a heuristic algorithm based on Newton's law of gravitation, which is used to solve optimization problems in multi-dimensional spaces. It assumes that there are particles in the space and each particle represents a solution to the optimization problem. At the end of each optimization, all particles will move toward the optimal particle. The interaction among particles is shown in Figure 4. We can see 4 particles in space, and the size of a particle represents its mass. M_1 receives 3 gravitational forces that come from M_2, M_3, and M_4, and their resultant force is F_1. That is, M_3 has the biggest mass, so M_1 tends to move to M_3 rather than moving in the directions of other particles.

GSA can be regarded as an independent mass system, which obeys Newton's law of gravitation and Newton's law of motion. To be more precise, GSA follows two rules:

(i) *Law of gravitation*: Each particle attracts every other particle in space, and the gravitational force between two particles is proportional to their masses and inversely proportional to the distance between them

(ii) *Law of motion*: The current velocity of a particle is equal to the sum of the part of the current velocity and the variation between the last velocity and the current velocity

There are several scientific formulas based on the two laws (considering there are N particles in GSA space):

$$X_i = \left(x_i^1, \ldots, x_i^d, \ldots, x_i^n\right), \tag{1}$$

where X_i represents the position of the i-th particle in GSA space and x_i^d represents the position of the i-th particle in the d-th dimension.

$$m_i(t) = \frac{f_i(t) - \max(f(t))}{\min(f(t)) - \max(f(t))}, \tag{2}$$

$$M_i(t) = \frac{m_i(t)}{\sum_{j=1}^{N} m_j(t)}, \tag{3}$$

where $f_i(t)$ represents fitness value of the i-th particle, $m_i(t)$ represents the gravitational mass of the i-th particle, and $M_i(t)$ represents inertial mass of the i-th particle at time t.

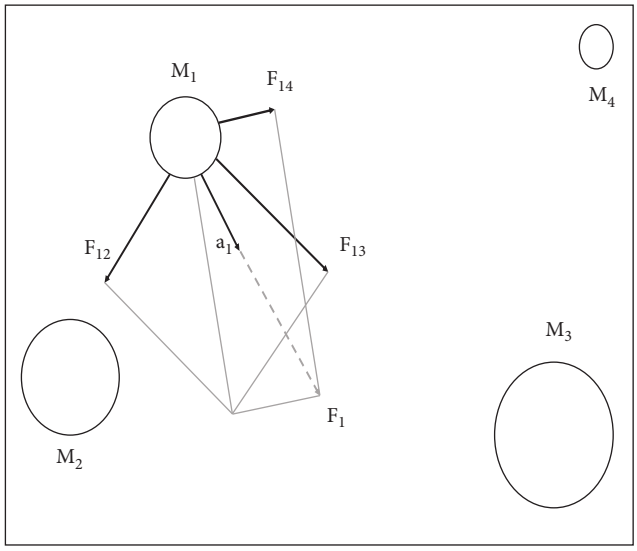

Figure 4: Particles attract each other in GSA space [21].

$$F_{ij}^{d}(t) = \frac{GM_i(t)M_j(t)}{R_{ij}(t)}\left(x_i^d(t) - x_j^d(t)\right),$$
$$F_i^d(t) = \sum_{j!=i}^{j\epsilon K_{\text{best}}} \text{rand}_i F_{ij}^d(t), \tag{4}$$

where $F_{ij}^{d}(t)$ represents received gravitational force of the i-th particle from the j-th particle in the d-th dimension at time t, G is a gravitational constant, R is Euclidean distance between the i-th particle and the j-th particle, K_{best} is the set of top K particles with the best fitness values and masses, and rand_i is a random number belongs to $(0, 1)$.

$$a_i^d(t) = \frac{F_i^d(t)}{M_i(t)}, \tag{5}$$

where $a_i^d(t)$ is the acceleration of the i-th particle in the d-th dimension at time t.

Based on these calculations, the positions of particles are updated by following two rules, considering one round of optimization as a time unit:

$$v_i^d(t+1) = \text{rand}_i \times v_i^d(t) + a_i^d(t),$$
$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1), \tag{6}$$

where v_i^d represents the velocity of the i-th particle in the d-th dimension.

Therefore, particles can approach the heaviest particle after being optimized with GSA, which means that they can move closer to the optimal position in space. Compared with the PSO algorithm, GSA has better convergence and faster optimization speed [21]. In addition to these two classic optimization algorithms, some original optimization algorithms also have an excellent performance like [14, 27], and some researches scheduled by them have also been greatly improved [15, 17, 27].

3.2. Particles. A particle in GSA-Fuzz is a probability distribution that contains the current probabilities of all operators. We use a two-dimensional array $\text{position}[x][d]$ to represent the position of the current particle in GSA space (each position represents a probability distribution of operators) and use $v[x][d]$ to save the current particle's velocities in all dimensions, where x is the number of particles and d is the dimension of the space. The particle also has other attributes, including mass, force, and acceleration, which are all related to its fitness value.

3.3. Fitness Value. The fitness value is the core concept of GSA; it determines the mass and inertia of the particle. In Newton's law of gravitation, the bigger mass one object owns, the stronger inertia and stability it has. Thus, in GSA-Fuzz, a particle's fitness value is determined by its efficiency in generating interesting seeds. That is, an efficient particle will get a larger fitness value, and it also has a bigger mass and stronger inertial. We use the following rule to calculate the fitness value of each particle:

$$f_i(t) = \frac{P_i(t)}{C_i(t)}, \tag{7}$$

where $P_i(t)$ and $C_i(t)$ are the number of generated seeds of the i-th particle and the mutation times of the i-th particle at time t.

When fitness values of all particles are determined, their gravitational masses (m) and inertial masses (M) can be calculated. The calculations are shown in equations (2) and (3). After each particle has an inertial mass, GSA-Fuzz can calculate the forces and accelerations of particles in all dimensions. We use equations (4), (6), and (10) to update related parameters.

$$F_i^d(t) = \sum_{j!=i} \text{rand}_i F_{ij}^d. \tag{8}$$

Note that in our repeated experiments, we found that the value of K_{best} has almost no effect on our learning process. Without this parameter, the optimization process will become even more unlikely to fall into the local optimum. Thus, we removed the K_{best} parameter in the GSA-Fuzz.

Based on these values, the positions of particles can be updated. More specifically, for the d-th dimension of the i-th particle at the end of time t, we update particles' velocities and positions for the time $t + 1$ as follows:

$$v_{t+1}[i][d] = \text{rand}_i \times v_t[i][d] + a_t[i][d],$$
$$\text{position}_{t+1}[i][d] = \text{position}_t[i][d] + v_{t+1}[i][d]. \tag{9}$$

3.4. Learning Selection Probability Distributions of Operators and Segments with GSA. In learning the selection of operators, each particle represents a probability distribution of operators. GSA-Fuzz applies each particle to guide seed mutation and keeps evaluating particle efficiencies in generating interesting seeds. Note that the time for each particle to guide the seed mutation is equal. Once all particles are

executed, GSA-Fuzz would optimize the positions of particles according to their efficiencies and use the updated particles to guide the selection of operators in later fuzzing. Finally, GSA-Fuzz can get the optimal probability distribution of operators through the repeated learning process.

Similarly, in learning the selection of mutation positions, GSA-Fuzz divides a seed into some segments and regards the mutation probability of each segment as a particle. Note that too many segments will greatly reduce the learning efficiency of GSA, and too few segments do not reflect the discrepancy among different mutation positions. In our experiments, we found that the most suitable number of segments for mutation is five. At the beginning of fuzzing, GSA-Fuzz uses a uniform distribution to select mutation positions of seeds, and it records the generated seeds of each segment. After a period, GSA-Fuzz updates the uniform distribution with GSA, and the segment that has generated more seeds will get a greater mutation probability. The learning of segments is also a repeated process, and it is synchronized with the learning process of mutation operators.

3.5. Mutating Seeds with Position-Sensitive Strategy. Mutation-based fuzzers usually follow a random scheme to select mutation operators and mutation positions to guide seed mutation. We have shown that both of them have different efficiencies in generating interesting seeds in Section 2.2. In order to improve the random mutation scheme, we design a position-sensitive strategy to improve the efficiency of seed mutation, which may give efficient operators and efficient mutation positions more chances for mutation.

In the strategy, GSA-Fuzz uses learned probability distributions of operators and segments to guide seed mutation. To be precise, it selects a mutation position according to the distribution of segments first and then applies suitable distribution of operators to mutate the seed. This strategy works most of the time in fuzzing, and we detailedly introduce it in Section 4.4.

4. Implementation of GSA-Fuzz

4.1. Main Framework of GSA-Fuzz. As shown in Figure 5, the main workflow of GSA-Fuzz is marked as yellow arrows. GSA-Fuzz mainly consists of *GSA initialization*, *GSA fuzzing*, *position-sensitive fuzzing*, *GSA updating*, and *p-segment updating* modules. In these modules, the *GSA fuzzing* and the *position-sensitive fuzzing* modules are fuzzing parts, and the other three modules contain parameter processing. We briefly introduce the modules as follows and introduce some modules in detail in the following subsections.

In general, GSA-Fuzz is a cyclic fuzzing process composed of multiple modules. In the process, the *GSA initialization* module is only executed once and the other four modules form a loop to optimize the selection probability distributions of operators and segments. Note that GSA-Fuzz is implemented in the indeterministic stage of AFL, if it is not set to skip the deterministic stage, it will enable the *flip* mode to evaluate the efficiencies of the two stages. The *flip* mode keeps working in the *GSA fuzzing* module and the *position-sensitive fuzzing* module. We introduce the *flip* mode in detail in Section 4.5.

In order to distinguish two particle models of learning mutation position distribution and learning operator distribution, we mainly use the word "particle" in this and following subsections to only represent the particle model of learning operator distribution and use the word "segment" directly to represent the particle model of learning segment distribution.

Assuming that GSA-Fuzz is set to skip the deterministic stage, it first executes the *GSA initialization* module to initialize the values of particles' attributes. Once all particles are initialized, GSA-Fuzz enters the *GSA fuzzing* module and employs particles to select mutation operators to mutate seeds in turn. Meanwhile, it evaluates the efficiencies of particles and the efficiencies of segments in generating interesting seeds. So it can get the fitness values of segments and further use them in the *p-segment updating* module. The *p-segment updating* is the module to update the probability distributions of operators and segments. Then, GSA-Fuzz executes the *position-sensitive fuzzing* module and uses the position-sensitive strategy to guide seed mutation. After this module, it updates the values of particles' attributes in the *GSA updating* module and returns to the *GSA fuzzing* module for the next round of optimization.

4.2. GSA Initialization. GSA-Fuzz executes the *GSA initialization* module to initialize GSA parameters when it starts. More specifically, it (1) sets the initial position of each particle with a random value and normalizes all dimensions in particles to 1, (2) sets all particles' gravitational masses to 1, and (3) sets initial velocities of dimensions in particles to 0. The reason for setting the initial masses of particles to 1 and the initial velocities of dimensions to 0 is to ensure that the learning process of all particles is fair. This module is only executed once.

4.3. GSA Fuzzing. After GSA parameters are initialized, GSA-Fuzz executes the *GSA fuzzing* module. In this module, the fuzzer employs particles to guide seed mutation in turn. Meanwhile, it also calculates the fitness values of particles and segments. In order to get these values, GSA-Fuzz records four parameters in the module: (1) the number of generated seeds of each particle, (2) the number of generated seeds of each segment, (3) the mutation times of each particle, and (4) the mutation times of each segment. Each particle's fitness value is calculated by (1) dividing (3), and each segment's fitness value is calculated by (2) dividing (4). After GSA-fuzz gets all fitness values, it enters the *p-segment updating* module to update related parameters for the *position-sensitive fuzzing* module.

4.4. Position-Sensitive Fuzzing. The *Position-sensitive fuzzing* module is the core module of GSA-Fuzz. When AFL mutates a seed, it selects operators first and then selects a mutation position of the seed to mutate. In this module, GSA-Fuzz selects a mutation position of the seed according to the

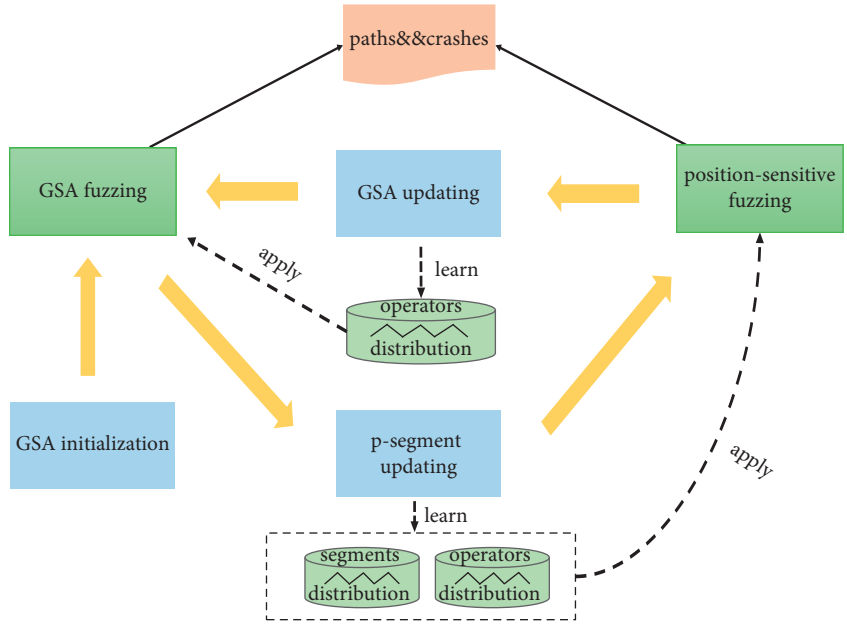

Figure 5: The framework of GSA-Fuzz.

learned selection probability distribution of segments first and then selects appropriate operators for mutation. The mutation strategy of AFL and GSA-Fuzz are shown in Table 2, and the details of the position-sensitive strategy are shown as follows.

Figure 6 shows how the position-sensitive strategy works to guide seed mutation. More specifically, (1) fuzzer selects a seed from the seed pool and divides it into 5 segments, (2) fuzzer selects a mutation position of the seed according to the selection probability distribution of segments, and judges which segment it belongs to, (3) fuzzer uses operator probability distribution of the segment to guide seed mutation, and (4) fuzzer saves the interesting seed and judges if mutation energy of the seed is over. If energy is not over, the fuzzer will return to step (2), and if energy is over, the fuzzer will return to step (1) and start the next loop. Note that mutation energy is the assigned mutation times of each seed by GSA-Fuzz.

4.5. Flip Mode. GSA-Fuzz is built in the indeterministic stage of AFL. When the deterministic stage is enabled, GSA-Fuzz will be used only after a long time. However, the deterministic stage is also very useful under some conditions. For example, its speed of finding new paths is faster than the indeterministic stage in some periods, and AFL has more opportunities to trigger the potential bugs of target programs after performing a complete deterministic stage. In order to utilize the deterministic stage efficiently, GSA-Fuzz keeps evaluating the efficiencies of the deterministic stage and indeterministic stage to implement smart switching.

When the deterministic stage is enabled, if GSA-Fuzz fails to find a new path within a period, it will directly enter the indeterministic stage. After a period of fuzzing, the fuzzer backs to the deterministic stage according to the efficiencies of the two stages. We calculate their efficiencies by the following equations:

$$E_{\text{det}} = \frac{F_{\text{det}}}{T_{\text{det}}}, \tag{10}$$

$$E_{ind} = \frac{F_{ind}}{T_{ind}}, \tag{11}$$

where F_{det} and F_{ind} represent the number of generated seeds in the deterministic stage and indeterministic stage, respectively, and T_{det} and T_{ind} represent the execution time of the deterministic stage and indeterministic stage, respectively.

Compared with the *pacemaker* mode of MOPT-AFL, our *flip* mode does not require users to specify the time of the deterministic stage, and it may dynamically switch between the two stages instead of always first executing the deterministic stage and then switching to the indeterministic stage. Also, the *pacemaker* mode will never reuse the deterministic stage once it enters the indeterministic stage.

We evaluate the *flip* mode in Section 5.6. It has the following advantages:

(i) *Flip* mode may save a lot of time by skipping the deterministic stage, and speed up the convergence of GSA particles

(ii) *Flip* mode may accelerate GSA-Fuzz to enter the position-sensitive fuzzing module

(iii) *Flip* mode evaluates the efficiencies of the deterministic stage and indeterministic stage in real-time and implements a smart switching between the two stages

5. Evaluation

5.1. Experiment Configuration

5.1.1. Target Programs. We evaluate GSA-Fuzz on 10 open-source programs. The reason for choosing these programs is that most of them were evaluated by existing AFL-type

Table 2: The differences in mutation strategy between AFL's havoc stage and GSA-fuzz's position-sensitive module.

AFL's mutation strategy in havoc stage	GSA-fuzz's mutation strategy in position-sensitive module
(1) Select a mutation operator by a uniform distribution	**(1) Segment a seed**
	(2) Choose a mutation position according to the distribution of segments
(2) Choose a random mutation position of the seed	**(3) Judge which segment the position belongs to**
	(4) Use the operator distribution of the segment to select an operator
(3) Mutate the position with the selected operator	(5) Mutate the position with the selected operator

Bold values represent that the difference between GSA-Fuzz's position-sensitive module and AFL's havoc stages.

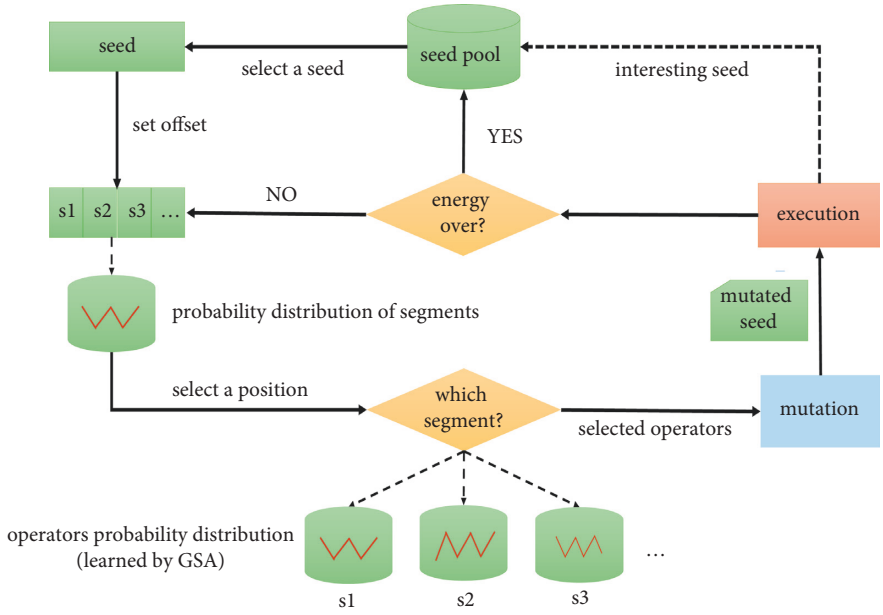

Figure 6: The workflow of the position-sensitive strategy in seed mutation.

Table 3: Target programs.

Target	Version	Format	Command
Sass	libsass-3.6.4	scss	./sassc @@
pngtest	libpng-1.6.37	png	./pngtest @@
cxxfilt	binutils-2.35.1	txt	./cxxfilt
pdfimages	xpdf-4.02	pdf	./pdfimages @@/dev/null
pdftotext	xpdf-4.02	pdf	./pdftotext @@/dev/null
Server	openssl-1.1.1i	elf	./server
objdump	binutils-2.35.1	elf	./objdump -xsSD @@
nm-new	binutils-2.35.1	elf	./nm-new -A -a -l -S -s -C @@
infotocap	ncurses-6.2	txt	./infotocap @@
tcpdump	tcpdump-4.9.3	pcap	./tcpdump -ee -vv -nnr @@

fuzzers [4, 20], and their detailed information is shown in Table 3. All experiments are executed without dictionaries.

5.1.2. Baseline. We compare GSA-Fuzz with three AFL-type fuzzers, AFL [19], MOPT-AFL [20], and EcoFuzz [23].

5.1.3. Platform. All the experiments are finished on a 64 bit machine with 40 cores (Intel(R) Xeon(R) CPU E5-2640 v4 2.40 GHz) and 62 GB RAM. The operating system of the machine is Ubuntu 20.04. We fuzz all target programs for 24 hours and repeat each experiment 5 times.

5.1.4. Evaluation Metrics. Like other fuzzers [4, 20, 23], we evaluate the performance of fuzzers using three metrics, which are unique paths, line coverage, and unique crashes. Besides, we also compare the CPU time that GSA-Fuzz and MOPT-AFL consume in probability optimization. We use *afl-cov* [28] to measure line coverage of target programs and count the number of unique crashes by running crashes on recompiled target programs with *AddressSanitizer* [29] and *stack hash* [30].

5.2. Paths Analysis. As shown in Figure 7 and the left half of Table 4, GSA-Fuzz outperforms the other three AFL-type fuzzers on most target programs except nm. On libpng, tcpdump, openssl, and infotocap, the performance of GSA-Fuzz is better than the other three fuzzers. On pdftotext, GSA-Fuzz finds 13689 paths, which are 134% more than AFL, 127% more than MOPT-AFL, and 275% more than EcoFuzz. On pdfimages and objdu-mp, the discovered paths of GSA-Fuzz are also far ahead of the other three fuzzers. The reason for GSA-Fuzz has such good performance is that the position-sensitive strategy generates more interesting seeds, and these seeds cover more lines of the programs. On nm,

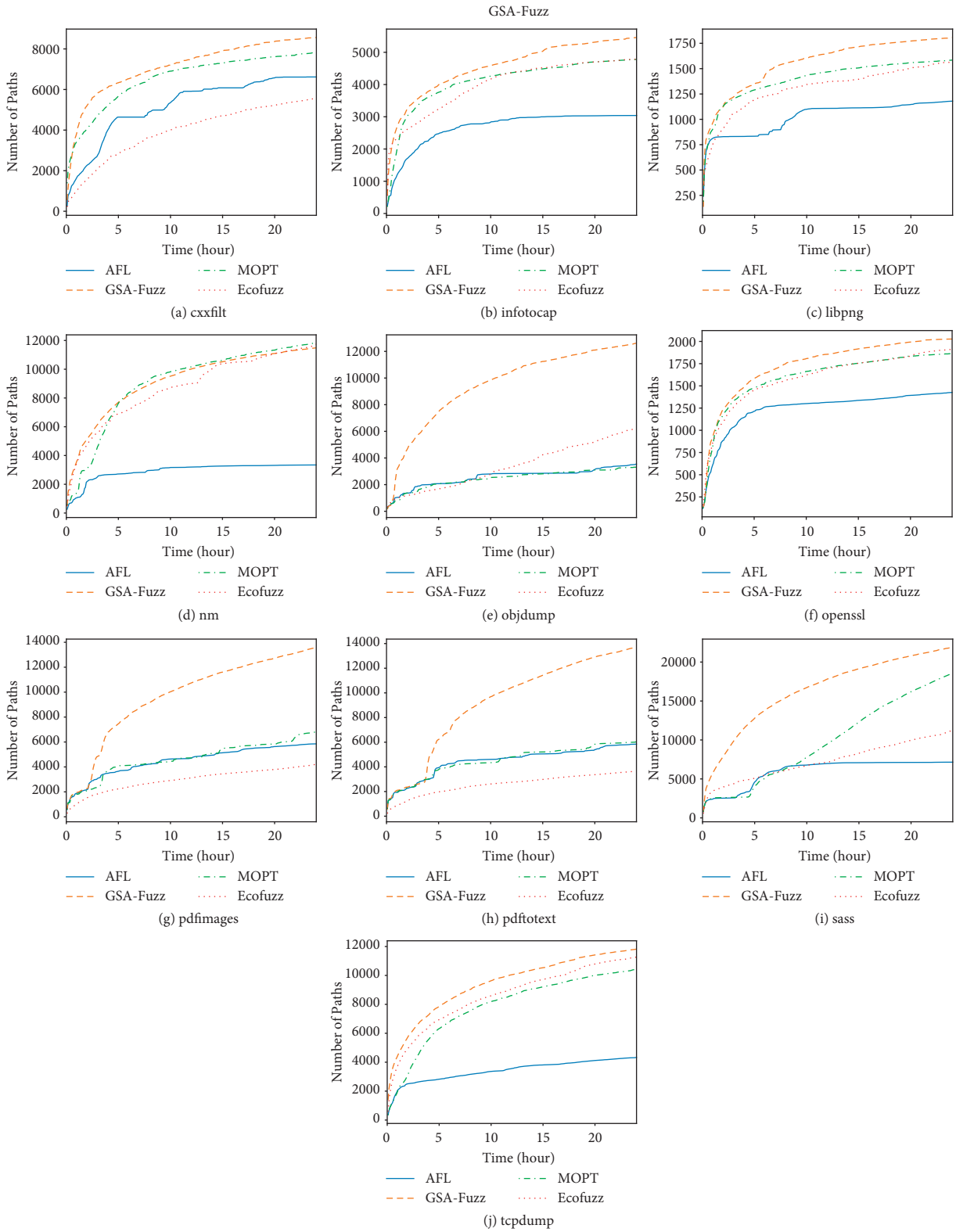

Figure 7: Average paths that AFL, MOPT-AFL, EcoFuzz, and GSA-Fuzz found over 5 runs in 24 hours.

GSA-Fuzz loses to MOPT-AFL at a slight disadvantage. This may be due to the fact that the selected seeds have very uniform key information, which leads to deviations in the learned mutation probabilities of segments. Since the discrepancy between the discovered paths of GSA-Fuzz and MOPT-AFL is tiny and the performance of GSA-Fuzz is superior to MOPT-AFL on other programs, we can still believe that GSA-Fuzz is the more effective one.

Table 4: The average of three evaluation metrics on 10 target programs.

Targets	Paths/unique crashes				Line coverage			
	AFL	MOPT-AFL	EcoFuzz	GSA-Fuzz	AFL (%)	MOPT-AFL (%)	EcoFuzz (%)	GSA-Fuzz (%)
cxxfilt	6,709/0	7,742/0/	5,674/0	**8,498/0**	18.61	18.75	18.63	**22.44**
infotocap	2,851/0	4,794/0	4,759/0	**5,458/0**	4.94	5.18	5.27	**5.28**
libpng	1,180/0	1,584/0	1,563/0	**1,807/0**	68.02	**69.04**	68.30	68.66
nm	3,344/0	**11,813/0**	11,740/0	11,488/0	40.62	**41.82**	41.06	41.54
objdump	3,544/0	3,308/0	6,260/0	**13,219/0**	42.0	41.8	41.9	**45.8**
openssl	1,357/0	1,755/0	1,867/0	**2,027/0**	23.60	24.25	24.16	**24.53**
pdfimages	5,851/2	6,801/3	42,16/4	**13,588/6**	17.85	16.92	15.58	**21.33**
pdftotext	5,842/2	6,023/2	3,647/2	**13,689/5**	19.14	19.00	17.92	**21.42**
sass	7,150/0	18,216/2	11,211/0	**21,881/4**	43.89	51.15	41.28	**51.76**
tcpdump	4,330/0	10,413/0	11,266/0	**11,822/0**	15.99	26.60	23.43	**27.05**

Furthermore, we can also find that GSA-Fuzz has the fastest speed of discovering paths in the first five hours, and this is because GSA-Fuzz can quickly cover all parts of a seed. Once a certain part of the seed has been mutated many times, the efficiency of GSA-Fuzz in discovering new paths will become lower. Therefore, GSA-Fuzz has the best performance in the first few hours of fuzzing, and its efficiency gradually decreases afterward.

5.3. Coverage Analysis. GSA-Fuzz also shows its good performance in line coverage. From the right half of Table 4, we can find that GSA-Fuzz has higher line coverages on most target programs. For instance, GSA-Fuzz covers 22.44% of lines in cxxfilt while AFL covers 18.61%, MOPT-AFL covers 18.75%, and EcoFuzz covers 18.63%. On objdump, GSA-fuzz has 21.42% line coverage, which is more than 19.14% of AFL, 19% of MOPT-AFL, and 19.92% of EcoFuzz. On pdfimages, tcpdump, and pdftotext, GSA-Fuzz also has an obvious advantage compared with the other fuzzers. The reason is that GSA-Fuzz generates much more interesting seeds than the other three fuzzers on these three programs. On infotocap, openssl, and sass, GSA-Fuzz has a similar performance to AFL, MOPT-AFL, and EcoFuzz. On nm, GSA-Fuzz loses to MOPT-AFL because it has not learned the optimal selection probability distributions of segments in two runs. On libpng, GSA-Fuzz generates more interesting seeds than MOPT-AFL, but its coverage is still lower than MOPT-AFL. The reason is that MOPT-AFL generates some seeds that trigger rare branches of libpng. Although GSA-Fuzz finds more paths, it does not detect a new search space of libpng.

5.4. Vulnerabilities Analysis. We test the performance of fuzzers in discovering vulnerabilities, and the results show that fuzzers do not find a vulnerability in most cases for 24 hours. But, in the programs that trigger vulnerabilities, GSA-Fuzz has the best performance. As shown in Table 4, GSA-Fuzz finds 15 vulnerabilities on pdfimages, pdftotext, and sass, while AFL finds 4 vulnerabilities, MOPT-AFL finds 7 vulnerabilities and EcoFuzz finds 6 vulnerabilities. Among the 15 vulnerabilities that are triggered by GSA-Fuzz, there are 9 stack-overflows, as well as 3 memory leaks and 3 SEGVs. In detail, there are 3 and 2 stack-overflows in pdftotext and sass, respectively, and 4 stack-overflows in pdfimages. Besides, GSA-Fuzz triggers each 1 memory leak and each 1 SEGV in pdftotext and pdfimages, and it also triggers 2 SEGVs in sass. Compared with GSA-Fuzz, the vulnerabilities triggered by AFL, MOPT-AFL, and EcoFuzz in pdfimages and pdftotext are all included in the triggered vulnerabilities of GSA-Fuzz. On sass, there are only MOPT-AFL and GSA-Fuzz trigger vulnerabilities, and GSA-Fuzz triggers more vulnerabilities than MOPT-AFL.

We take sass as an example to introduce discovered vulnerabilities in detail. In sass, GSA-Fuzz triggers 4 vulnerabilities, which are 2 stack-overflows and 2 SEGVs. For 2 stack-overflows, one occurs in the *has_real_parent_ref ()* method online 363 of *src/selectors.cpp*, and other one occurs in the *has_real_parent_ref ()* method online 549 of *src/selectors.cpp*. This method is in a deep recursion, and the frequent pushing of parameters is the cause of the vulnerabilities. Between 2 SEGVs, one SEGV occurs in the *src/ast.hpp*, and it is caused by a read memory access. The other SEGV is triggered by accessing an invalid address in *src/erreor_handling.hpp*.

In summary, for both crash type and crash number, the performance of GSA-Fuzz is better than AFL, MOPT-AFL, and EcoFuzz.

5.5. Statistical Analysis. In order to ensure the evaluation is comprehensive, we conduct a statistical analysis of our experiments. We use the p value to measure the difference between GSA-Fuzz and other fuzzers. As shown in Table 5, $p1$, $p2$, and $p3$ represent the difference between GSA-Fuzz with AFL, MOPT-AFL, and EcoFuzz. In Table 5, on total paths, $p1$ is smaller than 0.01 in all evaluations, which indicates that the found paths of GSA-Fuzz and AFL differ significantly. Then $p2$ is almost all less than 0.05 except on infotocap, nm, and tcpdump, which means there is no significant difference between the found paths of GSA-Fuzz and MOPT-AFL on these three evaluations. Furthermore, $p3$ is also smaller than 0.05 except on nm and tcpdump, indicating that GSA-Fuzz triggers almost the same paths as EcoFuzz on nm and tcpdump. Online coverage, both $p1$ and $p3$ are smaller than 0.05, which proves that GSA-Fuzz outperforms AFL and EcoFuzz. In most evaluations, $p2$ is smaller than 0.05 but not on libpng and nm, indicating that

Table 5: The p value in each evaluation.

Targets	Total paths			Line coverage		
	$p1$	$p2$	$p3$	$p1$	$p2$	$p3$
cxxfilt	$2.1 * 10^{-3}$	$4.4 * 10^{-2}$	$2.8 * 10^{-5}$	$3.5 * 10^{-5}$	$7.0 * 10^{-2}$	$3.1 * 10^{-3}$
infotocap	$2.6 * 10^{-3}$	$3.0 * 10^{-1}$	$3.2 * 10^{-2}$	$1.4 * 10^{-2}$	$3.1 * 10^{-2}$	$2.5 * 10^{-2}$
libpng	$2.9 * 10^{-5}$	$7.8 * 10^{-4}$	$2.6 * 10^{-2}$	$4.0 * 10^{-2}$	$1.3 * 10^{-1}$	$8.4 * 10^{-4}$
nm	$2.7 * 10^{-5}$	$2.9 * 10^{-1}$	$5.7 * 10^{-1}$	$8.5 * 10^{-3}$	$3.6 * 10^{-1}$	$2.7 * 10^{-2}$
objdump	$9.7 * 10^{-4}$	$1.0 * 10^{-3}$	$4.1 * 10^{-2}$	$2.6 * 10^{-6}$	$2.1 * 10^{-6}$	$5.7 * 10^{-3}$
openssl	$3.1 * 10^{-5}$	$8.5 * 10^{-3}$	$4.1 * 10^{-1}$	$4.1 * 10^{-4}$	$2.5 * 10^{-2}$	$4.9 * 10^{-2}$
pdfimages	$7.0 * 10^{-4}$	$1.7 * 10^{-3}$	$5.3 * 10^{-5}$	$4.5 * 10^{-3}$	$5.7 * 10^{-3}$	$2.4 * 10^{-4}$
pdftotext	$1.6 * 10^{-4}$	$1.8 * 10^{-4}$	$1.3 * 10^{-4}$	$8.7 * 10^{-4}$	$8.4 * 10^{-4}$	$3.3 * 10^{-4}$
sass	$1.2 * 10^{-6}$	$3.4 * 10^{-2}$	$3.9 * 10^{-4}$	$2.3 * 10^{-7}$	$8.1 * 10^{-3}$	$5.7 * 10^{-4}$
tcpdump	$1.7 * 10^{-3}$	$4.7 * 10^{-1}$	$4.1 * 10^{-1}$	$6.2 * 10^{-5}$	$1.2 * 10^{-1}$	$1.5 * 10^{-2}$

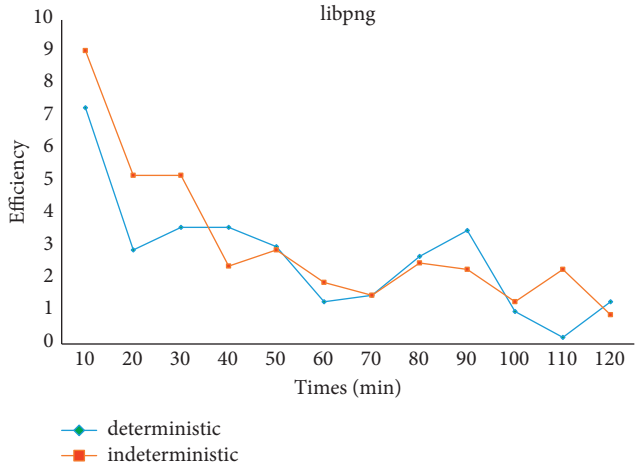

Figure 8: Efficiencies of the deterministic stage and indeterministic stage vary over time.

the performance of GSA-Fuzz is similar to MOPT-AFL on libpng and nm, and its performance is superior to MOPT-AFL on other evaluations.

In summary, the performance of AFL is steadily lower than GSA-Fuzz. Then MOPT-AFL performs as well as GSA-Fuzz in discovering paths on some programs such as infotocap and nm, and its performance of line coverage corresponds with its discovered paths. Finally, EcoFuzz finds many paths in some programs such as nm and openssl, but its line coverage is generally low in most programs.

5.6. Efficiencies of the Deterministic Stage and Indeterministic Stage in Flip Mode. In *flip* mode, GSA-Fuzz switches between the deterministic stage and indeterministic stage according to their efficiencies in triggering new paths. Because GSA-Fuzz enables the deterministic stage, it executes this stage first. If GSA-Fuzz does not find a new path or a crash within a specified time, it will automatically enter the indeterministic stage. We test the efficiencies of the deterministic stage and indeterministic stage in the same period on *libpng*. As shown in Figure 8, the efficiency of the deterministic stage is higher in five of the 12 periods than in the indeterministic stage, and its efficiency is lower in the other 7 periods. In real-world testing, for programs with many execution paths such as tcpdump, GSA-Fuzz may execute the switching less than three times in 24 hours. For programs with few execution paths such as libpng, GSA-Fuzz may finish the switching more than three times. The reason is that the efficiency of the indeterministic stage is always higher than the deterministic stage in complex programs. Therefore, we can see that it is reasonable to automatically switch between the two stages for better efficiency.

5.7. Efficiency of Learning Selection Probability Distribution of Operators. In order to verify the effectiveness of GSA in optimizing the selection probability distribution of operators, we compare GSA-Fuzz with MOPT-AFL in probability optimization. Note that GSA-Fuzz here is its test version GSA-Fuzz (only-GSA), which only uses GSA to schedule the selection of operators, and skips the *position-sensitive fuzzing* module. We test 2 programs for 4 hours and show the results in Table 6. In order to more intuitively reflect the changes in operator probabilities, we only list the initial probabilities of 6 operators and their optimized probabilities after the fuzzer has been running for 4 hours.

Because MOPT-AFL and GSA-Fuzz initialize the operator probabilities with random values, there are differences in initial operator probabilities. However, these will not affect the optimization of operator probabilities. As shown in Table 6, on sass, GSA-Fuzz executes 22 rounds of probability optimization, which means GSA-Fuzz has learned operator probabilities 22 times with GSA in 4 hours. Meanwhile, MOPT-AFL executes 13 rounds of optimization. On tcpdump, GSA-Fuzz executes 28 rounds of optimization while MOPT-AFL executes 23 rounds. These show that GSA-Fuzz can learn probability faster than MOPT-AFL. We can also notice that there is a big difference between the learned operator probabilities of GSA-Fuzz and MOPT-AFL at the 4th hour. This is because the period of first 4 hours is the preliminary stage of probability learning. When increasing the time of probability learning, the learned operator probabilities of GSA-Fuzz and MOPT-AFL will eventually become stable. Moreover, it is observed that GSA-Fuzz finds 9,534 paths in *sass* and 6,558 paths in *tcpdump* by applying learned operator probabilities while MOPT-AFL finds 9,023 paths in *sass* and 6,330 paths in *tcpdump*. These indicate that the learned operator probabilities of GSA-Fuzz may be closer to the true optimal operator probabilities than MOPT-AFL's.

Table 6: 4-hour optimization of operator probabilities in GSA-Fuzz and MOPT-AFL.

Target	Fuzzer	Total paths	Round	P (op1)	P (op2)	P (op3)	P (op4)	P (op5)	P (op6)
sass	MOPT-AFL	9023	Initial	0.05598028	0.05894883	0.06759142	0.04992040	0.05076231	0.06495813
			13th (at the 4th hour)	0.04769217	0.04805608	0.04505965	0.02786796	0.03500074	0.03858628
sass	GSA-Fuzz(only-GSA)	9534	Initial	0.06469484	0.05510837	0.04111614	0.07399413	0.04763995	0.02720640
			22th (at the 4th hour)	0.06606950	0.04685606	0.04810734	0.06218584	0.04928828	0.04485260
tcpdump	MOPT-AFL	6330	Initial	0.06531494	0.03806765	0.05559310	0.05916312	0.08056653	0.06429057
			23th (at the 4th hour)	0.04539392	0.03002448	0.03649274	0.04031799	0.05268548	0.04458380
tcpdump	GSA-Fuzz(only-GSA)	6558	Initial	0.07699334	0.08450678	0.06903984	0.07498587	0.01667453	0.02720640
			28th (at the 4th hour)	0.07331001	0.05669671	0.07097238	0.06373888	0.04904132	0.04360007

5.8. Convergence Studies of MOPT-AFL and GSA-Fuzz. In order to more intuitively reflect the difference between MOPT-AFL and GSA-Fuzz in learning the probability distribution of operators, we count the optimization times of each case and record the optimized probability distribution of operators in each optimization. Although there are more than a dozen mutation operators in AFL, their optimization processes are similar. Therefore, we only select operator bitflip1/1 to show its convergence study.

As shown in Figure 9, on one hand, fuzzers have different execution speeds in various programs, and the optimization times (the times of learning optimal selection probability distribution of operators) will also have a difference. For instance, on openssl, pdftotext, and pdfimages, both GSA-Fuzz and MOPT-AFL execute a few times of optimization and the probability of bitflip1/1 only has a slight change while not in other cases. On the other hand, MOPT-AFL has a more fluctuating learning process (such as cxxfilt and infotocap), and its range of probability variation is larger than GSA-Fuzz. On the contrary, GSA-Fuzz has a more detailed exploration process, which makes it work more stably by using a probability distribution that is close to the current optimal probability distribution.

5.9. CPU Time Comparaison between MOPT-AFL and GSA-Fuzz in Optimizing the Probability Distribution of Operators. We further compare the performance of MOPT-AFL and GSA-Fuzz in terms of CPU time consumption. In this subsection, each case has been tested for 4 hours and their consumed CPU time in optimization of probability distribution was recorded. Table 7 shows that the consumed CPU time of GSA-Fuzz is slightly more than MOPT-AFL in all cases. This is because GSA-Fuzz executed more optimization times than MOPT-AFL and it has more computations in each optimization. However, due to the magnitude of consumed CPU time being only in microseconds (10^6 s) for each optimization, the extra overhead in optimization has nearly no effect on the performance of GSA-Fuzz.

5.10. Efficiency of Learning Selection Probability Distribution of Positions. As shown in Section 2.2, mutation positions have different efficiencies in generating interesting seeds. Therefore, we use GSA to learn the selection probability distribution of mutation positions, and GSA-Fuzz gives more chances to the efficient parts in later fuzzing. In contrast, we compare GSA-Fuzz to itself that disenables position learning.

The results are shown in Table 8 and Figure 10. From Table 8, GSA-Fuzz changes mutation probabilities of segments when it enables position learning, which means that GSA-Fuzz has learned the mutation positions that are suitable to mutate. For instance, in sass, GSA-Fuzz has learned that segment 5 is more suitable to generate seeds, and its mutation probability increased by nearly 0.24. In tcpdump, the mutation probabilities of segment 1 and segment 2 increased by nearly 0.2 and 0.1, respectively. After applying the learned distribution to guide the seed mutation, the discovered paths of GSA-Fuzz are also increased. As shown in Figure 10, the discovered paths increased 5% on sass and 8% on tcpdump when GSA-Fuzz enables position learning. From the experiments, we can see that GSA-Fuzz brings benefits to fuzzing through learning mutation positions. Therefore, it is meaningful to give more fuzzing chances to the core parts of seeds.

5.11. Skip-Deterministic Experiments. In this subsection, we run skip-deterministic experiments to further verify the performance of GSA-Fuzz. In the experiments, AFL, MOPT-AFL, and GSA-Fuzz are all set to skip the deterministic stage. We use AFL (-d), MOPT (-d), and GSA-Fuzz (-d) to represent fuzzers that are set to skip the deterministic stage. The environment settings are the same as in Section 5.1.

The results are shown in Table 9, from which we can see that GSA-Fuzz's performance in finding paths is better than MOPT-AFL (-d) and AFL (-d). For instance, GSA-Fuzz (-d) finds averaged 21494 paths over five runs in sass, which are 11% more than MOPT-AFL (-d) and AFL (-d). In pdfimages, GSA-Fuzz finds 14005 paths while MOPT-AFL finds 12751 paths and AFL (-d) finds 12439 paths. Both two comparison experiments demonstrate that GSA-Fuzz (-d) can work more efficiently than AFL (-d) and MOPT-AFL (-d). In conclusion, the learning of operators and mutation positions can truly improve the performance of fuzzing.

Figure 9: Convergence study of bitflip1/1 in each case: (a) cxxfilt, (b) infotocap, (c) infotocap, (d) nm, (e) objdump, (f) openssl, (g) pdfimages, (h) pdftotext, (i) sass, and (j) tcpdump.

6. Related Work

6.1. Two Main Types of Fuzzing. In generating interesting seeds, there are mainly two types of fuzzing, which are generation-based fuzzing and mutation-based fuzzing.

Generation-based fuzzing. This type of fuzzing usually needs sufficient knowledge of the seed format [31–33]. It usually predefines a seed configuration file, and all newly generated seeds are based on the configuration file. Without a concise document, it would be a very difficult task for fuzzers to analyze the seed and generate the seed configuration file. For example, Skyfire [34] uses the knowledge in existing samples to generate seed inputs with good distribution. GRIMOIRE [35]

Table 7: The consumed CPU time (in seconds) of MOPT-AFL and GSA-Fuzz in optimization of probability distribution (4-hour runs).

Target	MOPT-AFL	GSA-Fuzz
cxxfilt	$1.08 * 10^{-4}$	$3.12 * 10^{-4}$
infotocap	$7.00 * 10^{-5}$	$1.35 * 10^{-4}$
libpng	$7.5 * 10^{-5}$	$1.72 * 10^{-4}$
openssl	$1.4 * 10^{-5}$	$4.7 * 10^{-5}$
nm	$5.3 * 10^{-5}$	$1.11 * 10^{-4}$
objdump	$3.4 * 10^{-5}$	$1.14 * 10^{-4}$
pdfimages	$1.1 * 10^{-5}$	$2.1 * 10^{-5}$
pdftotext	$6 * 10^{-6}$	$2.2 * 10^{-5}$
sass	$3.7 * 10^{-5}$	$1.35 * 10^{-4}$
tcpdump	$8.7 * 10^{-5}$	$2.19 * 10^{-4}$

Table 8: The example of 24-hour position learning in one of five runs.

Target	Fuzzer	Enable position learning	P (segment1)	P (segment2)	P (segment3)	P (segment4)	P (segment5)
sass	GSA-Fuzz	N	0.2	0.2	0.2	0.2	0.2
		Y	**0.06**	**0.11**	**0.15**	**0.24**	**0.44**
tcpdump	GSA-Fuzz	N	0.2	0.2	0.2	0.2	0.2
		Y	**0.38**	**0.30**	**0.13**	**0.12**	**0.07**

GSA-Fuzz learned the probability of each segment.

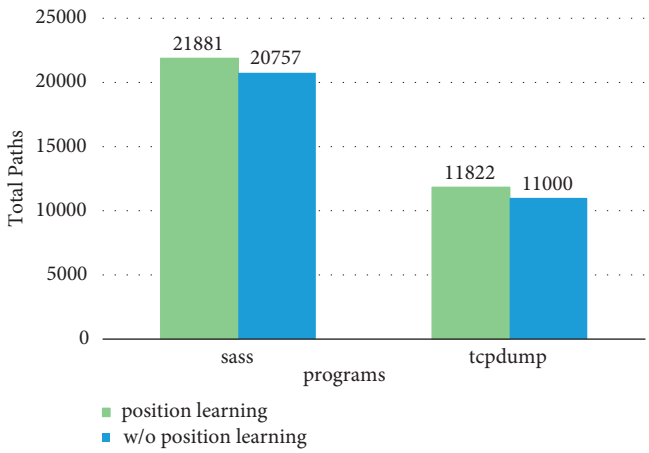

Figure 10: Averaged paths that GSA-Fuzz has found with enabling position learning and disenabling position learning over 5 runs in 24 hours.

Table 9: The performance of AFL, MOPT-AFL, and GSA-Fuzz in skip-deterministic experiments when fuzzing sass.

Target	Fuzzer	Total paths
sass	AFL (-d)	19105
	MOPT-AFL (-d)	19391
	GSA-Fuzz (-d)	21494
pdfimages	AFL (-d)	12439
	MOPT-AFL (-d)	12751
	GSA-Fuzz (-d)	14005

uses grammar inference to generate highly structured seeds.

Mutation-based fuzzing. Mutation-based fuzzing does not need to know the structure of the file in advance, and they are widely studied by researchers [19, 24, 25, 36, 37]. AFL-Fast [1] is a classic improvement fuzzer of AFL. It introduces the Markov Chain model to distribute higher energy to the low-frequency paths, which improves the performance of AFL. EcoFuzz [23] carries out deeper research on the energy distribution of seeds and proposes a variant of the Adversarial Multi-Armed Bandit model to make promotion. Aiming at improving code coverage, Zhu proposed CSI-Fuzz [38], which greatly reduces path collisions. There are also solutions combined with other technologies (e.g. symbolic execution [10, 39, 40], machine learning [22], reinforcement learning [41], taint tracking [42, 43], etc.) to improve the efficiency of fuzzing [11, 12, 35, 44–47].

6.2. Mutation Strategies in Fuzzing. We present GSA-Fuzz in this paper, which is to generate high-quality seeds by guiding the seed mutation with the GSA scheduling scheme. MOPT [20] is a novel fuzzer that focuses on improving seed mutation in scheduling the selection of mutation operators, and it applies the PSO algorithm to learn the operator probabilities for different programs. However, it has been proven that the PSO algorithm does not perform well in exploring the optimal solution in high-dimensional space [21], and this causes MOPT to spend much time learning the operator probabilities. Compared with MOPT, our GSA-Fuzz divides a seed into multiple segments and applies GSA to optimize operator probabilities in each segment. In learning operator probabilities, GSA-Fuzz can learn faster than MOPT-AFL.

FairFuzz [4] marks the core parts of the seed according to specific conditions and keeps these key parts from being mutated. This strategy can reduce the fuzzing of high-frequency paths. However, its efficiency highly depends on the specific conditions, and it does not consider the whole seed. Unlike FairFuzz, GSA-Fuzz lifts the restriction, and it regards a seed as five parts rather than marking some small parts of the seed. When GSA-Fuzz conducts a seed mutation, it will

mutate each part differently. Godefroid et al. [22] consider a seed input as continuous input sequences and use RNN to generate new seed inputs. But this scheme is just designed for seed input of PDF format while GSA-Fuzz is a general solution for the seed of all formats. Other fuzzers such as AFLSmart [48] require users to provide prior knowledge to keep the attribute of the seed format unchanged. In order to get prior knowledge of seed format, users may need to search for lots of initial seeds and analyze them, which is difficult to fuzz real-world programs. In contrast, GSA-Fuzz does not require any prior knowledge and can be used conveniently.

6.3. Seed Selection Strategies in Fuzzing. Our GSA scheme can also combine with seed selection strategies because it can produce many high-quality seeds for fuzzers. Therefore, we summarize some related seed selection strategies.

Generally speaking, the efficiency of fuzzing is related to the quality of the provided initial seeds. In order to improve the quality of initial seeds, some solutions include selecting the initial seeds from a large number of candidate seeds [8] or generating high-quality seeds by processing high-structured seeds [34]. Besides, some researchers also design their standards to select seeds for mutation. For example, AFLFast [1] and VUzzer [25] prefer to choose the seed with a low frequency for mutation. DeepFuzzer [7] has a statistical seed selection algorithm that is used to judge the quality of seeds, and CEREBRO [6] uses an online multi-objective-based algorithm to evaluate the quality of seeds.

7. Conclusion

Popular mutation-based fuzzers such as AFL follow a random seed mutation strategy, which is inefficient in handling mutation operators and mutation positions of a seed. Therefore, in this paper, we presented GSA-Fuzz to overcome the issue. GSA-Fuzz uses the Gravitational Search Algorithm (GSA) to learn the selection probability distributions of mutation operators and mutation positions. It can quickly find efficient operators and seed mutation positions and use them to guide the mutation in real time, which greatly improves the mutation efficiency of AFL. In addition, GSA-Fuzz manages AFL's two mutation stages more rationally through the provided *flip* mode. Our testing of GSA-Fuzz on 10 open-source programs showed that GSA-Fuzz found more paths and unique crashes, and had a higher line coverage in most cases, compared with three state-of-the-art fuzzers, AFL, MOPT-AFL, and EcoFuzz. We also conducted a systematic analysis of all experiments and proved the rationality and practicality of GSA-Fuzz. Overall, GSA-Fuzz can significantly improve the efficiency of seed mutation in AFL, and its GSA scheme may also give benefits to other mutation-based fuzzers.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Authors' Contributions

Mingmin Lin contributed to conceptualization, methodology, software, validation, data curation, formal analysis, and writing original draft of the manuscript. Yingpei Zeng contributed to methodology, resources, validation, investigation, project administration, and funding acquisition. TingWu contributed to resources and supervision. QiuhuaWang contributed to resources and supervision. Linan Fang contributed to supervision. Shanqing Guo contributed to resources and funding acquisition.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant no. 61902098; in part by the Zhejiang Provincial Natural Science Foundation of China under Grant no. LY22F020022; in part by the National Natural Science Foundation of China under Grant no. 91546203; in part by the Key Research Project of Zhejiang Province under Grant nos. 2020C01078, 2019C01 012, and 2017C01062; and in part by Major Scientific and Technological Innovation Projects of Shandong Province, China under Grants no. 2017CXGC0704, 2018CXGC0708, and 2019JZZY010132 and the Qilu Young Scholar Program of Shandong University.

References

[1] M. Bohme, V. T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[2] S. Gan, C. Zhang, X. Qin et al., "Collafl: path sensitive fuzzing," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P)*, pp. 679–696, IEEE, San Francisco, CA, USA, May 2018.

[3] Z. M. Jiang, J. J. Bai, K. Lu, and S. M. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, pp. 2595–2612, USA, May 2020.

[4] C. Lemieux and K. Sen, *Fairfuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage*, 2017, https://arxiv.org/abs/1709.07101.

[5] T. Ji, Z. Wang, Z. Tian et al., "Aflpro: direction sensitive fuzzing," *Journal of Information Security and Applications*, vol. 54, Article ID 102497, 2020.

[6] Y. Li, Y. Xue, H. Chen et al., "Cerebro: context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*pp. 533–544, 2019.

[7] J. Liang, Y. Jiang, M. Wang et al., "Deepfuzzer: accelerated deep greybox fuzzing," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[8] A. Rebert, S. K. Cha, T. Avgerinos et al., "Optimizing seed selection for fuzzing," in *USENIX Security Symposium (USENIX Security 14)*, pp. 861–875, 2014.

[9] Y. Chen, P. Li, J. Xu et al., "Savior: towards bug-driven hybrid testing," in *2020 IEEE Symposium on Security and Privacy (S&P)*, pp. 1580–1596, IEEE, 2020.

[10] M. Cho, S. Kim, and T. Kwon, "Intriguer: field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM*

SIGSAC Conference on Computer and Communications Security, pp. 515–530, 2019.

[11] N. Coppik, O. Schwahn, and N. Suri, "Memfuzz: using memory accesses to guide fuzzing," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST),* pp. 48–58, IEEE, 2019.

[12] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (S&P),* pp. 1497–1511, IEEE, 2020.

[13] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pp. 39–43, Ieee, 1995.

[14] H. Rezaei, "Grey wolf optimization (gwo) algorithm," in *Advanced Optimization by Nature-Inspired Algorithms*, pp. 81–92, 2018.

[15] F. Al Thobiani, S. Khatir, B. Benaissa, E. Ghandourah, S. Mirjalili, and M. Abdel Wahab, "A hybrid pso and grey wolf optimization algorithm for static and dynamic crack identification," *Theoretical and Applied Fracture Mechanics*, vol. 118, Article ID 103213, 2022.

[16] Q. Chen, R. Jia, and S. Pang, "Deep long short-term memory neural network for accelerated elastoplastic analysis of heterogeneous materials: an integrated data-driven surrogate approach," *Composite Structures*, vol. 264, Article ID 113688, 2021.

[17] A. Ouladbrahim, I. Belaidi, S. Khatir, E. Magagnini, R. M. Capozucca, and M. Abdel Wahab, "Experimental crack identification of api x70 steel pipeline using improved artificial neural networks based on whale optimization algorithm," *Mechanics of Materials*, vol. 166, Article ID 104200, 2022.

[18] H. Tran-Ngoc, S. Khatir, H. Ho-Khac, G. De Roeck, and T. M. Bui-Tien, "Efficient artificial neural networks based on a hybrid metaheuristic optimization algorithm for damage detection in laminated composite structures," in *Composite Structures*, 2021.

[19] M. Zalewski, *American Fuzzy Lop*, 2014.

[20] C. Lyu, S. Ji, C. Zhang et al., "Mopt: optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1949–1966, 2019.

[21] E. Rashedi, H. Nezamabadi-pour, and S. Saryazdi, "Gsa: a gravitational search algorithm," *Information Sciences*, vol. 179, no. 13, pp. 2232–2248, 2009.

[22] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE),* pp. 50–59, IEEE, 2017.

[23] T. Yue, P. Wang, Y. Tang et al., "Ecofuzz: adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2307–2324, 2020.

[24] K. Serebryany, *Continuous Fuzzing with Libfuzzer and Addresssanitizer*157 pages, IEEE Cybersecurity Development (SecDev), IEEE, 2016.

[25] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, *Vuzzer: Application-Aware Evolutionary Fuzzing*, pp. 1–14, NDSS, 2017.

[26] A. D. Householder and J. M. Foote, *Probability-based Parameter Selection for Black-Box Fuzz Testing*, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Technical Report, 2012.

[27] H. Tran-Ngoc, S. Khatir, T. Le-Xuan, G. De Roeck, T. M. Bui-Tien, and M. Abdel Wahab, "A novel machine-learning based on the global search techniques using vectorized data for damage detection in structures," *International Journal of Engineering Science*, vol. 157, 2020.

[28] M. Rash, "afl-cov," 2014.

[29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: a fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pp. 309–318, 2012.

[30] Y. Li, S. Ji, Y. Chen et al. "Unifuzz: a holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *30th USENIX Security Symposium*vol. 21, 2021.

[31] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*vol. 43, no. 6, pp. 206–215, 2008.

[32] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium (USENIX Security 12)*, pp. 445–458, 2012.

[33] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "Slf: fuzzing without valid seed inputs," in *IEEE/ACM 41st International Conference on Software Engineering (ICSE)* pp. 712–723, IEEE, 2019.

[34] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: data-driven seed generation for fuzzing," in *IEEE Symposium on Security and Privacy (S&P),* pp. 579–594, IEEE, 2017.

[35] T. Blazytko, M. Bishop, C. Aschermann et al. "Grimoire: synthesizing structure while fuzzing," in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1985–2002, 2019.

[36] M. Böhme, V. T. Pham, M. D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2329–2344, 2017a.

[37] P. Chen and H. Chen, "Angora: efficient fuzzing by principled search," in *IEEE Symposium on Security and Privacy (S&P)* pp. 711–725, IEEE, 2018.

[38] X. Zhu, X. Feng, X. Meng et al., "Csi-fuzz: full-speed edge tracing using coverage sensitive instrumentation," *IEEE Transactions on Dependable and Secure Computing*, p. 1, 2020.

[39] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: incremental hybrid fuzzing with polyhedral path abstraction," in *2020 IEEE Symposium on Security and Privacy (S&P)* pp. 1613–1627, IEEE, 2020.

[40] Y. Yao, W. Zhou, Y. Jia, L. Zhu, P. Liu, and Y. Zhang, "Identifying privilege separation vulnerabilities in iot firmware with symbolic execution," in *European Symposium on Research in Computer Security*, pp. 638–657, Springer, 2019.

[41] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE),* pp. 1410–1421, IEEE, 2020.

[42] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 499–513, 2019.

[43] S. Gan, C. Zhang, P. Chen et al., "Greyone: data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2577–2594, 2020.

[44] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (S&P),* pp. 1597–1612, IEEE, 2020.

[45] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *2018 IEEE Security and Privacy Workshops (SPW),* pp. 116–122, IEEE, 2018.

[46] W. Drozd and M. D. Wagner, *Fuzzergym: A Competitive Framework for Fuzzing and Learning*, arXiv preprint arXiv: 1807.07490, 2018.

[47] Y. Li, S. Ji, C. Lyu et al., "V-fuzz: vulnerability prediction-assisted evolutionary fuzzing for binary programs," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3745–3756, 2022.

[48] V. T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, *Smart Greybox Fuzzing*, IEEE Transactions on Software Engineering, 2019.