applied sciences

MDPI

Article

PSOFuzzer: A Target-Oriented Software Vulnerability Detection Technology Based on Particle Swarm Optimization

Chen Chen [1,2,*], Han Xu [1] and Baojiang Cui [1]

[1] School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China; xuhan123@bupt.edu.cn (H.X.); cuibj@bupt.edu.cn (B.C.)
[2] School of Information and Navigation, Air Force Engineering University, Xi'an 710077, China
* Correspondence: 00152tenten@bupt.edu.cn

Abstract: Coverage-oriented and target-oriented fuzzing are widely used in vulnerability detection. Compared with coverage-oriented fuzzing, target-oriented fuzzing concentrates more computing resources on suspected vulnerable points to improve the testing efficiency. However, the sample generation algorithm used in target-oriented vulnerability detection technology has some problems, such as weak guidance, weak sample penetration, and difficult sample generation. This paper proposes a new target-oriented fuzzer, PSOFuzzer, that uses particle swarm optimization to generate samples. PSOFuzzer can quickly learn high-quality features in historical samples and implant them into new samples that can be led to execute the suspected vulnerable point. The experimental results show that PSOFuzzer can generate more samples in the test process to reach the target point and can trigger vulnerabilities with 79% and 423% higher probability than AFLGo and Sidewinder, respectively, on tested software programs.

Keywords: fuzzing; model-based fuzzing; vulnerability detection; code coverage; open-source program; directed fuzzing; static instrumentation; source code instrumentation

Citation: Chen, C.; Han, X.; Cui, B. PSOFuzzer: A Target-Oriented Software Vulnerability Detection Technology Based on Particle Swarm Optimization. *Appl. Sci.* **2021**, *11*, 1095. https://doi.org/10.3390/app11031095

Received: 11 November 2020
Accepted: 18 January 2021
Published: 25 January 2021

1. Introduction

The appearance of an increasing number of software vulnerabilities has made automatic vulnerability detection technology a widespread concern in industry and academia. The main technology used in automatic vulnerability detection is fuzzing, the principle of which is shown in Figure 1. The core component of fuzzing is the sample generator. It generates many mutated samples, and then sends them to the target program for execution. These mutated samples will greatly expand their execution behavior in the program, which may meet the constraint requirements of the sample behavior that the vulnerability be triggered. One type of the most serious vulnerabilities, memory corruption vulnerability, will be exposed in the form of program crashes in this process.

Early fuzzing technology [1,2] implemented non-oriented fuzzing, which randomly mutates samples, including randomly adding, deleting and flipping bytes in the sample. This process does not adjust the mutation strategy of subsequent samples by returning the runtime information of previous tests, and the deep logic of the program cannot be executed because most of the generated samples are invalid due to the destroyed format of the samples by this kind of random mutation. Technology to restrict the sample format has been applied to fuzzing [3–8] to greatly increase the number of valid samples, but the efficiency of fuzzing technology is low because the runtime state of the program cannot be perceived to adjust the fuzzing strategy.

In contrast to non-oriented fuzzing, coverage-oriented fuzzing [9–13] can use lightweight instrumentation to obtain the runtime information of the program to effectively guide the fuzzing process to pursue high coverage in control flow graph (CFG). For example, by selecting the sample with a new edge as the seed sample for the next round of mutations, the execution of the sample will have a greater possibility of obtaining high

coverage [9–11]; coverage-oriented fuzzing can also use heavyweight instrumentation to perform dynamic symbol execution [14–21]. This approach transforms external data into symbols to participate in the execution, collects the symbol expressions at each program node, and generates samples that try to execute the full path by solving the expressions. Dynamic symbol execution can fully perceive the state of the program, but it is difficult to solve the expressions when they are overly complex. In addition, the problem of path explosion [22] is difficult to solve.

Coverage-oriented fuzzing [9–21,23–28] increases the probability of triggering vulnerabilities by executing code blocks or paths as much as possible. However, it causes a large quantity of computational resources to be allocated to code that does not have vulnerabilities. Target-oriented fuzzing [29–32] leads the test path to suspected vulnerable points by taking the samples close to the vulnerable point as the seed samples of the next mutation or solving the symbolic expression of the path to the vulnerable point. Although target-oriented fuzzing provides a clearer goal, the generated test samples do not effectively approach the goal because the guidance algorithm of the fuzzing and the sample mutation strategy are independent of each other, which results in a poor guidance effect. Moreover, the blindness of the random mutations results in a weak penetration ability of the samples to deeper program logic, and it is difficult to generate samples using dynamic symbol execution technology when the program is complex.

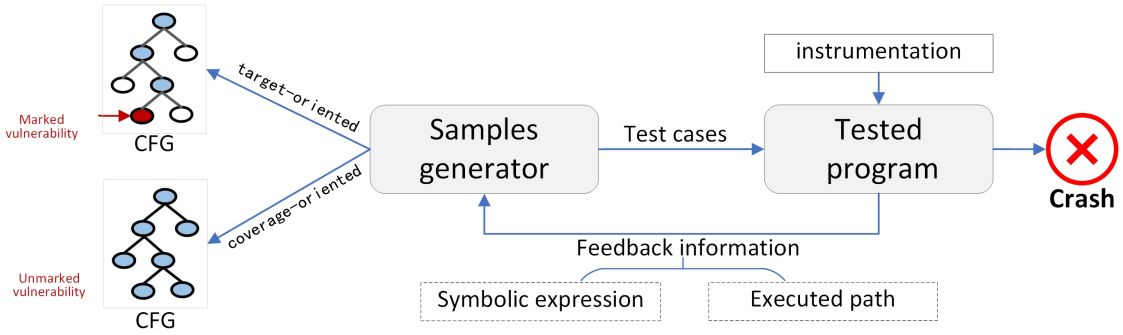

Figure 1. The principle of fuzzing.

In this paper, we transform fuzzing into a mathematical optimization problem by leveraging particle swarm optimization (PSO) [33], which learns the high-quality characteristics of all the samples in the whole time and generates many samples that can be executed to the target point. Additionally, we use format constraint technology to enhance the penetration ability of the samples in the program and improve the probability of trigger vulnerabilities.

The main contributions of this paper are as follows:

(1) We theoretically analyze the feasibility of introducing the PSO algorithm to generate samples in the fuzzing process.
(2) We use format constraint technology to build a fuzzing system based on the PSO algorithm that can be used for target-oriented vulnerability detection.
(3) We experimentally verify the effectiveness of the PSO algorithm in fuzzing.

The remainder of this paper is structured as follows. Section 2 discusses the related work. Section 3 analyzes the relationship between the PSO algorithm and fuzzing. Section 4 elaborates the design of PSOFuzzer. Section 5 describes the experiment with the PSOFuzzer and analyzes the results. Section 6 analyzes the challenges and limitations of PSOFuzzer. Section 7 points out the development direction of PSOFuzzer. Finally, Section 8 presents the conclusions.

2. Related Work

We refer to the query methods mentioned in the literature [34,35], query the papers that use the keywords 'fuzzing' and 'vulnerability detection', and collect and analyze these papers in the last 15 years. The main strategy to improve the efficiency of fuzzing in these papers is to guide the fuzzing process. From the perspective of guidance, fuzzing includes three types: coverage-oriented, target-oriented and non-oriented fuzzing.

2.1. Coverage-Oriented Fuzzing

Coverage-oriented fuzzing has become a predominant technology because it can automatically detect vulnerability in software without human participation. The use of runtime information in different ways has encouraged considerable new development. On the basis of the degree of acquisition and use of state information in the process of program testing, coverage-oriented fuzzing can be divided into three categories: light use, heavy use and medium use.

AFL [9] is a typical representative of light use of state information that leverages instrumentation technology to obtain path coverage information. This information is used to evaluate samples and guide tests to execute paths not executed. Some technologies have improved AFL, such as AFLFast [10] and EcoFuzz [25], which leverage the Markov model and a variant of the adversarial multi-armed band model to allocate excessive energy to seeds that separately exercise the low-frequency paths. EcoFuzz [25] achieves a better performance. VUzzer [23] presented an application-aware evolutionary fuzzing strategy to maximize coverage and explore deeper paths based on static and dynamic analysis. Its error-handling basicblock detection technique is similar to the technique to identify low-frequency based on the Markov model in AFLFast [10], albeit much lightweight. CollAFL [11] achieves greater path coverage by reducing path conflicts in AFL. MOPT [13] optimizes the selection of mutation operations to improve the path coverage using the PSO algorithm. Unlike MOPT, PSOFuzzer leverages the PSO algorithm to generate samples, while MOPT leverages the PSO algorithm to assist AFL in selecting the mutation algorithm. In addition, PSOFuzzer is a target-oriented fuzzer, while MOPT is a coverage-oriented fuzzer. AFL++ [28] integrates several fuzzing studies and provides a benchmark to assess other fuzzing technologies. Because AFL++ is built on coverage-oriented fuzzing systems, it is not suitable for evaluating PSOFuzzer. In addition, some fuzzing techniques have been applied to specific types of software vulnerability detection. HFL [21] applies hybrid fuzzing to kernel testing by converting indirect control transfers, inferring system call sequences and identifying nested argument types of system calls, and shows a higher coverage than Moonshine [36] and Syzkaller [37]. FuzzGen [24] achieved better code coverage in testing the library by whole system analysis to infer the library's interface. MUZZ [26] hunts more bugs than AFL under a multithreading context by three novel thread-aware instrumentations to engender runtime feedback to accentuate execution states caused by thread interleavings. Ijon [27] introduced human analysis to guide the fuzzer by an annotation mechanism, which systematically explores the program's behavior by the internal state of the program. it obtains a higher coverage and more crash than AFL.

Fuzzing via heavy use of state information is based mainly on symbol execution. For example, SAGE [17] achieves greater code coverage by solving symbolic expressions on different program execution paths. S2E [15] and MAYHEM [16] are similar to SAGE [17]. S2E [15] uses path analyzers to drive the target system down all the execution paths of interest while checking the security properties of each path. MAYHEM [16] can not only automatically detect vulnerabilities by dynamic symbol execution, but also generate hijack exploits automatically. PANGOLIN [20] leverages polyhedral path abstraction to preserve the exploration state in the concolic execution stage to implement incremental fuzzing. QSYM [19] achieves a better performance than VUzzer [23] and Driller [18] by losing the strict soundness requirements of conventional concolic executors. Symbol execution relies on many computing resources and faces the problem of path

explosion [22]. Therefore, it is not more practical than coverage-oriented fuzzing based on sample mutation [9–13].

The development of fuzzing also brings new ideas by the medium use of state information. NEUZZ [12] leverages surrogate natural network models to incrementally learn smooth approximations of a program's branching behavior and then uses gradient-guided optimization to generate inputs that can trigger different bugs. Driller [18] combines the advantages of AFL and symbol execution to overcome the difficulty of passing some program nodes that AFL cannot pass using symbol execution technology to calculate the constraints of the node and generates samples that can meet the conditions, thereby improving the depth and breadth of the path coverage.

2.2. Target-Oriented Fuzzing

Target-oriented fuzzing can be divided into gray-box testing and white-box testing. Representative gray-box testing methods include AFLGo [31] and Sidewinder [32]. AFLGo leverages a simulated annealing-based power schedule that gradually assigns more energy to seeds that are close to critical system calls or dangerous locations. AFLGo optimizes the allocation of computing resources by evaluating samples, while PSOFuzzer uses the PSO algorithm to combine the guidance process and a sample mutation algorithm to provide better mutation efficiency for samples. Sidewinder [32] uses the inheritance ability of genetic algorithms for the high-quality characteristics of samples, and it guides the execution path of test samples to the suspected vulnerability location in the control flow chart. However, this method is not suitable for programs with complex sample types. PSOFuzzer provides a constraint mechanism in the process of mutation via sample normalization to restrict mutations that do not cause significant damage to the format of the sample to ensure that the generated sample has sufficient penetration ability for programs with a complex sample format.

The main representatives of white-box fuzzing are BORG [30] and Dowser [29]. Both guide the execution to expected vulnerability locations by means of symbolic execution. The difference between these methods is that BORG targets overread vulnerabilities, while Dowser targets buffer overflow vulnerabilities. Fuzzing based on symbol execution technology transforms the problem of sample generation into an optimization problem. The disadvantage is that when there is a complex symbol expression to solve, it cannot generate samples that meet the expression. PSOFuzzer transforms the problem of sample generation into a problem of mathematical optimization, which is equivalent to decomposing the difficulty of the problem at the time level to generate samples that meet the conditions after many iterations.

2.3. Non-Oriented Fuzzing

In non-oriented fuzzing, the test process is not influenced by any control, usually due to the lack of runtime information. The test mode is mainly represented by black-box fuzzing. The most typical representative is the first fuzzing system $fuzz$ [1], which mutates samples randomly, sends them to a Linux program to be tested and monitors whether there is an exception. This kind of random variation will cause the sample format to be seriously damaged, which will cause the program to exit in the format verification stage because the sample cannot meet the format requirements when it is executed.

An effective fuzzing technology enhanced by a sample format constraint is used to solve this problem. The representative technologies include SPIKE [4], Peach [3] and PROTOS [5]. SPIKE [4] is a block-based fuzzing approach that satisfies the relations between different fields of input by nested data blocks. Peach [3] restrains the input data by constructing Peach Pits, which describe the types of data chunks and fields and the relationships between them in XML (eXtensible Markup Language) format files. PROTOS [5] can infer a model of a packet structure from given samples, and it can leverage the model to automatically generate samples for fuzzing. The three fuzzing systems use different techniques to achieve the same goal, i.e., to increase the number of valid samples

after mutation, but they cannot adjust the test fuzzing strategy according to the runtime state, which results in low test efficiency. PSOFuzzer introduces this sample format constraint technology to perform the normalization of samples to improve the quality of the samples in the process of mutation.

3. Particle Swarm Optimization and Fuzzing

Fuzzing is a process of continuously generating and executing samples to trigger a crash to identify vulnerabilities. PSO is an optimization algorithm, which constantly updates the particle value to gradually seek the optimal solution. We observe the relationship between PSO and fuzzing, so we can apply the PSO algorithm to fuzzing to optimize the sample generation.

3.1. Principle of Particle Swarm Optimization

The PSO algorithm was proposed by Eberhart and Kennedy in 1995 [33]. Its basic core is to make use of the information sharing of individuals in a group in such a way that the movement of the whole group can produce an evolutionary process from disorder to order in a problem-solving space to obtain the optimal solution of a problem. PSO originates from research on the predatory behavior of birds. Imagine a scene where a group of birds is foraging, and there is a corn field in the distance. The birds do not know where the corn field is, but they know how far away they are from the corn field. Thus, the best and simplest strategy to find the corn field is to search the area around the nearest birds.

In PSO, the solution of every optimization problem is the position of a bird in the search space, which is called a "particle", and the optimal solution of the problem corresponds to the position of the corn field. All the particles have a position vector (the position of the particle in the solution space) and a velocity vector (determining the direction and speed of the next flight), and the fitness value of the current position can be calculated according to the objective function, which can be understood as the distance from the corn field. In each iteration, the examples learn not only from their own experience (historical position) but also from the experience of the optimal particles in the population to determine how to adjust and change the flight direction and speed in the next iteration. In this way, the iteration will gradually lead to the optimal solution for the whole population.

3.2. The Relationship between Fuzzing and Particle Swarm Optimization

Fuzzing is widely used due to its high automatization and efficiency. The number of generated high-quality samples is the key to the fuzzing efficiency. High-quality samples in different types of fuzzing are different. In coverage-oriented fuzzing, high-quality samples refer to those samples that can execute to new edges. In target-oriented fuzzing, high-quality samples refer to those samples that can be executed to suspected vulnerable points. Different fuzzing techniques use different guidance methods for the test process to increase the number of high-quality samples. We discuss the relationship between PSO and fuzzing using different guidance methods.

Non-oriented fuzzing focuses only on whether the program crashes in the test process; it does not interfere with the mutation according to the real-time internal state of the program. Therefore, the generation of test samples is conducted in a blind state that does not account for relevant information. In contrast, the PSO algorithm accounts for the other relevant information of the sample; as a result, the PSO algorithm cannot be used in this way.

The goal of coverage-oriented fuzzing is to improve the coverage of the sample execution. The coverage is the attribute of many samples, not the attribute of a single sample. However, the evaluation of particles in the PSO algorithm is based on a single sample. That the coverage of a single sample is high does not mean that the sample can improve the coverage rate, and whether a single sample can improve the coverage rate in different test processes is inconsistent. Therefore, the indicators to improve the coverage

rate here cannot be used as indicators to measure a single sample, and the PSO algorithm is not suitable for generating samples directly in coverage-oriented fuzzing.

The goal of target-oriented fuzzing is to make the sample execute the designated target point; thus, a sample that can execute the target point is a high-quality sample. The evaluation of the current sample can be measured as the sample distance from the target: the closer the sample is to the target, the higher the evaluation value. The sample distance can be specified as the shortest distance of sample execution blocks from the target code block. The sample distance is an attribute of a single sample, which is consistent with the evaluation of a single particle in the particle swarm. Therefore, we attempt to introduce the PSO algorithm into target-oriented fuzzing, and we use PSO's powerful ability to quickly approach the optimal solution to generate many test samples that can reach the target point to improve the probability of triggering a vulnerability.

4. Fuzzing System Design

PSOFuzzer is a fuzzing system based on the particle swarm optimization algorithm that implements target-oriented sample generation technology. PSOFuzzer includes a static analyzer, distance calculator, instrumentator and sample generator. The overall framework of the technology is shown in Figure 2.

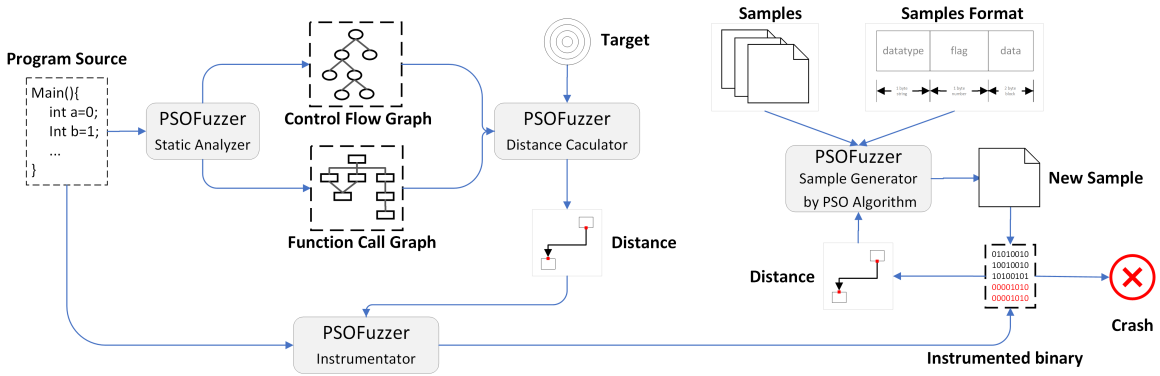

Figure 2. PSOFuzzer Technology Framework.

PSOFuzzer extracts the function call graph and control flow graph by static analysis, which are applied to calculate the static distances between each basic code block and the target location, where the suspected vulnerability is located. PSOFuzzer employs code instrumentation technology to insert the code to calculate the actual distances between the execution path of the sample and the target position using the calculated static distances when the program is running dynamically. In the stage of fuzzing, PSOFuzzer uses the PSO algorithm and sample format constraint technology to generate samples. Sample format constraint technology can effectively improve the penetration ability of the mutated samples. The PSO algorithm can realize continuous learning and iterations among multiple samples, including the current samples and historical optimal samples. Using the distance between the sample path and the vulnerable point, the algorithm guides additional samples to be executed to the target point.

4.1. A Measurement of the Distance between a Seed Input and Target Point

PSOFuzzer is a target-oriented fuzzing technology that aims to let more samples pass through the target point. The closer the execution path of a sample is to the target point, the more beneficial it is for testing. Therefore, the index for the testing guidance process is the distance between the current sample and the target point. To analyze the distance between the current sample and the locations of vulnerable points, we consider two types of granularity: basic block level and function level.

The basic block refers to the sequence of statements that are executed in sequence. There is only one entry and one exit: entry is the first statement, and exit is the last

statement. A basic block enters only from its entry and exits from its exit. The execution path of sample s_{ij} is the execution flow from basic block b_i in the program to basic block b_j, which is a sequence composed of several basic blocks in a certain order $(b_i, b_{i+1}, b_{i+2}, \ldots b_j)$. A distance calculator is used to calculate the shortest distance between all code blocks in path s_{ij} to a specified target t_m as the distance between the current sample and the locations of the vulnerable points.

Functions are larger in granularity than basic blocks. To increase the weight of a function call when calculating the distance, we calculate the distance between the sample and the target point using a combination of the distance between two function calls and the distance between two code blocks in the function. The formula is shown in Formula (1):

$$d = df * \alpha + db \tag{1}$$

We guide the execution to the function containing the target point first, and then guide to the position of the target point in the function. In Formula (1), df is the distance on the function layer, i.e., the distance between the function and the target function, which includes the suspected vulnerable point. db is the distance on the basic block layer, i.e., the distance between the basic block and suspected vulnerable point. We use the Dijkstra algorithm to calculate these two types of distances. Additionally, to avoid db being too large to effectively guide the sample to execute the target function, we set the weight α for df, and the value of α is the farthest distance to the vulnerable point in the target function.

The distance between a sample and the target is calculated in three stages: static distance analysis stage, instrumentation stage and running stage. In the static distance analysis stage, the distance between each basic block and the target point, and the distance between each function and the function of the target point are calculated. In the instrumentation stage, the instrumentator inserts the distance calculation logic into the source code according to the distance between each function and the basic block from the target point. The logic is to compare the distance value of the current basic block or function with the executed values. If the former is less than the latter, the distance value is updated to the current value. In the running stage, the distance value is calculated by Formula (1) and compared with the historical minimum value according to the calculation logic of the instrumented code.

4.2. Mapping Fuzzing to Particle Swarm

To perform sample generation using the PSO algorithm, each element in the sample generation must be mapped with each element in the PSO algorithm. The PSO algorithm extracts the information of particles participating in the movement of a group, learns the information of the optimal sample, and updates the information for the other particles, which makes an evolutionary process from disorder to order in the problem-solving space to obtain the optimal solution. Assume that the number of particles in the swarm is n; at iteration time t, the coordinate position of each particle in d-dimensional space can be expressed as

$$\bar{x}_i(t) = (x_i^1, x_i^2, \ldots, x_i^j, \ldots, x_i^d) \tag{2}$$

In fuzzing, $\bar{x}_i(t)$ represents a specific sample generated at iteration t, and x_i^j represents a byte in the sample. The speed of the particle is expressed as

$$\bar{v}_i(t) = (v_i^1, v_i^2, \ldots, v_i^j, \ldots, v_i^d) \tag{3}$$

In fuzzing, $\bar{v}_i(t)$ represents the direction of the sample change. The coordinate position $\bar{x}_i(t)$ and speed $\bar{v}_i(t)$ of a particle at time $t+1$ are adjusted as follows:

$$\bar{v}_i(t+1) = \omega \bar{v}_i(t) + c_1 r_1 (\bar{p}_i(t) - \bar{x}_i(t)) + c_2 r_2 (\bar{p}_g(t) - \bar{x}_i(t)) \tag{4}$$

$$\bar{x}_i(t+1) = \bar{x}_i(t) + \bar{v}_i(t) \tag{5}$$

In Formulas (2) and (5), $\bar{x}_i(t)$ and $\bar{x}_i(t+1)$ represent the previous and current particles, respectively. In fuzzing, these values represent a sample generated at the previous and current iterations. In Formulas (3) and (4), $\bar{v}_i(t)$ and $\bar{v}_i(t+1)$ represent the previous and current speeds of the particle, respectively. In fuzzing, they represent the direction of the sample change in the previous iteration and current iteration.

Here, ω represents the weight of the particle inertia, which encourages expansion of the search space and exploration of new search areas, but it might restrict the local fine search in the later iterations of the algorithm. In fuzzing, this value is set to a random number distributed in [0,1] to adjust the intensity of the current sample change.

$\bar{p}_i(t)$ represents the optimal position experienced by the particle, which is called the "self-knowledge part" of the particle and refers to the ability of the particle to learn from itself. In fuzzing, this value represents the sample closest to the target point in the mutation process of the ith initial sample.

$\bar{p}_g(t)$ represents the best particle position in the swarm, which indicates the ability of particles to learn from the whole swarm. In fuzzing, this value represents the sample closest to the target point from all the initial samples and generated samples.

The parameters c_1 and c_2 represent the acceleration constants of the particles, which are usually taken to be [0,2]. Here, r_1 and r_2 are two random numbers that are uniformly distributed in [0,1].

Through the analysis, we map all types of elements in fuzzing to each element in PSO to provide a theoretical basis for the use of PSO to generate samples that continuously approach the preset target points in the program via continuous iteration.

4.3. Sample Normalization

When using PSO to generate samples, we have a problem to address. Every element of a vector in the PSO algorithm corresponds to a byte in the actual sample. The dimension of the vector involved in the calculation is determined, but the numbers of bytes in the actual samples are different. To ensure consistency between the two, a simple processing method is to fill with zeros, but this method leads to some problems, such as the example shown in Figure 3.

Figure 3. HTTP request format.

Figure 3 shows the format of the HTTP (HyperText Transfer Protocol) request. The gray block in the figure represents data with constant length, and the white block represents data with variable length. When the PSO algorithm is adopted, zeros are added to any sample with a short sample length to ensure that the calculation can be performed effectively. However, this method of zero filling causes data segments with different formats to be combined for the calculation; for example, the header name and uniform resource locator (URL) might be concurrently calculated, which would cause in many samples that do not conform to the necessary format to be discarded during program processing and result in low test efficiency. In addition, the number of vectors involved in the calculation is limited by the maximum sample size, which can cause difficulty in triggering overflow vulnerability due to having an insufficient sample length. To solve these problems, this paper uses the sample format template to match the sample bytes and the elements in the vector on the same type of fields. The principle is shown in Figure 4.

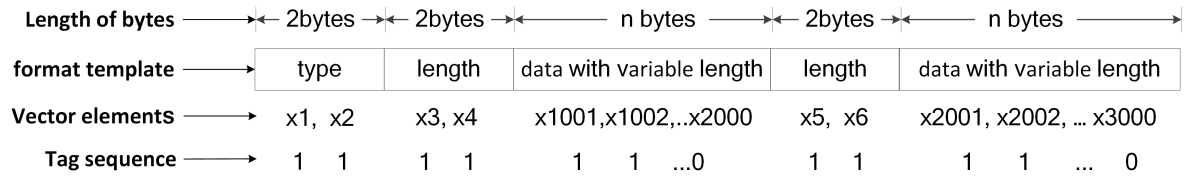

Figure 4. Principle of sample normalization.

We build a sample format template. The fields in the template can be divided into two different data types: variable length and fixed length. First, we allocate the corresponding number of vector elements for each fixed-length field from left to right according to the number of bytes; then, we allocate the variable-length field according to the maximum number of elements allocated in advance (the size can be set by the tester). Next, we map the sample data to the template uniformly, and we provide each sample with a tag sequence. This tag sequence is composed of 0 s and 1 s. If a field has no corresponding data, it is marked as 0; if there are data, it is marked as 1. This tag sequence realizes the association between the vector element and the sample data. The sample generator removes the vector element with the corresponding byte of 0 after generating a new vector; then, the other vectors are combined into a new sample for subsequent testing.

4.4. Fuzzing Based on the PSO Algorithm

The key to automatic vulnerability detection is to generate high-quality samples. In target-oriented fuzzing, high-quality samples are those that can be executed to the target point. We use the PSO algorithm to generate many high-quality samples through continuous iterative learning to reduce access to the invalid path space and improve the efficiency of vulnerability detection. The ability to achieve efficient learning among multiple samples by means of the PSO algorithm can improve the convergence speed of the target orientation to focus limited computing resources on the path space that is considered to be important.

Sample generation in fuzzing is performed continuously, which is consistent with the process of continuous iteration of the PSO algorithm. The core elements of the PSO algorithm include the particles, the positions of the particles and the values of the particles. Other elements, such as the particle speed, global optimal position and local optimal position, are calculated based on the three core elements. The samples in fuzzing are mapped to the particles in the PSO algorithm. The distance between the sample and the target point is mapped to the distance between the swarm and the target point. The degree of approximation to the vulnerable point is mapped to the weight of the particles. By the mapping operation, we design a sample generation mechanism, which is described in Algorithm 1:

The PSO algorithm describes an iterative process that includes three steps in each iteration.

- Update the local optimal particle: compare the fitness value of the current sample $S[i]$ with the optimal fitness value of this sample in history $L[i]$. If $f(S[i]) < f(L[i])$, then update $L[i]$ to $S[i]$. The fitness function f is used to obtain the distance between the current sample and the target point.
- Update the global optimal particle: compare each evaluation value of the local optimal samples $L[i]$ with the evaluation value of the global optimal sample g. If $f(L[i]) < f(g)$, update g with $L[i]$.
- Generate samples: map the current sample into the template, establish the relationship between sample and vector, and then generate new sample based on Formulas (4) and (5). At this time, the sample has good guidance but lacks enough destructiveness to trigger vulnerabilities. PSOFuzzer uses mutation algorithms applied to different types of data fields under a certain probability to improve the effect of mutation. The mutation algorithms the PSOFuzzer used are shown in Table 1.

Table 1. Mutation algorithms applied to different types of data fields.

Data Type	Mutation Algorithms
Blob	Flips part of total bits in a blob Slides a Double Word (DWORD) through the blob Grows the blob Shrinks the blob
String	Changes the case of a string Generates a string using the string in the dictionary Generates bad UTF-8 strings Injects Byte-order mark (BOM) markers into longer strings Generates bad long UTF-8 three-byte strings
Array	Changes the length of arrays to numerical edge cases Randomizes the order of the array Reverses the order of the array Changes the length of arrays from count −N to count +N
Number	Produces random numbers for each element Produces values that are unrelated to the default Value Produces defaultValue −N to defaultValue +N Changes the length of sized data to numerical edge cases Changes the length of sizes to numerical edge cases

Algorithm 1 PSO Algorithm.

while $stopCondition\ is\ false$ **do**
  for $i = 1$ *to* n **do**
    if $f(S[i]) < f(L[i])$ **then**
      $L[i] = S[i]$
    end if
  end for
  for $i = 1$ *to* n **do**
    if $f(L[i]) < f(g)$ **then**
      $g = L[i]$
    end if
  end for
  for $i = 1$ *to* n **do**
    $mapping(S[i], M)$
    $S_i = update(S[i], L[i], g)$
    if $random() > \lambda$ **then**
      $chooseFieldRandomly()$
      $mutateField()$
    end if
  end for
end while

S: current samples, n: number of samples, L: local optimal samples, g: global optimal sample, M: Data Model, f: function of calculating the fitness value of the sample.

5. Experiment

To verify the test effectiveness of PSOFuzzer, we conduct experiments on the generation efficiency of high-quality samples and the efficiency of verifying vulnerabilities. The former is used to verify that PSOFuzzer can efficiently generate samples that can execute to the target point, and the latter is used to verify that PSOFuzzer can effectively trigger known vulnerabilities. The trigger of these vulnerabilities will cause the software to crash or even remote code execute (RCE).

5.1. Experimental Preparation

Tools for comparison: In the experiments, we compare PSOFuzzer with AFLGo [31], a well-known target-oriented fuzzing tool, and Sidewinder [32], the tool that we reproduced using a genetic algorithm.

Experimental Environment: CPU: dual-core, 2.20 GHz 2.19 GHz, RAM: 4.00 GB, OS: ubuntu 16.04, kernel 4.15.0-123-generic, Compiler: gcc 5.4.0 is used.

Tested software: Some fuzzing techniques only obtain good results on specific types of software. To verify the extensive effectiveness of PSOFuzzer, we choose different types of software for testing. The tested objects are 12 software programs with vulnerabilities: gzip [38], tar [39], pdfresurrect [40], miniftp [41], binutils [42], libming [43], libjpeg [44], pngquant [45], ffjpeg [46], zziplib [47], exiv2 [48], and ngiflib [49]. Detailed information about the tested software is shown in Table 2.

Table 2. Software information.

Software	Type	Version Number	CVE ID	Vulnerability Position
gzip	compressor	1.2.4	2009–2624	inflate.c line 763
tar	compressor	1.14	2007–4476	safer_name_suffix.c line 1046
pdfresurrect	pdf analyzer	0.15	2019–14,267	pdf.c line 237
miniftp	file transfer	1.0	-	parseconf.c line 62
binutils	binary utilities	2.32	2019–14,444	readelf.c line 13347
libming	flash library	0.4.8	2019–9113	decompile.c line 381
libjpeg	jpeg library	9a	2018–11,213	rdppm.c line 153
pngquant	png compressor	2.7.0	2016–5735	rwpng_read_image24_libpng.c line 238
ffjpeg	encoder	24 February 2020	2020–13,438	jfif.c line 748
zziplib	decoder	0.13.67	2018–6381	mmapped.c line 685
exiv2	metadata library	0.26	2018–5772	image.c line 468
ngiflib	gif decoder	0.4	2018–10,677	ngiflib.c line 808

5.2. Results

The first aim of the experiment is to verify the efficiency of high-quality sample generation. The tests are performed 10 times on 12 software programs, each time for 10 h. We observe the change in the average number of high-quality samples that are generated over time. The results are presented in Figure 5. In general, the result of PSOFuzzer is better. For libjpeg, binutils and ngiflib, the number of high-quality samples generated by PSOFuzzer is substantially greater than that generated by AFLGo. For tar, binutils, libming and ffjpeg, the number of high-quality samples generated by PSOFuzzer is substantially greater than that generated by Sidewinder.

The second aim of the experiment is to verify the effectiveness of PSOFuzzer in triggering vulnerability. We analyzed whether fuzzers can trigger vulnerabilities each time. When a vulnerability is triggered multiple times in a single test, the trigger is recorded only once. The results are shown in Table 3. PSOFuzzer triggered vulnerabilities more than 4 times for each software. AFLGo triggered vulnerabilities fewer times, and Sidewinder triggered the least vulnerabilities. For binutils and libjpeg, AFLGo did not trigger any vulnerabilities. Sidewinder only triggered vulnerabilities in gzip, miniftp and ngiflib.

Table 3. Number of times(out of 10) that a vulnerability is triggered.

	Gzip	Tar	Miniftp	Pdfresurrect	Binutils	Libjpeg	Libming	Pngquant	Ffjpeg	Zziplib	Exiv2	Ngiflib
AFLGo	5	8	5	3	0	0	2	4	1	6	10	2
PSOFuzzer	8	10	6	7	4	6	5	7	4	8	10	6
Sidewinder	7	0	6	0	0	0	0	0	0	0	0	3

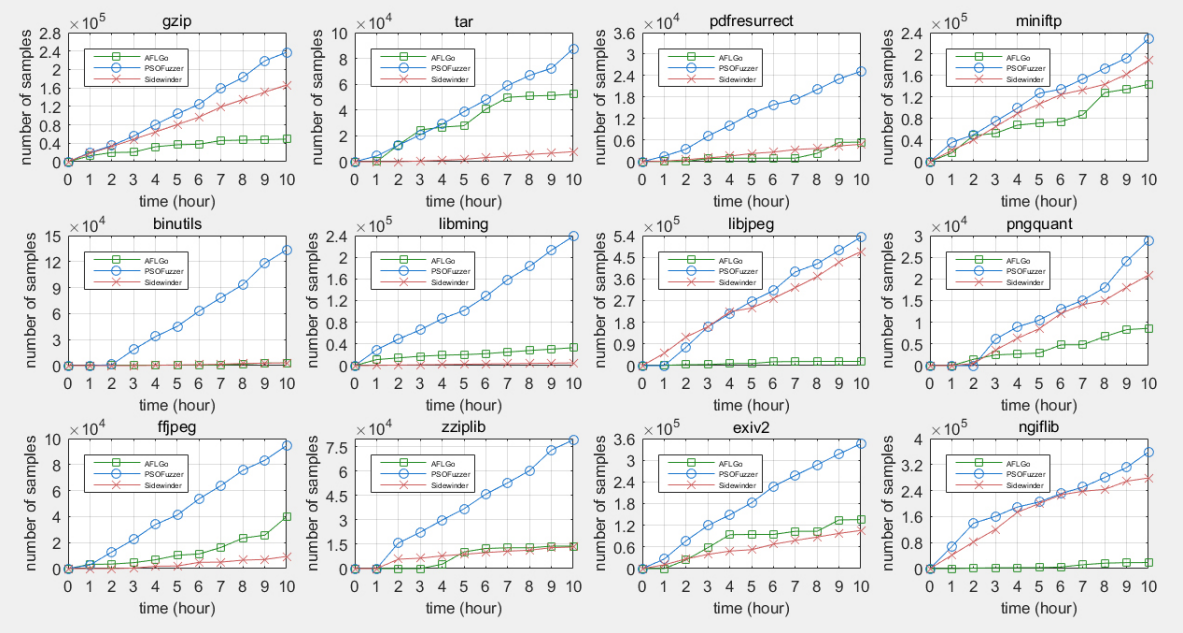

Figure 5. Growth in average number of high-quality samples over time.

5.3. Discussions

According to the data obtained in the experiment, we will perform further analysis from three aspects, including the capability of generating high-quality samples, the stability of generating high-quality samples, and the capability of triggering vulnerability.

5.3.1. Capability of Generating High-Quality Samples

High-quality samples are samples that can reach the target point when executing in the program. In target-oriented fuzzing, the larger is the number of samples that reach the suspected vulnerable points, the greater is the probability of triggering a vulnerability. AFLGo guides the test process to the target point by optimizing the scheduling mechanism. There is no connection between the mutations of the samples and the target. Sample generation is only performed by random mutation on a single seed, and the generated samples cannot learn the characteristics of other high-quality samples. Sidewinder can learn the characteristics of other high-quality samples but only between two seeds, so its learning ability is limited. The samples generated by the PSO algorithm inherit the high-quality characteristics of all the samples through the local optimal samples and the global optimal samples and quickly generate more high-quality samples, which leads to a strong correlation between the mutations of the samples and the target. As shown in Figure 5, PSOFuzzer generates the most high-quality samples.

5.3.2. Stability of Generating High-Quality Samples

To analyze the stability of the three fuzzers in the generation of high-quality samples, we calculate the relative standard deviation (RSD) of the number of high-quality samples in 10 tests according to Formula (6).

$$\sigma = \frac{\sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}}{\bar{x}} \times 100\% \tag{6}$$

Formula (6) reflects the fluctuation of the number of high-quality samples generated in 10 h in 10 tests, x_i represents the total number of high-quality samples generated in 10 h in the i_{th} test, and $\bar{x}$ represents the average number of all high-quality samples generated in 10 h. The larger the value is, the more unstable the number of high-quality samples generated. It can be seen from the Figure 6 that the curve formed by PSOFuzzer and

Sidewinder is lower than that formed by AFLGo, which indicates that PSOFuzzer and Sidewinder are more stable in generating high-quality samples. Although the stabilities of PSOFuzzer and Sidewinder are similar, PSOFuzzer generates more high-quality samples.

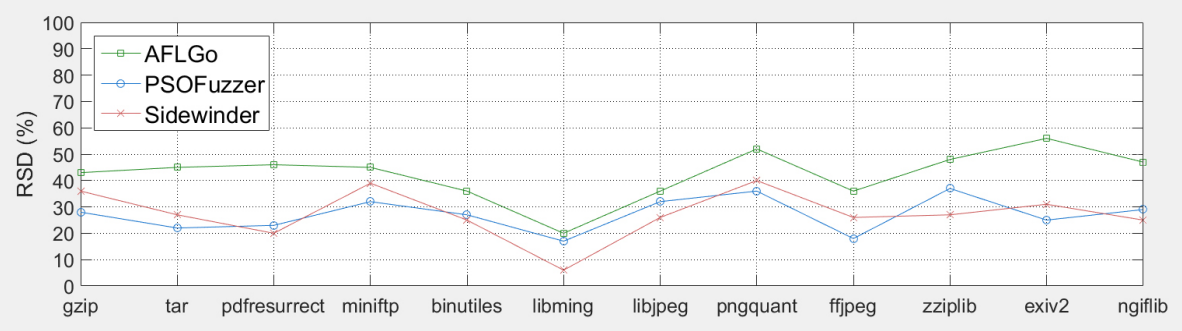

Figure 6. Stability of high-quality samples generation.

5.3.3. Capability of Triggering Vulnerabilities

The number of triggered vulnerabilities is the core index used to measure the efficiency of fuzzing tools. Figure 7 uses the data from Table 3 to draw a radar chart, which intuitively expresses the ability to trigger vulnerabilities. In Figure 7, PSOFuzzer has the largest capability coverage area and the smallest Sidewinder. Intuitively, PSOFuzzer is more capable of triggering vulnerabilities.

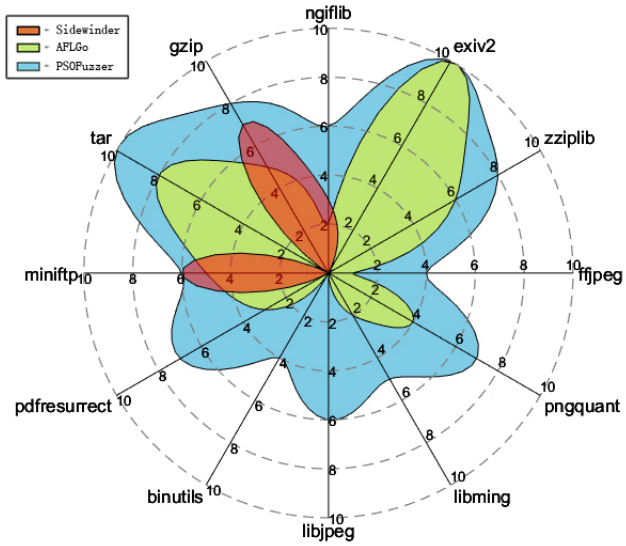

Figure 7. Ability to trigger vulnerabilities.

We apply Formula (7) to quantify the probability of triggering a specified software vulnerability in 10 h. In Formula (7), r_i represents the number of times a specified software vulnerability is triggered, N represents the number of tests, and M represents the total number of software to be tested.

$$p = \sum_{i=1}^{M} \left(\frac{r_i}{N \times M} \right) \tag{7}$$

Based on this formula, the probability that PSOFuzzer will trigger vulnerabilities is 68%, and the probability that AFLGo and Sidewinder will trigger vulnerabilities are 38% and 13%, respectively. The probabilities that PSOFuzzer will trigger vulnerabilities in the 12 designated software programs is 79% higher and 423% higher than that of AFLGo and

Sidewinder, respectively. Therefore, PSOFuzzer can trigger vulnerabilities with a greater probability on the assessed software.

6. Challenges/Limitations

We implement a target-oriented fuzzing tool, PSOFuzzer. Compared with traditional methods, PSOFuzzer can detect vulnerabilities more effectively, but there are still some shortcomings and challenges.

(1) The parameters and specified mutation algorithms that are employed in the swarm intelligence algorithm are mainly selected randomly, which cannot be adjusted by real-time test information. Thus, random selection may degrade the performance of the test.
(2) PSOFuzzer uses format constraint technology, which relies on known sample format information. In the absence of sample format information, the sample mutation is blind, which reduces the probability of triggering vulnerabilities.
(3) PSOFuzzer needs to specify the suspected vulnerability points in advance, which will increase the probability of false negatives when the suspected vulnerability points are inaccurate.

7. Future Directions

Based on the above shortcomings and challenges, we will continue to study the following three aspects in the future to further improve the performance and adaptability of the tool:

(1) Optimize the strategy for choosing the mutation algorithms and PSO parameters by changing the value of the sample distance to the target to further improve the efficiency of fuzzing.
(2) Collect the execution path by modifying a sample byte by byte and analyze the characteristics of the paths before and after modification. The sample format can be obtained in reverse to improve the adaptability of PSOFuzzer to the program without sample format information.
(3) By analyzing the relevant elements in the code that hide vulnerability and extracting them as the input data to train the graph embedding model, the trained model can be employed to identify and locate vulnerabilities in the software.

8. Conclusions

Software vulnerabilities have been increasingly exposed in recent years, and the impact of vulnerabilities is becoming increasingly serious. Fuzzing is currently the most effective method of vulnerability detection, but high-quality test samples are difficult to generate. In this paper, we implement a target-oriented fuzzing system named PSOFuzzer, which transforms fuzzing into a mathematical optimization problem. PSOFuzzer maps the elements in fuzzing to the PSO algorithm and leverages PSO optimization to generate more high-quality samples to execute to the target point. Additionally, PSOFuzzer uses format constraint technology to enhance the penetration ability of the samples in the program and improve the probability of trigger vulnerabilities. To verify the effectiveness of PSOFuzzer, we perform testing on 12 different software programs. The experimental results show that PSOFuzzer can steadily generate more high-quality samples and has a 79% higher and 423% higher probability of triggering vulnerabilities in designated software programs than AFLGo and Sidewinder, respectively.

Author Contributions: Methodology, C.C. and B.C.; software, C.C. and H.X.; validation, C.C. and H.X.; writing—original draft preparation, C.C.; writing—review and editing, B.C.; project administration, B.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Miller, B.P.; Fredriksen, L.; So, B. An Empirical Study of the Reliability of Unix Utilities. *Commun. ACM* **1990**, *33*, 32–44. [CrossRef]
2. Zzuf Fuzzer. Available online: https://github.com/samhocevar/zzuf (accessed on 9 February 2020).
3. Peach Fuzzer. Available online: http://www.peachfuzzer.com/products/peach-platform (accessed on 9 February 2020).
4. Spike Fuzzer Platform. Available online: http://resources.infosecinstitute.com/fuzzer-automation-with-spike/ (accessed on 28 August 2019).
5. Röning, J.; Laakso, M.; Takanen, A.; Kaksonen, R. PROTOS-Systematic Approach to Eliminate Software Vulnerabilities. In Proceedings of the Invited Presentation at Microsoft Research, Seattle, WA, USA, 17 September 2002.
6. Kamel, N.; Lanet, J.L. Analysis of HTTP Protocol Implementation in Smart Card Embedded Web Server. *Int. J. Inf. Netw. Secur.* **2013**, *2*, 417. [CrossRef]
7. Alimi, V.; Vernois, S.; Rosenberger, C. Analysis of Embedded Applications by Evolutionary Fuzzing. In Proceedings of the International Conference on High Performance Computing & Simulation (HPCS), Bologna, Italy, 21–25 July 2014; pp. 551–557.
8. Van, F.; Hond, B.; Torres, A.C. Security Testing of GSM Implementations. In Proceedings of the International Symposium on Engineering Secure Software and Systems, Munich, Germany, 26–28 February 2014; pp. 179–195.
9. American Fuzzy Lop. Available online: https://github.com/mirrorer/afl (accessed on 9 February 2020).
10. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* **2017**, *45*, 489–506. [CrossRef]
11. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. Collafl: Path sensitive fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–23 May 2018; pp. 679–696.
12. She, D.; Pei, K.; Epstein, D.; Yang, J.; Ray, B.; Jana, S. NEUZZ: Efficient fuzzing with neural program smoothing. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 803–817.
13. Lyu, C.; Ji, S.; Zhang, C.; Li, Y.; Lee, W.H.; Song, Y.; Beyah, R. MOPT: Optimized Mutation Scheduling for Fuzzers. In Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019; pp. 1949–1966.
14. Cadar, C.; Dunbar, D.; Engler, D.R. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA, 8–10 December 2008; Volume 8, pp. 209–224.
15. Chipounov, V.; Kuznetsov, V.; Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM Sigplan Not.* **2011**, *46*, 265–278. [CrossRef]
16. Cha, S.K.; Avgerinos, T.; Rebert, A.; Brumley, D. Unleashing Mayhem on Binary Code. In Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 21–23 May 2012; pp. 380–394.
17. Godefroid, P.; Levin, M.Y.; Molnar, D.A. Automated Whitebox Fuzz Testing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 10–13 February 2008; Volume 8, pp. 151–166.
18. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvil, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting Fuzzing through Selective Symbolic Execution. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 21–24 February 2016.
19. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In Proceedings of the USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
20. Huang, H.; Yao, P.; Wu, R.; Shi, Q.; Zhang, C. PANGOLIN: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In Proceedings of the Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1613–1627.
21. Kim, K.; Jeong, D.R.; Kim, C.H.; Jang, Y.; Shin, I.; Lee, B. HFL: Hybrid Fuzzing on the Linux Kernel. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 23–26 February 2020.
22. Chen, C.; Cui, B.; Ma, J.; Wu, R.; Guo, J.; Liu, W. A systematic review of fuzzing techniques. *Comput. Secur.* **2018**, *75*, 118–137. [CrossRef]
23. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2017.
24. Ispoglou, K.; Austin, D.; Mohan, V. FuzzGen: Automatic Fuzzer Generation. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 2271–2287.
25. Yue, T.; Wang, P.; Tang, Y.; Wang, E.; Yu, B.; Lu, K.; Zhou, X. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 2307–2324.
26. Chen, H.; Guo, S.; Xue, Y.; Sui, Y.; Zhang, C.; Li, Y.; Wang, H.; Liu, Y. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 2325–2342.

27. Aschermann, C.; Schumilo, S.; Abbasi, A.; Holz, T. Ijon: Exploring Deep State Spaces via Fuzzing. In Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA, 18–21 May 2020; pp. 1597–1612.
28. Fioraldi, A.; Maier, D.; Eifeldt, H.; Heuse, M. AFL++: Combining Incremental Steps of Fuzzing Research. Available online: https://www.usenix.org/conference/woot20/presentation/fioraldi (accessed on 25 January 2021).
29. Haller, I.; Slowinska, A.; Neugschwandtner, M.; Bos, H. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In Proceedings of the USENIX Security Symposium, Washington, DC, USA, 14–16 August 2013; pp. 49–64.
30. Neugschwandtner, M.; Milani Comparetti, P.; Haller, I.; Bos, H. The BORG: Nanoprobing Binaries for Buffer Overreads. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, San Antonio, TX, USA, 2–4 March 2015; pp. 87–97.
31. Böhme, M.; Pham, V.T.; Nguyen, M.D.; Roychoudhury, A. Directed Greybox Fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October 2017; pp. 2329–2344.
32. Embleton, S.; Sparks, S.; Cunningham, R. Sidewinder: An Evolutionary Guidance System for Malicious Input Crafting. Available online: http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Embleton.pdf (accessed on 9 February 2020).
33. Kennedy, J.; Eberhart, R. Particle swarm optimization. In Proceedings of the IEEE ICNN'95-International Conference on Neural Networks, Perth, Australia, 27 November 1995; Volume 4, pp. 1942–1948.
34. Shaukat, K.; Luo, S.; Varadharajan, V.; Hameed, I.A.; Chen, S.; Liu, D.; Li, J. Performance Comparison and Current Challenges of Using Machine Learning Techniques in Cybersecurity. *Energies* **2020**, *13*, 2509. [CrossRef]
35. Dar, K.S.; Luo, S.; Varadharajan, V.; Hameed, I.A.; Xu, M. A Survey on Machine Learning Techniques for Cyber Security in the Last Decade. *IEEE Access* **2020**, *8*, 222310–222354.
36. Pailoor, S.; Aday, A.; Jana, S. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In Proceedings of the 27th USENIX Security Symposium (Security), Baltimore, MD, USA, 15–17 August 2018; pp. 729–743.
37. Syzkaller. Available online: https://github.com/google/syzkaller (accessed on 20 September 2020).
38. Gzip. Available online: http://mirror.keystealth.org/gnu/gzip/ (accessed on 20 September 2020).
39. Goahead. Available online: http://mirror.keystealth.org/gnu/tar/ (accessed on 20 September 2020).
40. Pdfresurrect. Available online: https://github.com/enferex/pdfresurrect/ (accessed on 20 September 2020).
41. Miniftp. Available online: https://github.com/skyqinsc/MiniFtp/ (accessed on 20 September 2020).
42. Binutils. Available online: http://mirror.keystealth.org/gnu/binutils/ (accessed on 20 September 2020).
43. Libming. Available online: https://github.com/libming/libming (accessed on 20 September 2020).
44. Libjpeg. Available online: http://www.ijg.org/ (accessed on 20 September 2020).
45. Pngquant. Available online: https://github.com/kornelski/pngquant (accessed on 20 September 2020).
46. Ffjpeg. Available online: https://github.com/rockcarry/ffjpeg (accessed on 20 September 2020).
47. Zziplib. Available online: https://enterprise.dejacode.com/packages/public/d9018a4f-2e88-498d-8cb9-d52680be48f5/ (accessed on 20 September 2020).
48. Exiv2. Available online: https://www.exiv2.org/archive.html (accessed on 20 September 2020).
49. Ngiflib. Available online: https://github.com/miniupnp/ngiflib (accessed on 20 September 2020).