electronics

MDPI

Article

WolfFuzz: A Dynamic, Adaptive, and Directed Greybox Fuzzer

Qingyao Zeng [1], Dapeng Xiong [2,*], Zhongwang Wu [2], Kechang Qian [2], Yu Wang [2] and Yinghao Su [1]

[1] Institute of Graduate, Space Engineering University, Beijing 101416, China
[2] Institute of Aerospace Information, Space Engineering University, Beijing 101416, China
* Correspondence: xiongdapeng@hgd.edu.cn

Abstract: As the directed greybox fuzzing (DGF) technique advances, it is being extensively utilized in various fields such as defect reproduction, patch testing, and vulnerability identification. Nevertheless, current DGFs waste a significant amount of resources due to their simplistic distance definitions and overly straightforward energy distribution for the seeds. To address these issues, a dynamic distance-weighting-based distance estimation strategy is proposed first, which facilitates strategies for seed distribution that take energy into consideration. Second, to overcome the limitations of current seed energy distribution strategies, the gray wolf optimizer (GWO) is improved by integrating four strategies, leading to the development of the improved gray wolf optimizer (IGWO). Lastly, an adaptive search algorithm is proposed, and the WolfFuzz prototype tool is implemented. In vulnerability recurrence scenarios, WolfFuzz is 3.2× faster on average compared with the baseline and reproduces 76.4% of existing bugs faster. WolfFuzz also discovers nine different types of bugs in seven real-world programs.

Keywords: directed fuzzing; vulnerability mining; software security; fuzzing

Citation: Zeng, Q.; Xiong, D.; Wu, Z.; Qian, K.; Wang, Y.; Su, Y. WolfFuzz: A Dynamic, Adaptive, and Directed Greybox Fuzzer. *Electronics* **2024**, *13*, 2096. https://doi.org/10.3390/electronics13112096

Academic Editor: Valentina E. Balas

Received: 22 April 2024
Revised: 22 May 2024
Accepted: 27 May 2024
Published: 28 May 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Fuzzing has gained significant attention and utilization in both academic and industrial settings in recent years because of its powerful capabilities in automated vulnerability detection and validation [1]. Based on their goals, two types of fuzzing exist: coverage-guided fuzzing and target-guided fuzzing. The core concept of coverage-guided fuzzing is based on the notion that enhancing code coverage enhances the likelihood of discovering undisclosed vulnerabilities in a software program. However, vulnerabilities are frequently hidden in a complex manner, making it challenging to identify them as software systems become more intricate. When vulnerabilities grow harder to reach, traditional coverage-guided fuzzing faces increasing difficulties in detecting them. In response to this issue, target-guided fuzzing has been devised as a solution.

Target-guided fuzzing, or DGF, focuses on investigating particular areas and repeatedly generates seeds that aim at susceptible program sites until the target is met. This technique is intended to target specific program locations. DGF was revolutionized by AFLGo [2], which is considered a seminal work in the field. It laid the foundation for later techniques in this area [3–9]. However, AFLGo has other shortcomings that ultimately reduce the fuzzing effectiveness, such as imprecise distance definitions and excessively simplistic seed energy distribution algorithms.

DGF divides the fuzzing process into two phases, exploration and exploitation, and covers as many paths as possible in the exploration phase and gets the fuzzer as close as possible to the target code areas in the exploitation phase. Most articles follow this idea, while some articles split the two phases into three.

In an effort to trigger more bugs, recent DGFs have improved on these flaws.

First, for improvements in static analysis, Hawkeye [3] captures call graphs, functions, and distances between basic blocks and the target to provide dynamic metrics. These metrics are then utilized for seed prioritization, energy scheduling, and adaptive mutation.

DeFuzz [5] builds a Bi-LSTM deep learning [10] model with labeled C/C++ functions and uses it to anticipate potentially vulnerable functions in the target application. WindRanger [11] uses static analysis to identify possible deviations from the target position. During the fuzzing process, executed basic blocks are combined to identify deviant ones. It modifies distance values based on the difficulty of meeting criteria of deviating fundamental blocks; then, it sets seed priorities and phase-switching techniques. WolfFuzz, on the other hand, dynamically modifies distances during fuzzing based on the amount of time spent doing so. BEACON [6] uses lightweight static analysis to prune irrelevant execution pathways, resulting in significant and proven results. Halo [12] limits the input space by inferring invariant quantities from the input.

Second, for improvements in dynamic analysis, FishFuzz [13] splits fuzzing into three steps and constantly alters the emphasis among these phases to efficiently allocate fuzzing time. WolfFuzz, like FishFuzz, transitions between three phases. However, unlike FishFuzz, WolfFuzz uses a heuristic technique to filter the seed queue and sub-queues to assure priority for each specific target. LABRADOR [14] calculates the distance between firmware execution paths and the target using the strings returned from corpus inputs. PDGF [15] prunes branches by segmenting the target program into predecessor and non-predecessor regions. SCDF [16] presents sequence-coverage-directed fuzzing, which generates inputs that sequentially reach each statement in a set of target statement sequences and triggers program errors. DeepGo [17] models DGF as the process of reaching the target position through specific path transitions. Deep neural networks predict the reward of path transitions, and reinforcement learning [18] combines historical and predicted path transitions to generate the best path. LeoFuzz [19] creates two queues for the exploration and exploitation phases, and it suggests a new energy scheduling method to prevent slipping into or ignoring local optima. G-Fuzz [20] estimates the distance of code on the target-reachable path, decreases computational cost with breadth-first search technique, and finds indirect calls using a type analysis. Overall, some of these methods can be combined with WolfFuzz for better performance.

Nevertheless, some inadequacies persist [13,21–25]. Firstly, while some fuzzers provide more accurate definitions for distance metrics, they neglect the fact that distinct branches may have varying arrival probabilities in certain situations. This can lead to a significant loss of time for branches with difficult criteria. Second, there is little doubt that fuzzing with defined exploration and use durations will waste resources. As a result, depending on the fuzzing performance and state, it is required to dynamically transition between the two phases of exploration and usage.

WolfFuzz is proposed as a dynamic, adaptive, and directed greybox fuzzer in light of these problems. First, WolfFuzz adjusts the distance based on each round of fuzzing performance, treating the distance as a dynamic distance. Secondly, a heuristic approach is used to strike the targets after they are ranked according to the seeds that are currently accessible. Lastly, an adaptive search algorithm can dynamically transition between three states by considering the phase duration and the current fuzzing state. Specifically, the adaptive search algorithm divides the dynamic analysis procedure into three phases: initial exploration, triggering objective, and crash detection. The inspiration for these three phases comes from the hunting of wolves: imitating the process of picking a target, slowly approaching, and concentrating on hunting.

WolfFuzz has many real-world applications. In patch tests, WolfFuzz is used to verify whether the modified patch fixes bugs or introduces new bugs. In vulnerability recurrence, WolfFuzz is used to reproduce crashes and generate proof of concept (POC). When assisting with human analysis, WolfFuzz can verify the results of human analysis and assist with program testing. In addition, WolfFuzz can also detect special bugs.

WolfFuzz is derived from AFL and incorporates the concept of static distance as defined by FishFuzz. Among the other fuzzer-picked targets, seven common real-world programs are selected, covering six different input formats: png, mp4, tiff, swf, xml, and elf. According to the findings, WolfFuzz outperforms coverage-based guided fuzzers in terms

of performance. When contrasted with contemporary DGFs and state-of-the-art fuzzers, WolfFuzz demonstrates an average speed improvement of 3.20 times over the baseline and is able to reproduces 76.4% of known bugs at a faster rate. Additionally, WolfFuzz identifies nine distinct bug categories across seven real-world applications.

In summary, the main contributions of this paper comprise the following:

- WolfFuzz: a dynamic, adaptive, and directed fuzzing framework that dynamically transitions between three distinct phases;
- A distance calculation method that mitigates the impact of the distance factor on seed energy distribution;
- Numerous comparative experiments have been conducted against the state-of-the-art, revealing nine different types of vulnerabilities.

The rest of this article is organized as follows. An introduction to the background of DGF is provided in Section 2. Subsequently, Section 3 firstly introduces the overall design and flow of WolfFuzz and, secondly, explains the key points of it, including a dynamic distance weighting strategy, IGWO, and an adaptive search algorithm. Finally. Section 4 presents the evaluation of WolfFuzz, while the conclusions and future work are presented in Section 5.

2. Background

Owing to its unique characteristics, DGF has swiftly captured the attention of experts in related disciplines following the emergence of AFLGo. Nevertheless, numerous DGF sites may require further attention. This section presents the pertinent background information that underpins the development of WolfFuzz in this paper.

2.1. Distance Measurement

Distance measurements are used in targeted fuzzing to reduce the distance between the sample and the target, therefore raising the possibility of a crash and revealing vulnerabilities. Nevertheless, inaccurate definitions of distance can make it more difficult for the sample to reach the target, which reduces the efficacy of fuzzing. AFLGo combines distance measures to many targets and uses the average distance of those targets to direct its testing. Mutating samples, however, frequently only hit one target, wasting resources.

FishFuzz [13] enhances distance definition, initially addressing indirect call issues. Yet akin to AFLGo, it overlooks the different probabilities of branch reachability. In this paper, branch reachability probability is understood as the probability that the branch where a certain target location is located in a control flow graph (CFG) is visited by the seeds. A simple CFG example is shown in Figure 1, where nodes represent basic blocks and directed edges represent execution paths in the CFG. Then, there are two branches from A, A-B or A-C, due to different jump conditions: for example, if some variable x in basic block A ranges from 0 to 10 when $0 < x < 1$, the jump is from basic block A to basic block C; when $1 < x < 10$, the jump is from basic block A to basic block B. Then, we consider the probability of being able to visit branch A-B at 0.9 and the probability of being able to visit branch A-C at 0.1. In the scenario shown in Figure 1, if the goal is point F, two branches, A-C-F and A-B-D-F, are reachable from A. Using AFLGo's distance calculation method, the distance to A-B-D-F is $(2 + 2 + 1 + 0)/4 = 1.25$, while the distance to A-C-F is $(2 + 1 + 0)/3 = 1$, favoring the A-C-F branch. With FishFuzz's distance calculation, A-B-D-F is 3 and A-C-F is 2, also favoring the A-C-F branch. However, by incorporating branch reachability probabilities A to C at 0.1, A to B at 0.9, and B to D at 0.5, the likelihood of A-B-D-F becomes 0.45, and the likelihood of A-C-F becomes 0.1. Consequently, the A-B-D-F branch facilitates easier target triggering to induce a crash.

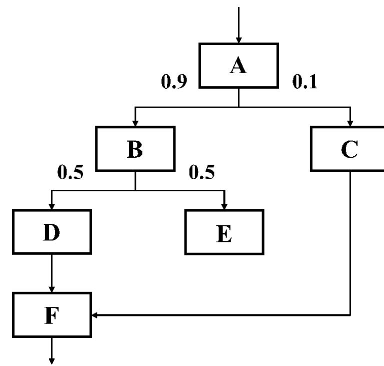

Figure 1. A simple CFG example.

As a result, depending merely on a count of basic block edges visited is insufficient for accurate distance measurement. It is critical to consider the probabilities of branch reachability when revising distance measures in fuzzing. WolfFuzz solves this problem by implementing a dynamic distance weighting strategy based on FishFuzz's distance definition. When calculating the distance, the probabilities of branch reachability are considered, and the distance is continuously updated through feedback during the fuzzing process.

2.2. Energy Distribution

Modern advanced fuzzers usually increase the risk of mutation of valuable samples when doing fuzzing as a strategy to speed up the triggering of the crash. Energy is a measure of a sample's chance of mutation: the higher the energy, the more likely the seed will be mutated. How to allocate energy reasonably is a problem, and most DGFs allocate energy based on simulated annealing (SA) [26].

SA is a probability-based stochastic optimization algorithm that affects the probability of finding the optimal solution from the adjacent solution space of the current solution through a change of the temperature parameter. The higher the temperature, the higher the probability of not finding the optimal solution from the proximity solution space of the current solution. The lower the temperature, the more inclined the algorithm is to find the optimal solution from the proximity solution space of the current solution. In directed fuzz testing, the optimal solution means the seed that is closest to the target location, and the proximity solution space of the current solution means the set composed of seeds obtained by mutating the current seed. The principle of SA is to start with a high initial temperature and gradually decrease the temperature until the stopping condition given in the paper is reached. This means that the fuzzer has the probability to allocate energy to other seeds, even though they are not currently performing well.

However, SA converges slowly, and it requires patience to wait for the emergence of the optimal solution. At the same time, the algorithm has lots of adjustable parameters: when the parameters are not set well, there is a possibility of falling into a local optimum, and it is necessary to take some additional special strategies to avoid this situation. WolfFuzz uses IGWO for energy distribution and combines the strategy of GWO [27] with the idea of simulating the fisher fishing (SFOA), which, with fewer parameters, accelerates the convergence speed and avoids falling into the local optimum by introducing a nonlinear control parameter, SFOA's idea, and a terminal elimination mechanism.

2.3. Search Strategy

In current DGFs, the allocation of time for exploration and exploitation is contradictory. If the exploration period is extended, new mutation samples may go unexecuted; conversely, a quick exploration phase may result in an inadequate corpus for future exploitation.

LeoFuzz [19] is a tool for fuzzing for multiple targets. In terms of how to allocate the time between exploration and exploitation, it proposes to use two queues to store seeds for exploration and for exploitation, respectively. LeoFuzz performs the exploration

and exploitation phases based on the number of seeds stored in the two queues. If the percentage of seeds located in the exploration queue out of the total seeds is too high, LeoFuzz switches to the exploitation phase. If the fuzzer has no new seeds added to the exploitation queue for a long time, it switches to the exploration phase. Through the adaptive coordination of the two queues, LeoFuzz improves on the drawback of static specification of exploration and exploitation, as in AFLGo. FishFuzz [13], inspired by the trawl fishing technique, splits the exploration phase into inter-function exploration and intra-function exploration, thus dividing the whole process into three phases. At the beginning of fuzzing, FishFuzz prioritizes the inter-function exploration phase to explore the function (expanding the net); when no new function arrives, FishFuzz enters the intra-function exploration phase to focus on the target arrival (closing the net). Once a seed reaches a new target, FishFuzz enters the exploitation phase and starts trying to trigger the target (catching fish). During the entire period, once a new function is accessed, FishFuzz immediately switches to the inter-function exploration phase to continue the function exploration. FishFuzz can rationally allocate the execution time between exploration and exploitation by setting appropriate transfer conditions in the three phases.

The above tools provide solutions on how to allocate the time between exploration and exploitation, but despite the adaptive switching, they do not guarantee that the current seed is suitable for the current target. WolfFuzz proposes an adaptive search algorithm to solve this problem. Taking inspiration from wolf hunting, the adaptive search algorithm divides the dynamic analysis process into three phases, rationally allocates the time of the three phases by setting unique transfer conditions, and maximizes the efficiency of fuzzing by setting different target sub-queues for different targets.

In summary, there are still three challenges that need to be addressed:

1. Design a distance measurement method that can initially alleviate the problem of imprecise distance measurement due to differences in the probabilities of branch reachability;
2. Design an energy distribution algorithm with fewer adjustable hyperparameters, good convergence speed, and global search capability;
3. Design a search strategy that not only balances the time allocation problem between exploration and exploitation during fuzzing but also improves the adaptability of the current seed to the current target.

3. Methodology

This section outlines the principle and fundamental flow of WolfFuzz. Subsequently, the approach is elaborated upon in detail.

3.1. WolfFuzz's Design

In this paper, a DGF that uses a dynamic distance weighting strategy is proposed, and WolfFuzz, a prototype system based on AFL, is demonstrated. The system is divided into three modules: a static analysis module, a dynamic analysis module, and a results analysis module, each of which serves a specific function. The static analysis module basically determines the target's location following code preprocessing and estimates the distance to the target based on instruments. The dynamic analysis module is in charge of running DGF with an adaptive search algorithm and logging the findings, while the results analysis module is in charge of analyzing the results and presenting them.

The whole workflow is shown in Figure 2, following the standard fuzzing program flow. In the static analysis phase, firstly, a target program containing source code needs to be given, and then, we use the tools given by AFL for instruments ①. The next phase ② prepares the detection program for code preprocessing and extracts initial information useful for WolfFuzz, including the generation of function call graphs (FCGs) and control flow graphs (CFGs). Next, the target program is compiled and detected using a specific sanitizer ③ to generate a preliminary distance map ④; this map will be corrected during the dynamic analysis phase. In the dynamic analysis phase, the fuzzer controls the fuzzing process to switch between three phases based on a specific search strategy ⑤. During

this period, the seeds in the initialized corpus are subjected to energy distribution ⑥ and mutation ⑦. Additionally, the selected seeds and the distances between the seeds and the target computed by the corrected distance map are all passed into the search strategy, which plays a role in all time of the dynamic analysis phases, eventually outputs the final results ⑧, and carries out the final experimental evaluation ⑨.

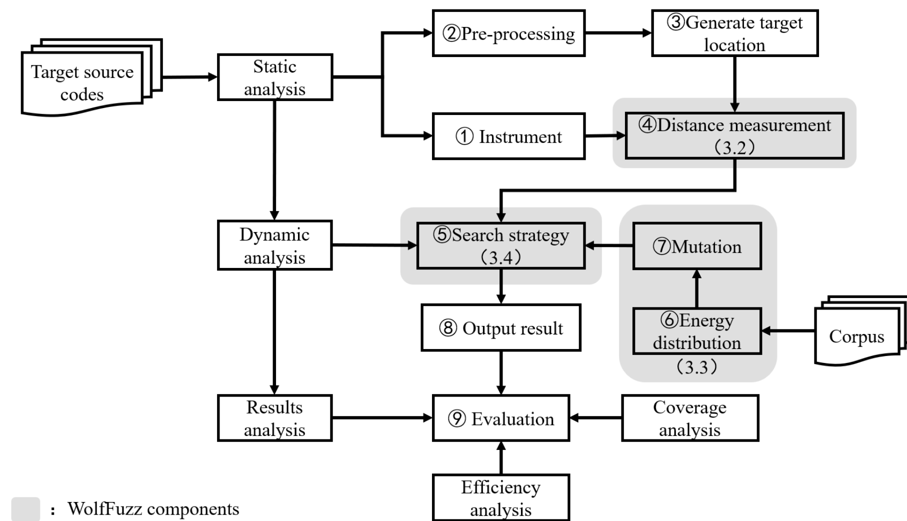

Figure 2. WolfFuzz overall framework: the parts covered by gray are the key approaches of WolfFuzz.

In the following subsections, dynamic distance weighting strategies (Section 3.2), IGWO (Section 3.3), and an adaptive search algorithm (Section 3.4) are proposed, respectively, to alleviate the three issues raised at the end of Section 2.

3.2. Dynamic Distance Weighting Strategy

AFLGo uses the reconciled mean to calculate the sample's common distance to numerous objectives. However, when the targets are vastly varied, a single assessment index cannot accurately describe the accuracy of matching this sample to a certain target. Based on this, this article uses a single target for distance estimation and continually corrects the distance between the sample and the target using the fuzzing procedure to improve the hit rate. The dynamic distance weighting strategy breaks down the estimated distance into three parts: distance of neighboring functions, distance between arbitrary functions, and distance after being dynamically corrected.

Distance of neighboring functions: Neighboring functions, for which one of the two functions is the caller or called of the other, are usually represented by a line between the two functions in the FCG. To calculate the distance between neighboring functions, a CFG is also generated for more accurate estimates. Specifically, assuming that the execution order of a path function is $f_1 \rightarrow f_2 \rightarrow f_3 \ldots \rightarrow f_d$, to calculate the distance between f_i and f_{i+1}, the minimum number of edges needs to be calculated between the entry point B_1 of f_i and the basic block B_f, which we can call f_{i+1}. The exact depiction is displayed here.

$$NeighDff(f_i, f_{i+1}) = \begin{cases} MinDbb\big(B_1, B_f\big), & if \exists B_f \in f_i \\ \infty \quad , & \text{otherwise} \end{cases} \tag{1}$$

where $NeighDff(f_i, f_{i+1})$ denotes the distance between neighboring functions f_i and f_{i+1}, and $MinDbb(B_1, B_f)$ denotes the minimal number of edges that must be passed between two basic blocks B_1 and B_f. This formula states that if f_i contains a basic block B_f which calls f_{i+1}, the distance between two neighboring functions f_i and f_{i+1} is the minimum number of edges that need to be passed. Otherwise, the distance is infinite and unreachable.

Distance between arbitrary functions: After computing neighboring functions' distances, they are utilized as weights for the directed edges of two neighboring functions in

the FCG, and Dijkstra's algorithm is used to find the set of shortest pathways between any functions as well as the smallest distance between any functions. The exact depiction is displayed below.

$$AnyDff(f_a, f_b) = \sum_{f_i, f_{i+1} \in ffSet(f_a, f_b)} NeighDff(f_i, f_{i+1}) \quad (2)$$

where $AnyDff(f_a, f_b)$ signifies the distance between two arbitrary functions f_a and f_b, and $ffset(f_a, f_b)$ is the set of the shortest pathways between the two arbitrary functions f_a and f_b, which are constructed using Dijkstra's method.

Because the target program will call a sequence of functions after inputting a seed, the distance between the target and the function closest to the target is used as the static distance between the seed and the target, as seen below.

$$StatDfs(f, s) = \begin{cases} \min_{f \in \mathbf{s}Set(\mathbf{s})} AnyDff(f, f_\mathbf{s}), & if \quad f \notin sSet(\mathbf{s}) \\ 0 \quad , & \text{otherwise} \end{cases} \quad (3)$$

where $sSet(s)$ represents the set of functions visited by the target program after inputting the seed s, and $StatDfs(f, s)$ denotes the static distance between the target function f and the seed s. When f is contained in the set $sSet(s)$, the distance is zero, indicating that the seed s has reached the target function f.

Distance after dynamic correction: After computing the static distance between the seed and the target function, the distance may be distorted since the CFG simply uses the number of passing basic blocks as the distance basis without considering the probability of branch reachability. Here, the dynamic distance coefficient $k(s, f)$ is defined and stated as follows.

$$k(f, s) = r_1^{(n_\alpha + n_\beta + n_\delta)} * r_2^{n_\omega}, r_1 \in (1, 2), r_2 \in (0, 1) \quad (4)$$

where n_α, n_β, n_δ, n_ω are the number of times the seed s participated in the fuzzing as the four classes in IGWO (Section 3.3), and r_1, r_2 are the distance factors, whose values represent the speed of dynamic correction. Now, $DynaDfs(f, s)$ (the distance after dynamic correction) is defined as follows.

$$DynaDfs(f, s) = k(f, s) * StatDfs(f, s) \quad (5)$$

This equation shows that the distance of the seed from the target is affected by the number of times it has led the population in fuzzing as an α, β, and/or δ individual. The more times it has led the population in fuzzing, the smaller its value. Accordingly, the distance of the seed from the target is affected by the number of times it has led the population in fuzzing as an ω individual. The higher the number of times, the closer the computed distance is and the more likely the seed is to be an α, β, or δ individual. Dynamic distance correction can help to reduce distance measurement inaccuracies caused by the probability of branch reachability.

3.3. IGWO

Distributing energy to interesting seeds during mutation can hasten the identification of significant results. IGWO can dramatically improve fuzzing efficiency by distributing energy to seeds in the seed queue that are waiting to be input. This tailored energy distribution strategy can assist with prioritizing and accelerating the examination of potentially significant corpora, resulting in more effective fuzzing results. Note that in this subsection, as part of the terminology of the population optimization algorithm that is introduced, samples and individuals are denoted as seeds in fuzzing.

In the context of optimization algorithms, the optimization algorithm for SFOA has significant global search capabilities but converges slowly due to its high unpredictability. On the other hand, GWO converges quickly, but it may converge prematurely and

become trapped in local optima as it optimizes around the top three solutions (α, β, and δ individuals).

In this subsection, IGWO is merged with SFOA concepts while keeping the essence of GWO. The fitness function is dependent on the distance between the current sample and the target. In addition, sample discretization is used, a nonlinear control parameter is introduced, and a terminal elimination mechanism is provided: Figure 3 depicts the full process. In the first step, the samples are subjected to a discretization operation and encoded populations, and the individual fitness of each sample to the current target is calculated for hierarchical stratification. In the second step, the samples are mutated using the CrossMutation method to update the individual positions, and then, the SelfMutation method is utilized to merge with SFOA. In the third step, the individual fitness of each sample to the current target is calculated again, hierarchy stratification is performed, the terminal elimination mechanism is triggered, and valid samples are added to the sub-queue of the current target. When the stopping condition is not reached, the second and third steps are repeated until the stopping condition is reached and the algorithm ends. The design principles and ideas of the key methods in IGWO are described below.

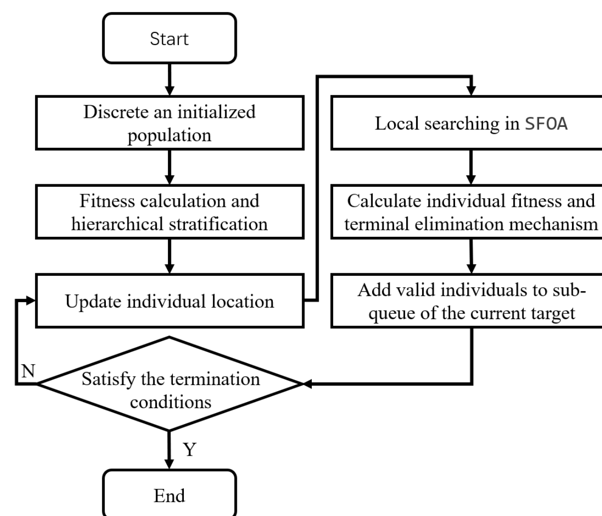

Figure 3. The overall process of IGWO.

Discretization: Because GWO is designed for optimization in continuous spaces, DGF only functions in discrete sample spaces, necessitating discretization processes. Specifically, each sample in the seed queue must be assigned to an individual in the GWO. Assuming there are n samples in the seed queue, the following equation describes the sample i at iteration t:

$$\bar{x}_i(t) = (x_i^1, x_i^2, \ldots, x_i^j, \ldots x_i^d) \quad (6)$$

where $\bar{x}_i(t)$ represents the sample i generated by the t-th iterative mutation, and x_i^j denotes the j byte in sample i, with a total of d bytes.

When updating individual positions (mutating samples), each individual undergoes crossmutation with the three preceding individuals at random, utilizing a roulette wheel selection strategy to approach the three preceding individuals' locations. This is particularly represented by the equation displayed below:

$$\bar{x}_i(t+1) = CrossMutation\left(\bar{x}_i(t), \bar{x}_j(t)\right), J \in \{\alpha, \beta, \delta\} \quad (7)$$

Table 1 shows how the mutation technique *CrossMutation* picks distinct crossmutation strategies based on sample type.

Table 1. CrossMutation.

Sample Type	Mutation Mode
binary	Randomly select α, β, γ, and some bytes of the individual to replace the corresponding bytes of the current individual; Randomly set two intersections to retain anyone after overall exchange
string	Merge two strings to produce a new seed; Randomly set one or two intersections to retain anyone after overall exchange
array	CrossMutation mutations are performed separately for each position in the array
number	summation; subtracting; multiply

Combine with SFOA: The α individual in GWO does not represent the location of the global optimal solution. To strengthen the global search, each individual is considered to have a certain probability to perform a variant search in the vicinity of its own location, as shown in the following equation.

$$\bar{x}_i(t+1) = SelfMutation(\bar{x}_i(t)) \tag{8}$$

Table 2 shows that the mutation method $SelfMutation$ selects different mutation techniques based on sample type.

Table 2. SelfMutation.

Sample Type	Mutation Mode
binary	Randomly flip some bits of binary; Randomly increase the number of binary bits; Randomly delete some binary bits
string	Change string case; Generate erroneous UTF-8 strings; Generate erroneous long UTF-8 three-byte strings
array	Change array length to numerical boundary +1; Randomize array order; Reverse array
number	Generate a random number for each element; Generate values unrelated to default; Take the opposite number for corresponding elements; Change the length of adjustable size data to numerical boundary +1; Modify the value to numerical boundary +1

Nonlinear control parameter: Because the control parameter a is represented as linearly decreasing in GWO, it does not show the real search process well in DGF. Since α, β, and δ individuals have different search strategies and abilities, the uniform value of a hinders the global search ability of α, β, and δ individuals from the balance of the local search ability. Based on this, the control parameter is transformed to a nonlinear control parameter, and the values of a for α, β, and δ individuals are determined individually, as shown below.

$$d_i = \sqrt{\left(W(\bar{x}_i(t)) - W(\bar{x}_J(t))\right)^2}, J \in \{\alpha, \beta, \delta\} \tag{9}$$

$$ds_t = d_i \Big/ \sum_{i=1}^{sizepop_J} \frac{d_i}{sizepop_J} \tag{10}$$

$$a_J = 2 - 2 \times \left(\frac{gen}{maxgen}\right)^{ds_t} \tag{11}$$

where d_i denotes the fitness difference between the current individual and the individual undergoing CrossMutation, $\bar{x}_i(t)$ denotes the current individual, $\bar{x}_J(t)$ denotes the individual undergoing CrossMutation with the current individual, and $W()$ denotes the fitness of the corresponding individual. The variable $sizepop_J$ is the number of individuals that followed α, β, δ, and ds_t, respectively, at the previous iteration. If it is the first iteration, the number is one-third of the total number of populations. The function ds_t denotes the ratio of the fitness difference of the current individual to the average fitness difference of the population. The variable a_J is the improved nonlinear control parameter, gen is the number of the current iteration, and $maxgen$ is the maximum number of iterations.

Terminal elimination mechanism: After updating the distance using the dynamic distance weighting strategy (Section 3.2), certain individuals will get increasingly distant from the goal during the iteration process, and such individuals are regarded to have lost their mutation value. As a result, in each round of updates, taking mutation efficiency into account, certain individuals with the greatest distances will be removed from the mutation process.

3.4. Adaptive Search Algorithm

When conducting fuzzing, a reasonable allocation of time for each stage can make full use of resources and maximize efficiency. Inspired by wolf hunting, the article presents an adaptive search algorithm, as demonstrated in Figure 4. First, the fuzzer conducts early exploration, during which good hunting targets are filtered out by constantly comparing each possible target, similar to the screening of the wolf pack's leader. Once the leading target has been chosen, an attempt is made to trigger the target, similar to the procedure of a wolf pack methodically encircling its prey. After successful approaching, crash detection occurs, mirroring the way wolves divide their prey.

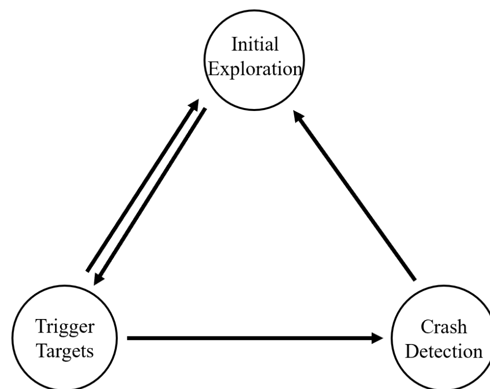

Figure 4. Flow chart of adaptive search algorithm.

The adaptive search algorithm breaks down the dynamic analysis process into three phases: initial exploration phase, trigger targets phase, and crash detection phase. Transfer conditions exist between the various phases, ensuring that computing resources are regularly moved back and forth between the three phases to maximize efficiency.

Initial exploration phase: WolfFuzz seeks to locate the first three samples that are closest to each target in the list of targets to be detected, which correspond to the α,β,δ in

IGWO. Once a target has the necessary α,β,δ persons, the trigger targets phase starts fast. Once a certain target has α,β,δ individuals corresponding to it, it is quickly added to the sub-queue of the corresponding target and enters the trigger targets phase.

Trigger targets phase: In this phase, IGWO (Section 3.3) and the dynamic distance weighting strategy (Section 3.2) are employed to try to get more samples to the target location. The goal of this phase is to increase the likelihood of triggering a crash by supplying more meaningful samples for future attempts to do so. At the same time, if no samples that can reach the target location are discovered for an extended length of time, it is assumed that this target location has rarely been covered before, and extra time is allotted for the next time a trigger for this target is done.

Crash detection phase: At this point, WolfFuzz attempts to cause a crash in order to facilitate vulnerability mining. Intuition tells us that the more interesting the samples, the higher the likelihood of a crash. Cross-mutating and Self-mutating the interesting samples gathered during the trigger targets phase increases the likelihood of creating a crash at the target location.

Although the three phases are progressive, some contingencies must be considered, making the transfer less seamless than it could be. Because of this, certain transfer conditions are established between the three phases:

1. If after a long time in the trigger targets phase there has been no seed to reach the target location, mark the target as imported and commence the initial exploration phase.
2. If after a period of time has passed in the trigger targets phase without a new target reaching the target location, enter the crash detection phase. Note that for this condition, at least one seed has reached the target location, unlike the first condition where no seeds have reached the target location for a long time.
3. If the crash detection phase has not caused a crash after some time, mark the target as not interesting, and commence the initial exploration phase.
4. When there are three or more samples in a given target sub-queue during the initial exploration phase:
   (a) If the target is marked as imported, extend the exploration time of the trigger targets phase and enter the trigger targets phase.
   (b) If the target marked as not interesting, skip this target with a certain probability and recommence the initial exploration phase.
   (c) If there are no additional marks, proceed to the trigger targets phase.

The pseudocode of the adaptive search algorithm is shown in Algorithm 1, where initial_mode corresponds to the initial exploration phase, hunt_mode corresponds to the trigger targets phase, and crash_mode corresponds to the crash detection phase.

Algorithm 1: Adaptive search algorithm

```
Input: target_queue, current_mode
Output: crash_seeds
1  while target_queue is not ∅ do
2      if current_mode is initial_mode then
3          if The target selected then
4              if target.import is True then
5                  time_for_hunt ← time_for_hunt + extended_time;
6              if target.not_interesting is True then
7                  high_probability_skip(target);
8                  reselect_target(target_queue);
9              current_mode ← hunt_mode;
10         else
11             reselect_target(target_queue);
12     else if current_mode is hunt_mode then
13         search_and_reach_target();
14         if total_hunt_time > time_for_hunt then
15             if no target reached then
16                 target.import is True;
17                 current_mode ← initial_mode;
18             else
19                 current_mode ← crash_mode;
20     else if current_mode is crash_mode then
21         crash_detection();
22         if crached then
23             drop_target(target_queue);
24             current_mode ← initial_mode;
25         else if total_crash_time > time_for_crash then
26             target.not_interesting is True;
27             current_mode ← initial_mode;
```

4. Evaluation

This section responds to three questions by using WolfFuzz and other fuzzers to evaluate some representative real-world source programs.

RQ1: How does WolfFuzz perform in terms of edge coverage?

RQ2: How good is WolfFuzz at reaching target code locations?

RQ3: How does WolfFuzz perform in vulnerability recurrence scenarios?

4.1. Environment

The environment used in this experiment is shown in Table 3.

Table 3. Experimental environment.

Classification	Configuration
SystemOS	Ubuntu 20.04.4 LTS
CPU	Intel® Xeon(R) Silver 4214 CPU @ 2.20 GHz × 48
Memory	128 GiB
Hard disk size	2.5 TB

Docker is used on the server to assign a CPU core to each test pair (fuzzer, target), and experiments are carried out. The initial seed corpora are drawn just from the limited seed libraries of AFL and AFLGo in order to maintain fairness.

A subset of publicly available fuzzers was collected at the time of writing, their compatibility with the experimental dataset was tested, and three classic fuzzers were included for comparison: namely, AFL [28], AFL++ [29], and AFLGo [2]. Since WolfFuzz is an AFL enhancement, it is only compared here with FF_AFL, which is, likewise, based on AFL advancements in FishFuzz. Table 4 lists the commit IDs for all fuzzers.

Table 4. All fuzzers and commits ID.

Fuzzer	Commit ID
AFL [28]	6103710
AFL++ [29]	143c9d1
AFLGo [2]	fa125da
TortoiseFuzz [30]	2270cab
ParmeSan [31]	fac5801
AFLFast [32]	d1d54ca
FairFuzz [33]	cf88127
EcoFuzz [34]	1fd9460
KScheduler [35]	6e78fbe
FishFuzz [13]	a72b16f

The target programs of the previously described fuzzers were carefully analyzed, and the same versions were chosen for comparative validation, as indicated in Table 5.

Table 5. The target programs.

Program	Commit ID	Corpus	Cmdline
exiv2 [36]	fa449a4	png	./exiv2 @@
MP4Box [37]	440d475	mp4	./MP4Box -diso @@
tiff2pdf [38]	020bd2f	tiff	./tiff2pdf @@
libming [39]	b72cc2f	swf	./swftophp @@
libxml2 [40]	bdec218	xml	./xmllint –valid –recover @@
cxxfilt [41]	a9d9a10	elf	./cxxfilt -t
objdump [41]	d7f734b	elf	./objdump -D @@

4.2. RQ1: How Does WolfFuzz Perform in Terms of Edge Coverage?

WolfFuzz is target-guided, but it attempts to maximize edge coverage during the initial exploration phase. It continues to activate new edges throughout the trigger targets phase when all individuals approach α, β, and δ individuals. To study this, the following experiment was devised, which used real-world programs as goals and involved 10 rounds of testing lasting 60 hours each. The edge coverage findings are shown in Table 6.

Table 6. Coverage results: the number represent how many edges were triggered by the seeds during the fuzzing process.

Fuzzer	Program					Average
	exiv2	MP4Box	tiff2pdf	libming	libxml2	
AFL	13,341.5	9146.9	17,951.7	12,434.5	7676.2	12,110.16
AFL++	18,194.4	10,991.3	17,265.9	15,656.7	8651.8	14,152.02
TortoiseFuzz	15,461.5	9748.4	16,819.6	15,431.2	8046.0	13,101.34
FishFuzz	18,895.9	11,466.8	17,986.9	14,962.5	7956.9	14,253.8
FairFuzz	17,164.1	10,546.8	15,193.8	14,976.3	7941.6	13,164.52
EcoFuzz	18,565.8	10,146.4	16,896.0	14,652.1	7766.9	13,605.44
KScheduler	18,146.6	11,463.5	17,416.6	15,091.4	8265.4	14,076.7
WolfFuzz	19,145.9	11,345.0	18,156.8	13,148.5	8041.8	13,967.6

Intuition tells us that the higher the code coverage, the more likely that the algorithm will find bugs. It can be seen that although the overall design of WolfFuzz is a target-guided fuzzer, its edge coverage score is still similar to or even better than those of well-known tools such as AFL, AFL++, and other fuzzers, which makes it possible for WolfFuzz to find bugs. Specifically, WolfFuzz outperforms AFL by 15% on average, which is due to the fact that WolfFuzz's efficient energy distribution strategy plays an important role in fuzzing. In libming and libxml2, tools such as AFL++ and TortoiseFuzz perform better in terms of coverage due to having strategies optimized for coverage.

4.3. RQ2: How Good Is WolfFuzz at Reaching Target Code Locations?

In the context of target-reaching evaluation, WolfFuzz was assessed alongside two prominent target-guided fuzzers, AFLGo and FishFuzz, as illustrated in Table 7.

Table 7. Target-reaching metric: the numbers represent how many seeds visited the function where the target selected in the static analysis stage is located during the fuzzing process.

Fuzzer	Program						Average
	exiv2	MP4Box	tiff2pdf	libming	cxxfilt	objdump	
AFLGo	5517.6	1922.6	2569.4	1251.6	1348.3	974.4	2263.98
FishFuzz	9645.1	2619.9	2604.1	1284.6	1722.7	1020.0	3149.40
WolfFuzz	10,456.3	2586.4	2853.8	1388.4	1726.8	1159.8	3361.92

The analysis demonstrates that WolfFuzz exhibits promising capabilities for reaching target code locations. In comparison to AFLGo, WolfFuzz performs notably better, showing an average improvement of 48.5%. It also surpasses FishFuzz by a margin of 6.7%. Notably, among various programs designed for target detection, including exiv2, tiff2pdf, libming, cxxfilt, and objdump, WolfFuzz stands out as a top performer. Additionally, WolfFuzz ranks second for the MP4Box program: still showcasing superior performance compared to AFLGo. In summary, WolfFuzz demonstrates superior efficacy in reaching target objectives.

4.4. RQ3: How Does WolfFuzz Perform in Vulnerability Recurrence Scenarios?

Vulnerability recurrence refers to the ability to generate corresponding vulnerability POCs by analyzing vulnerability release information on the Internet. It is an important application of DGF that demonstrates the capability of DGF in terms of target orientation. To better illustrate the vulnerability recurrence capability of WolfFuzz, Table 8 shows the time-to-exposure (TTE) of WolfFuzz and AFLGo for the same vulnerabilities. TTE is the time from the start of fuzzing to the appearance of the first erroneous input that acts on a specific target.

From the perspective of replicating the number of existing vulnerabilities within the specified time range (60 h), WolfFuzz takes less time than AFLGo to reproduce most of them, which undoubtedly proves WolfFuzz's vulnerability recurrence ability. AFLGo only performed better on four vulnerabilities (CVE-2017-11336, CVE-2017-17669, CVE-2017-5969, and CVE-2018-17985) due to the fact that WolfFuzz not only focused on reaching the target but also expanded the search area as much as possible throughout the process, resulting in some efficiency loss in the early stages. Overall, WolfFuzz was able to reproduce 76.4% (13 out of 17 bugs) of existing vulnerabilities faster. In terms of the time it takes to find a bug, WolfFuzz is on average 3.2 times faster than AFLGo, excluding vulnerabilities that AFLGo fails to find due to timeouts. Therefore, WolfFuzz has made great progress in both quantity detection and recurrence speed.

Table 8. The TTE results of AFLGo and WolfFuzz. T.O. indicates the fuzzer cannot reproduce the targets within the given time budget.

Program	CVE	Types	Description	AFLGo	WolfFuzz
exiv2	CVE-2017-11336	CWE-125	Out-of-bounds Read	0.22 h	0.32 h
exiv2	CVE-2017-17669	CWE-125	Out-of-bounds Read	0.24 h	0.48 h
exiv2	CVE-2018-10999	CWE-125	Out-of-bounds Read	58.94 h	23.53 h
exiv2	CVE-2018-17230	CWE-787	Out-of-bounds Write	T.O.	28.86 h
exiv2	CVE-2020-18898	CWE-674	Uncontrolled Recursion	40.89 h	13.42 h
MP4Box	CVE-2018-13005	CWE-125	Out-of-bounds Read	36.87 h	17.38 h
MP4Box	github_issue_1096	CWE-125	Out-of-bounds Read	T.O.	35.64 h
MP4Box	CVE-2019-20632	CWE-763	Release of Invalid Pointer or Reference	T.O.	44.83 h
MP4Box	github_issue_1446	CWE-125	Out-of-bounds Read	T.O.	55.61 h
tiff2pdf	CVE-2018-15209	CWE-787	Out-of-bounds Write	23.51 h	3.67 h
tiff2pdf	CVE-2018-16335	CWE-787	Out-of-bounds Write	25.48 h	2.82 h
libming	CVE-2018-13066	CWE-772	Missing Release of Resource after Effective Lifetime	50.43 h	15.67 h
libxml2	CVE-2017-5969	CWE-476	NULL Pointer Dereference	1.13 h	1.94 h
libxml2	CVE-2017-9047	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	T.O.	13.42 h
libxml2	CVE-2017-9049	CWE-125	Out-of-bounds Read	T.O.	20.37 h
cxxfilt	CVE-2019-9071	CWE-674	Uncontrolled Recursion	48.95 h	8.66 h
cxxfilt	CVE-2018-17985	CWE-400	Uncontrolled Resource Consumption	0.86 h	0.95 h

5. Conclusions and Future Work

In this paper, a directed greybox fuzzer based on dynamic distance weighting is proposed. Taking inspiration from wolf hunting, an adaptive search algorithm is proposed and utilized to construct the WolfFuzz framework and its system prototype. In WolfFuzz, the crash detection phase is the most crucial part of the fuzzing process as it determines whether the target can be captured. WolfFuzz alternates between the three phases by allocating time correctly. It employs a dynamic distance weighting strategy and IGWO to accelerate the generation of pertinent samples. Furthermore, WolfFuzz is validated against real-world targets. The experimental results show that WolfFuzz is more effective than other fuzzers at triggering crashes and detecting vulnerabilities.

Despite its capabilities, WolfFuzz still faces several limitations and challenges when it comes to real-world deployment. One such limitation is that the tool can only fuzz open-source targets. Furthermore, during the initial compilation of certain target programs, the static analysis module of WolfFuzz may encounter unforeseen errors, resulting in a selection of zero target points. Additionally, the current version of WolfFuzz lacks sufficient automation and necessitates manual configuration of specific hyperparameters before initiating fuzzing on target programs. In future development, it would be advantageous to consider integrating reverse modules into the static analysis stage to support closed-source binary programs. Moreover, implementing distinct phase switching conditions for various target programs could enhance WolfFuzz's efficiency and effectiveness in a more targeted manner.

Author Contributions: Conceptualization, D.X.; Data curation, Z.W. and Y.S.; Funding acquisition, D.X.; Methodology, Q.Z.; Project administration, Y.W.; Resources, K.Q.; Software, Q.Z.; Supervision, D.X.; Visualization, Z.W. and K.Q.; Writing—original draft, Q.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Science and Technology on Complex Electronic Systems Simulation Laboratory, grant number 614201002012204.

Data Availability Statement: The original contributions presented in the study are included in the article; further inquiries can be directed to the corresponding authors.

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

References

1. Wang, P.; Zhou, X.; Yue, T.; Lin, P.; Liu, Y.; Lu, K. The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing. *Softw. Testing, Verif. Reliab.* **2024**, *34*, e1869. [CrossRef]
2. 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–3 November 2017; pp. 2329–2344. [CrossRef]
3. Chen, H.; Xue, Y.; Li, Y.; Chen, B.; Xie, X.; Wu, X.; Liu, Y. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 2095–2108. [CrossRef]
4. Zong, P.; Lv, T.; Wang, D.; Deng, Z.; Liang, R.; Chen, K. {FuzzGuard}: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Online, 12–14 August 2020; pp. 2255–2269.
5. Zhu, X.; Liu, S.; Li, X.; Wen, S.; Zhang, J.; Seyit, C.; Xiang, Y. DeFuzz: Deep Learning Guided Directed Fuzzing. *arXiv* **2020**, arXiv:cs/2010.12149. [CrossRef].
6. Huang, H.; Guo, Y.; Shi, Q.; Yao, P.; Wu, R.; Zhang, C. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 36–50. [CrossRef]
7. Canakci, S.; Matyunin, N.; Graffi, K.; Joshi, A.; Egele, M. TargetFuzz: Using DARTs to Guide Directed Greybox Fuzzers. In Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May–3 June 2022; pp. 561–573. [CrossRef]
8. Zhu, K.; Lu, Y.; Huang, H.; Yu, L.; Zhao, J. Constructing More Complete Control Flow Graphs Utilizing Directed Gray-Box Fuzzing. *Appl. Sci.* **2021**, *11*, 1351. [CrossRef]
9. Wang, S.; Jiang, X.; Yu, X.; Sun, S. KCFuzz: Directed Fuzzing Based on Keypoint Coverage. In *Artificial Intelligence and Security*; Springer: Dublin, Ireland, 19–23 July 2021; pp. 312–325. [CrossRef]
10. Yao, R.; Zhang, Y.; Wang, S.; Qi, N.; Miridakis, N.; Tsiftsis, T. Deep Neural Network Assisted Approach for Antenna Selection in Untrusted Relay Networks. *IEEE Wirel. Commun. Lett.* **2019**, *8*, 1644–1647. [CrossRef]
11. Du, Z.; Li, Y.; Liu, Y.; Mao, B. WindRanger: A Directed Greybox Fuzzer Driven by Deviation Basic Blocks. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022; pp. 2440–2451. [CrossRef]
12. Huang, H.; Zhou, A.; Payer, M.; Zhang, C. Everything Is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference. In Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2024; p. 141.
13. Zheng, H.; Zhang, J.; Huang, Y.; Ren, Z.; Wang, H.; Cao, C.; Zhang, Y.; Toffalini, F.; Payer, M. {FISHFUZZ}: Catch Deeper Bugs by Throwing Larger Nets. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 1343–1360.
14. Liu, H.; Gan, S.; Zhang, C.; Gao, Z.; Zhang, H.; Wang, X.; Gao, G. LABRADOR: Response Guided Directed Fuzzing for Black-box IoT Devices. In Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2024; p. 126. [CrossRef]
15. Zhang, Y.; Liu, Y.; Xu, J.; Wang, Y. Predecessor-Aware Directed Greybox Fuzzing. In Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2024; p. 40. [CrossRef]
16. Fang, C.; Li, Y. SCDF: A Novel Single-Cell Classification Method Based on Dimension-Reduced Data Fusion. In *Proceedings of the Intelligent Computing Theories and Application*; Huang, D.S., Jo, K.H., Jing, J., Premaratne, P., Bevilacqua, V., Hussain, A., Eds.; Springer: Xi'an, China, 7–11 August 2022; pp. 196–206. [CrossRef]
17. Lin, P.; Wang, P.; Zhou, X.; Xie, W.; Zhang, G.; Lu, K. DeepGo: Predictive Directed Greybox Fuzzing. In Proceedings of the 2024 Network and Distributed System Security Symposium, San Diego, CA, USA, 26 February–1 March 2024. [CrossRef]
18. Jia, L.; Qi, N.; Chu, F.; Fang, S.; Wang, X.; Ma, S.; Feng, S. Game-Theoretic Learning Anti-Jamming Approaches in Wireless Networks. *IEEE Commun. Mag.* **2022**, *60*, 60–66. [CrossRef]
19. Liang, H.; Yu, X.; Cheng, X.; Liu, J.; Li, J. Multiple Targets Directed Greybox Fuzzing. *IEEE Trans. Dependable Secur. Comput.* **2023**, *21*, 325–339. [CrossRef]
20. Li, Y.; Chen, Y.; Ji, S.; Zhang, X.; Yan, G.; Liu, A.X.; Wu, C.; Pan, Z.; Lin, P. G-Fuzz: A Directed Fuzzing Framework for gVisor. *IEEE Trans. Dependable Secur. Comput.* **2024**, *21*, 168–185. [CrossRef]
21. Cao, S.; He, B.; Sun, X.; Ouyang, Y.; Zhang, C.; Wu, X.; Su, T.; Bo, L.; Li, B.; Ma, C.; et al. ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–24 May 2023; pp. 2726–2743. [CrossRef]

22. Luo, C.; Meng, W.; Li, P. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–24 May 2023; pp. 2693–2707. [CrossRef]
23. Yang, S.; He, Y.; Chen, K.; Ma, Z.; Luo, X.; Xie, Y.; Chen, J.; Zhang, C. 1dFuzz: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, 17–21 July 2023; pp. 867–879. [CrossRef]
24. Rong, H.; You, W.; Wang, X.; Mao, T. Toward Unbiased Multiple-Target Fuzzing with Path Diversity. *arXiv* **2023**, arXiv:2310.12419. [CrossRef].
25. Huang, H.; Yao, P.; Chiu, H.C.; Guo, Y.; Zhang, C. Titan: Efficient Multi-target Directed Greybox Fuzzing. In Proceedings of the 2024 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–22 May 2023; p. 58.
26. Kirkpatrick, S.; Gelatt, C.D.; Vecchi, M.P. Optimization by Simulated Annealing. *Science* **1983**, *220*, 671–680. [CrossRef] [PubMed]
27. Mirjalili, S.; Mirjalili, S.M.; Lewis, A. Grey Wolf Optimizer. *Adv. Eng. Softw.* **2014**, *69*, 46–61. [CrossRef]
28. American Fuzzy Lop—lcamtuf.coredump.cx. Available online: https://lcamtuf.coredump.cx/afl/ (accessed on 16 April 2024).
29. The AFL++ Fuzzing Framework | AFLplusplus—aflplus.plus. Available online: https://aflplus.plus/ (accessed on 16 April 2024).
30. Wang, Y.; Jia, X.; Liu, Y.; Zeng, K.; Bao, T.; Wu, D.; Su, P. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In Proceedings of the 2020 Network and Distributed System Security Symposium, San Diego, CA, USA, 23–26 February 2020. [CrossRef]
31. Österlund, S.; Razavi, K.; Bos, H.; Giuffrida, C. {ParmeSan}: Sanitizer-Guided Greybox Fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Online, 12–14 August 2020; pp. 2289–2306.
32. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-Based Greybox Fuzzing as Markov Chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, New York, NY, USA, 24–28 October 2016; pp. 1032–1043. [CrossRef]
33. Lemieux, C.; Sen, K. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 3–7 September 2018; pp. 475–485. [CrossRef]
34. 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 (USENIX Security 20), Online, 12–14 August 2020; pp. 2307–2324.
35. She, D.; Shah, A.; Jana, S. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 23–25 May 2022; pp. 2194–2211. [CrossRef]
36. GitHub—Exiv2/Exiv2: Image Metadata Library and Tools—github.com. Available online: https://github.com/Exiv2/exiv2/ (accessed on 16 April 2024).
37. GitHub—Gpac/Gpac: GPAC Ultramedia OSS for Video Streaming & Next-Gen Multimedia Transcoding, Packaging & Delivery—github.com. Available online: https://github.com/gpac/gpac (accessed on 16 April 2024).
38. GitHub—Libsdl-org/Libtiff: TIFF Decoding Library from github.com. Available online: https://github.com/libsdl-org/libtiff (accessed on 16 April 2024).
39. GitHub—Libming/Libming: SWF Output Library—github.com. Available online: https://github.com/libming/libming (accessed on 16 April 2024).
40. GitHub—GNOME/Libxml2: Read-Only Mirror of github.com. Available online: https://github.com/GNOME/libxml2 (accessed on 16 April 2024).
41. GitHub—Bminor/Binutils-gdb: Unofficial Mirror of Sourceware binutils-gdb Repository. Updated Daily.—github.com. Available online: https://github.com/bminor/binutils-gdb (accessed on 16 April 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.