2025 International Conference on Intelligent Computing
July 26-29, Ningbo, China
https://www.ic-icc.cn/2025/index.php

MSFuzz: Directed Greybox Fuzzing Using Multi-Target Sensitivity-Based Energy Scheduling

Chengwei Qin and Zhao Ma(✉)

School of Computer Science, China University of Geosciences, Wuhan 430078, China
`chwqin@cug.edu.cn`

Abstract. Directed Greybox Fuzzing (DGF) effectively targets specific program locations for bug discovery, but existing tools face challenges in multi-target directed fuzzing due to static stage division and coarse energy scheduling. Key challenges include global optimization biases that overlook lower-priority targets, inadequate prioritization of seeds that reach multiple targets, and inflexible exploration-exploitation stage allocation. This paper presents adaptive strategies to tackle these issues: a multi-target sensitivity-based energy scheduling approach that dynamically prioritizes seeds based on their target sensitivity; and a state-aware stage coordination strategy that balances exploration and exploitation using real-time fuzzing metrics to enable flexible stage transitions. We implemented these techniques in the tool MSFuzz, which optimizes resource allocation to avoid single-target bias and prevent inefficient stage durations. Evaluations on Magma, FuzzBench, and real-world programs show that MSFuzz outperforms state-of-the-art fuzzers like AFLGo, achieving 6.57× faster crash reproduction on Magma, 1.32× higher target-guided efficiency on FuzzBench. MSFuzz also discovered 27 unique crashes (13 CVEs) in real-world programs.

Keywords: Directed Greybox Fuzzing, Bug Discovery, Energy Scheduling.

1 Introduction

Fuzzing is a technique of great significance in the domains of software security and quality assurance. It aims to evaluate a target program's robustness against unexpected or malformed inputs by automatically generating random or semi-random test cases. Coverage-guided fuzzing (CGF) marked a major advancement in the field with the introduction of AFL[1], which leverages code coverage information to guide input mutations. However, since CGF primarily optimizes for global coverage, it often fails to effectively steer the fuzzing process toward high-risk areas in the code. In light of the limitations of CGF, directed greybox fuzzing (DGF) has gained increasing attention, especially following the release of AFLGo[2]. Unlike CGF, which aims for comprehensive code coverage, DGF introduces the concept of predefined targets and focuses testing resources on paths that are more likely to reach specific areas of interest, such as patched regions, critical functions, or potentially vulnerable code segments.

DGF tools have been widely adopted and are designed to support a variety of tasks, including deployment in CI/CD pipelines[4,5], reproducing crashes [6,6], and discovering vulnerabilities in binaries[8].

However, it is worth noting that directed fuzzers often perform poorly in multi-target directed fuzzing. One of the primary reasons is that existing DGF tools generally struggle with balancing global and local optimization during multi-target energy scheduling, and they exhibit low sensitivity to seeds that are capable of reaching multiple targets.

The energy scheduling strategy is designed to control the mutation frequency of seeds in fuzzing. In DGF, the core objective of energy scheduling is to allocate computational resources effectively, thereby improving the efficiency of reaching target regions and accelerating vulnerability triggering.

In multi-target directed fuzzing, the fuzzer should allocate more computational energy to seeds that exhibit higher efficiency in reaching multiple targets. By doing so, these seeds are more likely to trigger a greater number of unique execution paths and uncover more vulnerabilities after mutation.

DGF tools such as AFLGo, Hawkeye[6], and Windranger[7] employ static energy scheduling strategies when dealing with multiple targets. These approaches primarily rely on single-target guidance, and they often lack differentiated resource allocation mechanisms when facing large-scale target sets. As a result, the scheduling efficiency is generally suboptimal, leading to seed selection based on globally or locally suboptimal strategies.

For instance, AFLGo computes the cumulative basic block distance from each seed to all target locations. While this method helps improve overall convergence speed, it tends to suffer from a "winner-takes-all" problem in multi-target directed fuzzing—where a small number of high-priority targets monopolize the testing resources, leaving secondary targets systematically underexplored.

We summarize the following key challenges:

Problem 1: Global optimal strategies hinder multi-target directed greybox fuzzing. For example, during its directed fuzzing process, AFLGo gradually reduces the energy assigned to seeds with large distance values, resulting in an increasingly monotonous seed queue. This energy allocation becomes biased toward globally optimal seeds, causing seed execution to converge on a small number of high-priority target paths and hindering exploration of lower-priority paths.

Problem 2: Lack of fine-grained incentives for seeds that can reach multiple targets. Current energy scheduling strategies exhibit limited sensitivity to seeds that can reach multiple targets. Most existing strategies depend on single-target guidance functions, like reducing the distance to a particular target, which leads to inadequate identification and prioritization of seeds that can reach multiple targets. These methods do not dynamically evaluate a seed's potential impact on various targets, resulting in a failure to effectively encourage these seeds.

Problem 3: Imbalance in the coordination of exploration and exploitation. The fuzzing process in DGF can be categorized into two stages based on energy allocation strategies: the exploration stage, where seeds are altered to enhance code coverage, and the exploitation stage, which utilizes gathered knowledge to direct seeds toward the target and ultimately activate vulnerabilities. Currently, the time allocated for the exploration and exploitation stages in DGF is fixed. A prolonged exploration period might hinder the execution of new mutation seeds, while a quick exploration stage could lead to an insufficient corpus for the following exploitation stage.

To address these challenges, we propose two novel strategies: MSES (Multi-target Sensitivity-based Energy Scheduling) and SSCS (State-aware Stage Coordination Strategy). MSES is an adaptive energy allocation strategy specifically designed to tackle Problems 1 and 2. It dynamically adjusts energy distribution to enhance support for multi-target fuzzing. Unlike traditional strategies, MSES effectively identifies and prioritizes seeds that have the potential to reach multiple targets, allocating more energy to those with higher efficiency and reachability. This approach not only overcomes the limitations of global optimization heuristics but also enhances the effectiveness and scope of vulnerability discovery in multi-target environments. By accurately modeling the target sensitivity of the seed queue and flexibly tuning energy scheduling, MSES offers a more efficient and precise energy management scheme for DGF tools. SSCS, on the other hand, is a stage coordination strategy based on fuzzing state awareness, designed to address Problem 3. It introduces a state-aware switching mechanism that dynamically coordinates the exploration and exploitation stages according to the current fuzzing state. By adaptively adjusting stage transitions, SSCS enhances the overall fuzzing process.

The main contributions of this paper are as follows:

- We design an advanced energy allocation method that enhances DGF efficiency by prioritizing seeds with the potential to reach multiple targets.
- We develop a state-aware, two-stage coordination strategy that adaptively balances exploration and exploitation based on runtime metrics.
- We implement these techniques in MSFuzz and show, through extensive experiments on Magma, FuzzBench, and real-world programs, that MSFuzz significantly outperforms existing fuzzers in crash reproduction, target reaching, and vulnerability discovery.

The rest of this paper is structured as follows. The DGF-related background and our research motivations are presented in Section 2.The design details of MSFuzz are described in Section 3. The experimental evaluation of MSFuzz is provided in Section 4. Finally, We conclude the paper in Section 5.

2 Background

In this section, we first introduce the fundamental theories and techniques of DGF. We then analyze the limitations of existing energy scheduling strategies through a case study, and further elaborate on the key research problems addressed in this work.

2.1 Directed Greybox Fuzzing

DGF is an advanced fuzzing technique designed for goal-oriented testing. Its core idea is to introduce target guidance into the fuzzing process in order to prioritize the exploration of specific code regions or trigger particular behaviors. Unlike traditional Coverage-Guided Fuzzing (CGF), which aims to maximize global code coverage, DGF focuses on specific sensitive areas of a program, such as patch code[9], vulnerable functions[8,11], or exception handling logic[10]. As a result, DGF demonstrates higher precision and efficiency in tasks such as vulnerability reproduction, patch validation, and security assessment.

CGF employs a global exploration strategy that randomly mutates inputs to trigger as many program paths as possible. While this approach can comprehensively cover the execution flow of a program, its lack of guidance often results in low efficiency, spending significant time exploring areas unrelated to potential vulnerabilities.

In contrast, DGF guides test inputs toward predefined target regions, reducing exploration of non-critical paths and thus improving the efficiency of discovering high-risk vulnerability discovery. The selection of target regions can be based on various sources of information, including manually annotated code by security analysts, automatically identified sensitive functions or API calls through static analysis, runtime trace analysis to identify high-risk paths[12], vulnerability knowledge bases, or historical vulnerability data used to infer potentially dangerous code segments[13].To effectively approach these targets, DGF often incorporates lightweight static analysis techniques to construct call graphs and control flow graphs, which support core target-distance calculations.

DGF leverages a variety of optimization techniques to improve vulnerability discovery efficiency, including input optimization, distance metrics refinement, seed scheduling, energy scheduling strategies, and mutation optimization. These techniques work in synergy to enhance input effectiveness, improve target reachability, and optimize the use of testing resources. Among them, energy scheduling strategies play a crucial role by dynamically adjusting the allocation of computational resources based on feedback from fuzzing. This allows high-value test cases to receive more execution opportunities, improving overall fuzzing efficiency. For example, LOLLY[14] proposed a sequence-coverage-guided energy scheduling method that allocates energy based on a seed's ability to cover unique execution sequences. SLIME[15] introduced a program-feature-aware scheduling method that organizes seeds into multiple attribute-sensitive queues and applies a variance-aware Upper Confidence Bound (UCB-V) algorithm for seed selection.

DGF typically uses a two-stage scheduling strategy to balance path discovery and target-directed execution in energy scheduling. This strategy divides the fuzzing process into two stages: exploration and exploitation. In the exploration stage, the fuzzer prioritizes seeds that trigger new paths, aiming to accumulate comprehensive coverage information. This enriched coverage provides a solid foundation for subsequent exploitation. Once in the exploitation stage, the fuzzer allocates more energy to seeds that are already close to the target regions, enabling faster convergence towards the conditions required to trigger vulnerabilities.

2.2 Motivation

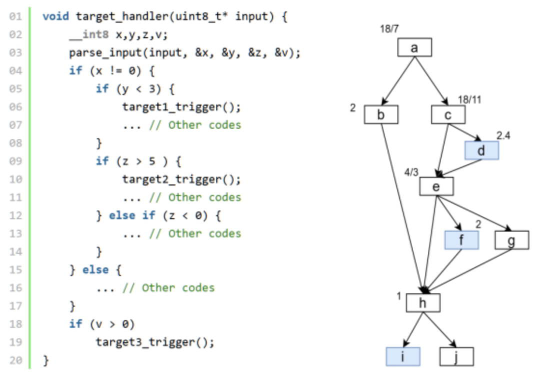
Energy scheduling strategies are designed to control the frequency of seed muta-tions during fuzzing. The primary goal of energy scheduling is to efficiently allocate computational resources in DGF, allowing for faster convergence to target regions and accelerating the triggering of vulnerabilities.

In multi-target directed fuzzing, the fuzzer should allocate more energy to seed files that are more efficient and capable of reaching multiple targets. These seeds, once mutated, are more likely to explore different execution paths and uncover additional vulnerabilities.

However, most existing DGF tools[2,6,10] employ static, globally optimal energy scheduling strategies when dealing with multiple targets. These approaches typically rely on single-target optimization and lack sensitivity to the differences between multiple targets. As a result, such strategies often lead to low scheduling efficiency when dealing with large sets of objectives.

AFLGo, for example, performs target-oriented optimization by computing the aggregated basic block distance from each seed to all targets. In multi-target directed fuzzing, this strategy tends to lead to over-concentration of resources, where a few high-priority targets tend to dominate fuzzing r over time, while others are systematically neglected.

Limitations of energy scheduling in multi-target reachability. Fig. 1 shows the sample code used for the case analysis in this section. Assume that there are three initial seeds during the fuzzing task: Seed s_1: $(x = 1, y = 0, z = 6, v = -1)$, Seed s_2: $(x = 2, y = 3, z = 1, v = 1)$, Seed s_3: $(x = 0, y = 7, z = 2, v = 1)$.



```
void target_handler(uint8_t* input) {
    __int8 x,y,z,v;
    parse_input(input, &x, &y, &z, &v);
    if (x != 0) {
        if (y < 3) {
            target1_trigger();
            ... // Other codes
        }
        if (z > 5 ) {
            target2_trigger();
            ... // Other codes
        } else if (z < 0) {
            ... // Other codes
        }
    } else {
        ... // Other codes
    }
    if (v > 0)
        target3_trigger();
}
```

Fig. 1. The sample code and its control flow graph (CFG) for the case analysis in multi-target directed greybox fuzzing.

The control flow graph (CFG) corresponding to this code snippet is illustrated in Fig. 1, where the blue nodes d, f, and i represent three target locations in the code. The execution path for seed s_1 is $a \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow h \rightarrow j$, seed s_1 is $a \rightarrow c \rightarrow e \rightarrow h \rightarrow i$, and for seed s_3 it is $a \rightarrow \text{b} \rightarrow h \rightarrow i$.

We compute the harmonic mean distance D from each node along these paths to the three targets, and annotate these values in Fig. 1. The computed average target distances for the three seeds are: $D_{s_1} = \frac{\frac{18}{7}+\frac{18}{11}+2.4+\frac{4}{3}+2+1}{6} \approx 1.82$, $D_{s_2} = \frac{\frac{18}{7}+\frac{18}{11}+\frac{4}{3}+1}{4} \approx 1.64$, $D_{s_3} = \frac{\frac{18}{7}+2+1}{3} \approx 1.86$.

AFLGo consistently selects the seed with the smallest global distance, in this case, the seed s_2. However, such a selection lacks target awareness, as it overlooks two target locations on the right-hand branch of the CFG. Among the other two seeds, seed s_1 is capable of reaching two targets, while seed s_3 reaches target i through a shorter execution path. These seeds should be considered valuable in multi-target directed fuzzing.

The shortest path seed is frequently prioritized over other seeds with significant target-reaching potential because of AFLGo's emphasis on global optimization. This example shows how uneven energy scheduling in multi-target directed fuzzing circumstances affects current techniques.

Furthermore, it can be observed that seed s_1 touches two target locations (nodes d and f) along its execution path, making it a high-quality seed in a multi-target context. It has the potential to reduce the number of mutations required to reach multiple targets. However, under AFLGo's energy scheduling strategy, seed s_2 is still given higher priority than seed s_1, without adequately considering seed s_1's multi-target reachability.

This highlights a key limitation in current approaches: the inability to effectively identify and reward multi-target-reachable seeds, reflecting a lack of sensitivity to multi-target testing requirements.

Inappropriate division between exploration and exploitation. The DGF is typically divided into two stages: exploration and exploitation. In the exploration stage, MSFuzz mutates and executes seed inputs to increase code coverage, thereby collecting more runtime information and generating new inputs that are more likely to reach the target locations. In the exploitation stage, MSFuzz uses the accumulated information to guide seeds closer to the targets, with the ultimate goal of triggering vulnerabilities.

However, most existing DGF approaches adopt a predefined time allocation strategy, statically dividing the total fuzzing time between exploration and exploitation. For example, AFLGo allocates 20 hours to exploration and 4 hours to exploitation. Such static scheduling can lead to two key problems:

Premature exploitation reduces target reaching: If the exploration stage is too short, MSFuzz may enter the exploitation stage without sufficient coverage information, making it difficult to generate high-quality, target-oriented seeds.

Excessive exploration wastes resources: If the exploration stage is too long, valuable resources may be spent on non-critical paths, delaying the transition to exploitation and weakening the overall focus of the fuzzing process.

3 Design of MSFuzz

In this section, we first present the overall architecture of MSFuzz, followed by a detailed description of its two core components.

3.1 Overview

By combining the MSES and SSCS components, MSFuzz enables energy scheduling that is sensitive to multiple targets. The framework of MSFuzz is shown in Fig. 2, and the fuzzing process consists of three stages: exploration, update, and exploitation.

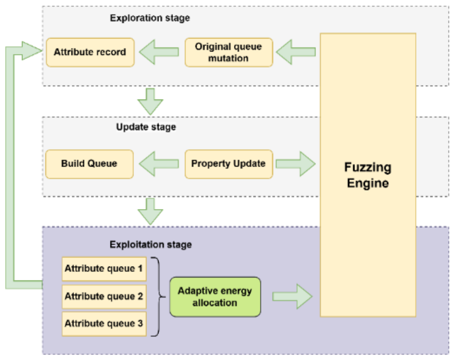


Fig. 2. The framework of MSFuzz.

Exploration Stage: In this stage, MSFuzz begins in the exploration stage during the initial stage of fuzzing. The primary goal in this stage is to generate new seeds and identify potential execution paths. Seeds from the original queue undergo mutation to produce new test cases. If a mutated input triggers a new path, it is saved as a new seed, added back to the queue, and its associated attributes are recorded. During exploration, the SSCS strategy monitors the proportion of seeds that fail to reach any target. When this metric drops below a threshold calculated by SSCS, MSFuzz transitions into the update stage.

Update Stage: Before entering the exploitation stage, MSFuzz re-evaluates each seed by updating its attribute values and rebuilding the attribute queues. During this process, MSFuzz iterates over all seeds, ranks them based on their performance in each attribute, and populates the corresponding attribute queues. If a queue has not reached its maximum size, or if the current seed outperforms the lowest ranked seed in the queue, it is added to the queue. If adding a new seed causes the queue to exceed the limit, the lowest performing seed is removed. This process reconstructs the queues based on the latest attribute values and provides the basis for energy scheduling.

Exploitation Stage: Once all attribute queues have been updated, MSFuzz enters the exploitation stage to perform energy scheduling. MSFuzz estimates the potential of each attribute queue to discover new targets or trigger crashes. Based on this estimation, it prioritizes queues with the highest potential and allocates energy to mutate the seeds within. When the generation rate of target-reaching seeds falls below the stage-switching threshold defined by SSCS, MSFuzz re-enters the exploration stage and begins a new iteration.

3.2 Multi-target Sensitivity-based Energy Scheduling

To enable multi-target sensitivity optimization, MSFuzz extends the SLIME[15] framework by introducing the MSES. Unlike the coverage-guided design of SLIME, MSES incorporates target-directed principles into fuzzing by defining multiple target-sensitive attributes. It constructs a set of attribute-specific seed queues and performs seed selection and prioritization based on these attributes. To further optimize energy distribution, MSES employs a variance-aware Upper Confidence Bound (UCB-V) algorithm inspired by the multi-armed bandit model to select the seed from the queue with the highest estimated reward.

The properties of MSES: To ensure seed sensitivity to multiple targets during the fuzzing process, MSFuzz performs fine-grained energy allocation that adapts to both global and local prioritization strategies. Based on existing seed evaluation metrics in fuzzers[2,6,8], MSES introduces a set of target-guided seed queues designed for multi-target scenarios. The following target-sensitive attributes are defined:

Definition 1. Global Average Distance (GAD). The GAD value of a seed s is computed using the seed distance formula defined in AFLGo. Seeds are sorted in descending order of $\mathrm{GAD}(s)$, and seeds with shorter GAD values are given higher scheduling priority.

Definition 2. Target Reach Count (TRC). The TRC value of a seed s represents the number of unique targets that can be triggered by the seed's mutations, defined as:

$$\mathrm{TRC}(s) = |\{\tau \in \mathcal{T} | \exists t \in T(s), \mathrm{Trigger}(t, \tau)\}| \tag{1}$$

where $T(s)$ is the set of test cases mutated from s, and $\mathcal{T}$ is the target set. A higher TRC value indicates greater potential for multi-target coverage.

Definition 3. High-Risk Target Count (HRTC). The HRTC value quantifies the number of high-risk basic blocks near targets that are covered by a seed's execution path. Using control-flow graph analysis, we extract basic blocks b whose static distance to a target τ satisfies $d_{\mathrm{static}}(b, \tau) \leq \theta_d$. Seeds are sorted in descending order of $\mathrm{HRTC}(s)$, prioritizing those that trigger more high-risk blocks:

$$\mathrm{HRTC}(s) = \textstyle\sum_{\tau \in \mathcal{T}} |\{b \in B_{\mathrm{risk}}^{\tau} \mid b \in \mathrm{Path}(s)\}| \tag{2}$$

Where $B_{\mathrm{risk}}^{\tau} = \{b \mid d_{\mathrm{static}}(b, \tau) \leq \theta_d\}$ $(\theta_d = 3)$, $Path(s)$ denotes the set of basic blocks covered by seed s.

Definition 4. Target Coverage Gain (TCG). TCG measures a seed's ability to introduce new target coverage through mutation, defined as:

$$\mathrm{TCG}(s) = \frac{|\mathrm{NewTargets}(s_{\mathrm{mut}})|}{|\mathrm{Mutations}(s)|} \tag{3}$$

where s_{mut} is a mutated test case generated from s, $\mathrm{NewTargets}(s_{\mathrm{mut}})$ denotes the set of newly covered targets, and $\mathrm{Mutations}(s)$ is the total number of mutations applied to s.

Definition 5. Target Relevance Entropy (TRE). TRE measures the distribution entropy of target-relevant basic blocks observed in historical execution paths of a seed, defined as:

$$P(b) = \frac{\text{Count}(b)}{\sum_{b' \in B_{\text{rel}}} \text{Count}(b')} \tag{4}$$

$$TRE(s) = -\sum_{b \in B_{\text{rel}}} P(b) \log_2 P(b) \tag{5}$$

where B_{rel} is the set of basic blocks with control dependencies to target locations, and $P(b)$ is the empirical probability of block b occurring in historical execution paths.

During the exploitation stage, MSFuzz select one of the attribute queues and mutate seeds from the selected queue. Since queues that are more likely to contain high-quality seeds should be prioritized, MSES models the queue selection process as a Multi-Armed Bandit (MAB) problem.

Under this formulation, selecting an attribute queue corresponds to pulling an arm of the bandit, where successfully discovering a new target or triggering a crash is considered a reward. MSES dynamically evaluates the priority of each queue by analyzing its historical reward potential and preferentially selects seeds from queues with higher estimated returns.

To solve this MAB problem, MSES employs the UCB-V algorithm to estimate the upper confidence bound for each queue, which is used as the primary decision criterion. As a queue is selected more frequently, the confidence in its ability to trigger valuable test cases increases, further influencing its selection probability.

Let $Q[i]$ denote the number of interesting test cases generated from queue i. Assume there are M attribute queues. For each selected queue i ($i \in M$), we define:

$$Q[i] = \sum_{s \in S} n_s \tag{6}$$

where S is the set of seeds in queue i, and n_s is the number of interesting test cases triggered by seed s. The total reward for queue i, denoted as $R[i]$, is defined as:

$$R[i] = \sum_{i=1}^{M} Q[i] \tag{7}$$

The squared sum of rewards for queue i, denoted as $R_{SQ}[i]$, is calculated as:

$$R_{SQ}[i] = \sum_{i=1}^{M} (Q[i] \times Q[i]) \tag{8}$$

Let $N[i]$ denote the number of times queue i has been selected, and N_{total} the total number of queue selections:

$$N_{total} = \sum_{i=1}^{M} N[i] \tag{9}$$

The estimated variance for queue i is then computed as:

$$Variance[i] = \frac{R_{SQ}[i]}{N[i]} - \frac{R[i] \times R[i]}{N[i] \times N[i]} \tag{10}$$

Finally, the Upper Confidence Bound with Variance (UCB-V) for queue i is calculated as:

$$UCB\_V[i] = \frac{R[i]}{N[i]} + \sqrt{\frac{2 \times Variance[i] \times \log(N_{total})}{N[i]}} + \frac{3 \times \log(N_{total})}{N[i]} \tag{11}$$

This UCB-V score is used by MSES to select the most promising attribute queue for mutation, ensuring that energy is allocated to high-potential seeds in a statistically balanced and adaptive manner.

3.3 State-aware Stage Coordination Strategy

To address the imbalance between exploration and exploitation stages, MSFuzz also implements a State-aware Stage Coordination Strategy (SSCS). SSCS dynamically coordinates the transitions between exploration and exploitation based on real-time fuzzing states, enabling MSFuzz to flexibly and adaptively manage the time allocation between the two stages.

We define the following stage dynamic state attributes to support the exploration and exploitation two-stage switching decision, as shown in Table 1.

Table 1. Definition of State-Aware Attributes

Symbol	Description
Stage	0 (Exploration), 1 (Exploitation)
nts	Number of seeds triggering unique paths through target blocks
sqe	Size of the seed queue in the exploration stage
R_{new}	Number of new paths discovered per time unit
ϕ	Mutation utility ratio of seeds
T_{util}	Duration of the exploitation stage
$T_{\max}$	Time threshold to force stage switching when no new paths are found
nos	Number of switch

SSCS constructs core state parameters by quantifying the distributional characteristics of the seed queue, supporting dynamic decisions between exploration and exploitation stages.

It defines the non-target seed ratio ρ to represent the effectiveness of exploration in reaching target blocks, computed as follows:

$$\rho = \frac{sqe - nts}{sqe} \tag{12}$$

Here, sqe denotes the total number of seeds in the current exploration stage, while nts is the number of seeds that uniquely pass through target blocks. A higher ρ value indicates that the generated seeds are less likely to reach targets, suggesting diminishing returns in further exploration.

To guide adaptive stage switching, we define a dynamic coordination coefficient λ_{exploit} that governs when MSFuzz transitions from exploration to exploitation. As λ_{exploit} is influenced by feedback from the previous exploitation stage, it is updated only at the end of an exploration cycle. A larger λ_{exploit} leads to prolonged exploration, whereas a smaller value favors early transition to exploitation for intensifying target-guided scheduling. The stage transition mechanism is illustrated in Fig. 3.

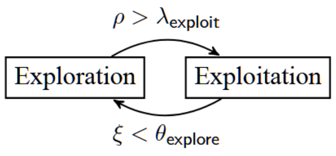

$\rho > \lambda_{\text{exploit}}$
Exploration
Exploitation
$\xi < \theta_{\text{explore}}$

Fig. 3. Two-stage state switching workflow between the exploration stage and the exploitation stage.

The transition condition from exploration to exploitation is defined as: $\rho > \lambda_{\text{exploit}}$, which signals saturation in path diversity, prompting MSFuzz to focus on target-directed mutation. The update of λ_{exploit} adopts a non-linear decay mechanism inspired by LeoFuzz[16], defined as:

$$\lambda^{*}_{\text{exploit}} = \lambda_{\text{exploit}} - \beta\left(\tanh\left(\frac{f(R_{\text{new}},\phi,T_{\text{util}})}{\sqrt{t}}\cdot\sqrt{nos}\right) - \delta\right) \tag{13}$$

Here, $\lambda^{*}_{\text{exploit}}$ is the threshold for the next cycle, $\tanh()$ ensures controlled updates, t is the duration of the current exploitation stage, and nos tracks the number of stage transitions. The function $f(R_{\text{new}},\phi,T_{\text{util}})$ captures feedback signals from the exploitation stage:

$$f(R_{\text{new}},\phi,T_{\text{util}}) = \omega_1\cdot R_{\text{new}} + \omega_2\cdot\phi_{\text{avg}} + \omega_3\cdot\frac{T_{\text{util}}}{T_{\text{util}}+c} \tag{14}$$

R_{new} is the number of new paths discovered per unit time, representing exploration effectiveness. ϕ_{avg} is the average mutation utility across seeds, reflecting mutation depth. T_{util} is the duration of the current exploitation stage, and c is a smoothing constant. This feedback-driven, non-linear weighting mechanism enables λ_{exploit} to adaptively regulate the switch from exploration when test gain becomes saturated.

To avoid local optima and prolonged stagnation, the fuzzer monitors the target seed generation rate ξ during exploitation. If ξ falls below a threshold θ_{explore}, the fuzzer switches back to the exploration stage. ξ is computed as:

$$\xi = \frac{\Delta nts}{T_{\text{util}}} \tag{15}$$

Here, Δnts denotes the number of newly discovered target-covering seeds, and T_{util} is the exploitation duration. A continuously declining ξ indicates saturation in target-space coverage, suggesting that the exploitation stage has exhausted its potential and should transition back to exploration. Conversely, if ξ remains high, the fuzzer continues exploiting the local target region.

The update rule for θ_{explore} is defined as:

$$\theta_{\text{explore}}^{*} = \theta_{\text{explore}} + \beta \cdot e^{-\Delta nts} \cdot \frac{T_{\text{util}}}{T_{\text{util}}+c} \quad (16)$$

Here, β is a decay factor controlling the update speed of θ_{explore}. The exponential term $e^{-\Delta nts}$ reflects recent target discovery activity—larger Δnts slows down the decay to preserve the current exploitation stage, while smaller values accelerate decay to encourage earlier transitions. The time-weighted factor $\frac{T_{\text{util}}}{T_{\text{util}}+c}$ further smooths updates and prevents overly aggressive switching when $\Delta\text{nts} = 0$.

To prevent MSFuzz from stalling during the exploitation stage, SSCS introduces a fallback switching mechanism. When the time elapsed since the last discovery of a new path exceeds a predefined threshold T_{max}, i.e., $T_{\text{new}} > T_{\text{max}}$, MSFuzz forcibly switches back to the exploration stage. This mechanism ensures that fuzzing continues to make progress even in the presence of local stagnation.

4 Evaluation

4.1 Experiment Setup

This section aims to answer the following three research questions:

- RQ1: How efficient is MSFuzz in reproducing crashes?
- RQ2: How good is the ability of MSFuzz in reaching target code locations?
- RQ3: Can MSFuzz find vulnerabilities in read-word programs?

Compared fuzzers. We compare MSFuzz to several widely used open source fuzzers, including AFL++[17], AFLGo, and Entropic[19]. First, we select AFL++, an actively maintained community-driven extension of AFL, which integrates a variety of advanced fuzzing techniques. Second, we include AFLGo as a reference for DGF, given its widespread use and established role as a benchmark in DGF research. Lastly, we select Entropic, a state-of-the-art fuzzer with advanced energy scheduling strategies, to highlight the performance differences among various energy allocation mechanisms.

Benchmarks. We evaluate MSFuzz on the Magma[18] and UniBench[20] benchmark suites. To evaluate the ability to reproduce crashes, we use the Magma dataset. Magma is a widely used fuzzing benchmark suite that contains a set of carefully selected real-world programs with complex input parsing and computation logic. UniBench is a comprehensive open-source benchmark designed specifically for fuzzing research. The test programs we selected from UniBench are shown in Table 2.

Table 2. The selected programs in UniBench.

Program	Version	Input format	Test instruction
exiv2	0.26	image	@@
lame	3.99.5	audio	@@ /dev/null
pdftotext	4.00	text	@@ /dev/null
tcpdump	4.8.1	network	-e -vv -nr @@

Real-world target programs. To evaluate the effectiveness of MSFuzz in detecting previously unknown vulnerabilities, we selected two real-world software projects as target programs: Bento4 and libming. We used the latest available branches-v1.6.0-641 of the Bento4 codebase and v0.4.8 of the libming codebase.

Configuration. To reduce randomness and ensure the reliability of experimental results, each experiment was conducted for 24 hours and repeated 10 times. All experiments were performed on a machine equipped with an Intel Xeon(R) Silver 4210 CPU, 128 GB of RAM, running Ubuntu 22.04.1.

Table 3. Comparison of crash reproducibility in Magma.

Bug ID	AFLGo		AFL++		Entropic		MSFuzz
	μTTE	Factor	μTTE	Factor	μTTE	Factor	
PNG003	15s	1.00	21s	1.40	19s	1.27	15s
PNG006	T.O.	N.A.	7m43s	0.20	22h26m	35.24	38m12s
PNG007	18h57m	2.17	3h19m	0.38	T.O.	N.A.	8h44m
TIF002	T.O.	N.A.	19h33m	0.88	22h29m	1.02	22h08m
TIF007	6m05s	1.87	4m42s	1.45	19m21s	5.95	3m15s
TIF009	14h13m	0.63	21h45m	0.96	18h16m	0.81	22h43m
TIF012	11h9m	11.84	1h26m	1.52	1h28m	1.56	56m31s
TIF014	19h04m	20.02	4h07m	4.32	3h42m	3.88	57m09s
PDF010	5h42m	1.66	2h35m	0.75	T.O.	N.A.	3h26m
PDF016	53m40s	13.76	T.O.	N.A.	2h13m	34.10	3m54s
PDF021	T.O.	N.A.	T.O.	N.A.	T.O.	N.A.	19h25m
PHP004	35m04s	15.14	T.O.	N.A.	T.O.	N.A.	2m19s
PHP009	3h53m	8.09	10h19m	21.49	T.O.	N.A.	28m48s
PHP011	52m12s	7.07	59m	7.99	1h57m	15.85	7m23s
SQL002	6h53m	1.38	5h17m	1.06	22h32m	4.52	4h59m
SQL014	21h47m	2.69	6h59m	0.86	8h02m	0.99	8h06m
SQL018	T.O.	N.A.	13h45m	4.85	T.O.	N.A.	2h50m

4.2 Efficiency in Reproducing Crashes (RQ1)

One of the core applications of directed fuzzing is the efficient reproduction of software crashes, which plays a critical role in vulnerability analysis and remediation. In real-world scenarios, due to data sensitivity or limited logging, crash reports submitted by users may lack the original input that triggered the vulnerability. In such cases, developers must rely solely on limited stack trace information. Directed fuzzing tools help address this challenge by guiding execution toward the suspected vulnerable region and generating test cases that reproduce the same crash, thereby assisting in debugging, patch validation, and root cause analysis.

Table 4. Comparison of target site reaching capability of different fuzzers.

Program	Target	AFLGo		AFL++		Entropic		MSFuzz
		μTTT	Factor	μTTT	Factor	μTTT	Factor	
exiv2	actions.cpp:194	5h24m	1.33	4h57m	1.22	4h38m	1.14	4h04m
	types.cpp:398	11h23m	1.07	8h21m	0.79	8h56m	0.84	10h39m
	basicio.cpp:1031	12h52m	0.92	13h46m	0.98	13h07m	0.93	14h02m
	image.cpp:492	1h38m	0.61	5h12m	1.99	3h15m	1.19	2h40m
	types.cpp:157	16h26m	0.89	18h53m	1.02	19h22m	1.05	18h31m
	basicio.cpp:1281	14h23m	1.12	13h32m	1.06	13h41m	1.07	12h48m
	tiffvisitor.cpp:1299	4h21m	1.43	12h45m	4.19	11h24m	3.76	3h02m
	image.cpp:700	1h46m	1.07	3h03m	1.85	3h54m	2.36	1h39m
	value.cpp:302	4h54m	0.94	6h24m	1.23	6h45m	1.29	5h13m
lame	util.c:608	9m39s	0.99	30m12s	3.09	10m27s	1.07	9m46s
	util.c:606	1m37s	0.55	27m18s	9.36	10m30s	3.60	2m55s
	vbrquantize.c:184	42m12s	1.85	56m44s	2.49	53m18s	2.34	22m47s
	get_audio.c:1452	9h51m	1.09	11h23m	1.26	11h16m	1.25	9h02m
	util.c:688	13m56s	0.65	35m42s	1.67	45m21s	2.13	21m21s
	mpglib_inter-face.c:332	6h38m	1.02	6h13m	0.96	5h59m	0.92	6h29m
	get_audio.c:1289	8h37m	0.93	11h24m	1.24	10h23m	1.12	9h14m
pdftotext	XRef.cc:1074	2h08m	0.77	4h37m	1.67	8h42m	3.14	2h46m
	Function.cc:193	4h24m	1.54	3h43m	1.30	2h40m	0.94	2h51m
	JPXStream.cc:329	4h36m	0.83	4h51m	0.87	6h09m	1.11	5h33m
	JPXStream.cc:1184	6h13m	1.34	9h48m	2.12	8h34m	1.85	4h38m
	AcroForm.cc:271	5h45m	1.63	4h26m	1.25	5h02m	1.42	3h32m
	GfxState.cc:3550	9h24m	1.21	5h57m	0.76	7h43m	0.99	7h48m
	JPXStream.cc:419	5h21m	1.02	8h09m	1.54	7h28m	1.41	5h17m
	XFAForm.cc:665	14h02m	0.69	20h44m	1.03	18h01m	0.90	20h12m
	Stream.cc:3051	19h26m	1.03	15h18m	0.81	18h14m	0.97	18h53m
	Stream.cc:3662	19h58m	0.97	20h01m	0.97	20h47m	1.01	20h35m
tcpdump	print-arp.c:210	4h47m	1.26	4h42m	1.24	4h55m	1.29	3h47m
	print-sl.c:65	9h21m	1.41	6h36m	0.99	7h23m	1.12	6h38m
	print-atm.c:405	14h15m	0.98	13h26m	0.93	13h34m	0.94	14h24m
	print-llc.c:233	14h13m	0.67	15h52m	0.75	19h53m	0.95	21h20m
	print-ppp.c:1662	5h57m	1.25	6h23m	1.34	6h11m	1.30	4h46m
	print-udp.c:160	10h08m	1.42	8h05m	1.13	8h38m	1.21	7h09m
	print-udp.c:367	13h33m	1.05	13h04m	1.01	13h29m	1.04	12h56m
	print-bootp.c:281	17h58m	0.94	19h26m	1.01	17h37m	0.92	19h10m

To evaluate the crash reproduction efficiency of MSFuzz, we conducted comparative experiments using the Magma dataset. We compared MSFuzz against three fuzzers: AFLGo, AFL++, and Entropic. Each tool was executed independently for 10 runs, with each run lasting 24 hours. We measured the average time-to-exposure (μTTE) required to reproduce a crash, and calculated the acceleration factor relative to MSFuzz.

The results summarized in Table 3 show that MSFuzz significantly outperforms all baselines in crash reproduction efficiency. On average, MSFuzz achieved a 6.72× improvement over AFLGo, a 9.56× improvement over Entropic, and a 3.44× improvement over AFL++.

4.3 Target Site Reaching Capability (RQ2)

To evaluate the target site reaching capability of MSFuzz, we conducted comparative experiments on the test programs listed in Table 2. To construct representative fuzzing targets, we queried the CVE vulnerability database to extract function names and corresponding line numbers for each known vulnerability, and used these locations as the target positions for directed fuzzing.

Table 5. List of vulnerabilities discovered by MSFuzz.

Program	Vulnerability Type	Vulnerable Function	CVE ID
Bento4	Memory Leaks	AP4_DescriptorFactory::CreateDescriptorFromStream	2025-25945
Bento4	Memory Leaks	SampleArray::SampleArray	2025-25942
Bento4	Memory Leaks	AP4_Processor::Process	2025-25946
Bento4	Buffer overflow	AP4_RtpAtom::AP4_RtpAtom	2025-25944
Bento4	Segmentation fault	AP4_AtomParent::RemoveChild	2025-25947
Libming	Memory Leaks	parseSWF_PLACEOBJECT3	2025-29486
Libming	Memory Leaks	parseSWF_INITACTION	2025-29488
Libming	Memory Exhaustion	parseABC_NS_SET_INFO	2025-29484
Libming	Memory Exhaustion	parseABC_STRING_INFO	2025-29487
Libming	Segmentation fault	decompileCALLMETHOD	2025-29490
Libming	Segmentation fault	decompileSETVARIABLE	2025-29492
Libming	Segmentation fault	decompileGETPROPERTY	2025-29493
Libming	Segmentation fault	decompileDUPLICATECLIP	2025-29496

We also selected AFL++, AFLGo, and Entropic as baseline tools to compare their performance in terms of target-guided efficiency. Each fuzzer was executed for 10 independent runs, with each run lasting 24 hours, to ensure the stability and reproducibility of the results. The experiments recorded the average time-to-target (μTTT)—i.e., the time required to reach the target line number in each run—and the acceleration factor relative to MSFuzz.

According to the results shown in Table 4, MSFuzz achieved an average improvement in target-guided efficiency of 1.07×, 1.43×, and 1.62× over AFLGo, Entropic, and AFL++, respectively. These results demonstrate that MSFuzz can reach vulnerability-related locations in significantly less time, validating the effectiveness of its target-guided mechanism in multi-target directed fuzzing.

4.4 Vulnerability Discovery Capability (RQ3)

To evaluate the effectiveness of MSFuzz in discovering previously unknown vulnerabilities, we selected two real-world software projects as target programs: Bento4 and libming. We used the latest branches of their respective codebases—v1.6.0-641 for Bento4 and v0.4.8 for libming—to ensure that the experimental results reflect MSFuzz's applicability and performance on up-to-date software versions.

During the experiments, all crash instances were deduplicated and thoroughly analyzed. MSFuzz successfully detected 27 unique crash cases, covering a variety of vulnerability types, including memory leaks, buffer overflows, heap overflows, and segmentation faults. Among them, 13 vulnerabilities were previously undisclosed, demonstrating MSFuzz's capability in discovering unknown bugs. As confirmed by the CVE assignment authority, 13 of these vulnerabilities have been officially assigned CVE identifiers (see Table 5).

5 Conclusion

We present MSFuzz, a directed greybox fuzzing framework that integrates two novel approaches: a multi-target sensitivity-based energy scheduling strategy and a state-aware stage coordination strategy. Empirical results demonstrate that in crash reproduction experiments, MSFuzz's average performance is 6.57× faster than baseline counterparts; in multi-target directed fuzzing experiments, it achieves an average performance improvement of 1.32× over baseline tools. Additionally, MSFuzz successfully detected 27 unique crashes in real-world programs, 13 of which have been assigned CVE identifiers, proving its practical effectiveness in discovering previously unknown vulnerabilities.

References

1. 2021. American Fuzzy Lop. https://github.com/google/AFL.
2. Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing[C]//Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. 2017: 2329-2344.
3. Huang H, Guo Y, Shi Q, et al. Beacon: Directed grey-box fuzzing with provable path pruning[C]//2022 IEEE Symposium on Security and Privacy (SP). IEEE, 2022: 36-50.
4. Sharma A, Cadar C, Metzman J. Effective Fuzzing within CI/CD Pipelines (Registered Report)[C]//Proceedings of the 3rd ACM International Fuzzing Workshop. 2024: 52-60.

5. Huang M, Lemieux C. Directed or Undirected: Investigating Fuzzing Strategies in a CI/CD Setup (Registered Report)[C]//Proceedings of the 3rd ACM International Fuzzing Workshop. 2024: 33-41.
6. Chen H, Xue Y, Li Y, et al. Hawkeye: Towards a desired directed grey-box fuzzer[C]//Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. 2018: 2095-2108.
7. Du Z, Li Y, Liu Y, et al. Windranger: A directed greybox fuzzer driven by deviation basic blocks[C]//Proceedings of the 44th International Conference on Software Engineering. 2022: 2440-2451.
8. Zheng H, Zhang J, Huang Y, et al. FISHFUZZ: Catch deeper bugs by throwing larger nets[C]//32nd USENIX Security Symposium (USENIX Security 23). 2023: 1343-1360.
9. Xiang Y, Zhang X, Liu P, et al. Critical code guided directed greybox fuzzing for commits[C]//33rd USENIX Security Symposium (USENIX Security 24). 2024: 2459-2474.
10. Österlund S, Razavi K, Bos H, et al. ParmeSan: Sanitizer-guided greybox fuzzing[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 2289-2306.
11. Zhu X, Liu S, Li X, et al. Defuzz: Deep learning guided directed fuzzing[J]. arXiv preprint arXiv:2010.12149, 2020.
12. Liang H, Jiang L, Ai L, et al. Sequence directed hybrid fuzzing[C]//2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020: 127-137.
13. Li S, Li Y, Chen Z, et al. TransferFuzz: Fuzzing with Historical Trace for Verifying Propagated Vulnerability Code[J]. arXiv preprint arXiv:2411.18347, 2024.
14. Liang H, Zhang Y, Yu Y, et al. Sequence coverage directed greybox fuzzing[C]//2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE Computer Society, 2019: 249-259.
15. Lyu C, Liang H, Ji S, et al. SLIME: program-sensitive energy allocation for fuzzing[C]//Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis. 2022: 365-377.
16. Liang H, Yu X, Cheng X, et al. Multiple targets directed greybox fuzzing[J]. IEEE Transactions on Dependable and Secure Computing, 2023, 21(1): 325-339.
17. 2025. AFLplusplus. https://github.com/AFLplusplus/AFLplusplus.
18. Hazimeh A, Herrera A, Payer M. Magma: A ground-truth fuzzing benchmark[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2020, 4(3): 1-29.
19. Böhme M, Manès V J M, Cha S K. Boosting fuzzer efficiency: An information theoretic perspective[C]//Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020: 678-689.
20. Li Y, Ji S, Chen Y, et al. UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers[C]//30th USENIX Security Symposium (USENIX Security 21). 2021: 2777-2794.