SyzDiversity: Diversity-Guided Linux Kernel Fuzzing

KUN HU, Fudan University, China
JIAJI QIN, Fudan University, China
CHAOFENG SHA, Fudan University, China
BIHUAN CHEN*, Fudan University, China
SHUORAN BAI, Harbin Engineering University, China
QICAI CHEN, Fudan University, China
CHENGLIN WANG, Fudan University, China
XIN PENG, Fudan University, China
WENYUN ZHAO, Fudan University, China

While coverage-guided kernel fuzzers have been proposed to uncover Linux kernel vulnerabilities, their code coverage and bug-finding capability are limited due to the lack of seed diversity, which is caused by the compounding effect of initial seed generation, seed scheduling, and seed mutation. To address this limitation, we propose a diversity-guided kernel fuzzer SyzDiversity. Specifically, to mitigate overvaluation of early seeds, it leverages proof-of-concept (PoC) seeds derived from real-world vulnerabilities as initial seeds, and further partitions these seeds into multiple communities. To improve diversity guidance in seed scheduling, it leverages a novel metric, community popularity rate (CPR), to model community diversity, and introduces a CPR-aware hierarchical Multi-Armed Bandit (MAB) algorithm that integrates CPR and code coverage as reward signals to prioritize the scheduling of diverse seed communities and seeds. Further, to efficiently populate sparse communities or break through community boundaries, it adopts a CPR-guided seed mutation strategy that adaptively allocates higher mutation frequencies to communities that are more conducive to the diversity evolution of the seeds. Our extensive experiments on Linux kernel versions v5.15 and v6.14 have demonstrated that SyzDiversity improves code coverage and bug-finding capability by 17.2% and 6.4×, respectively, compared to the state-of-the-art kernel fuzzers. It has discovered 32 unique new vulnerabilities, with 12 of them confirmed.

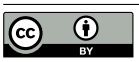
CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: kernel fuzzing, seed diversity, Linux kernel, vulnerability detection

ACM Reference Format:
Kun Hu, Jiaji Qin, Chaofeng Sha, Bihuan Chen, Shuoran Bai, Qicai Chen, Chenglin Wang, Xin Peng, and Wenyun Zhao. 2026. SyzDiversity: Diversity-Guided Linux Kernel Fuzzing. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA020 (October 2026), 23 pages. https://doi.org/10.1145/3832111

*Bihuan Chen is the corresponding author.

Authors' Contact Information: Kun Hu, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, huk23@m.fudan.edu.cn; Jiaji Qin, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, jjtan24@m.fudan.edu.cn; Chaofeng Sha, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, cfsha@fudan.edu.cn; Bihuan Chen, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, bhchen@fudan.edu.cn; Shuoran Bai, Harbin Engineering University, Harbin, China, baishuoran@hrbeu.edu.cn; Qicai Chen, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, qcchen23@m.fudan.edu.cn; Chenglin Wang, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, chenglinwang23@m.fudan.edu.cn; Xin Peng, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, pengxin@fudan.edu.cn; Wenyun Zhao, College of Computer Science and Artificial Intelligence, Fudan University, Shanghai, China, wyzhao@fudan.edu.cn.



ACM 2994-970X/2026/10-ARTISSTA020
https://doi.org/10.1145/3832111

1 Introduction

The Linux kernel underpins a wide spectrum of critical systems ranging from cloud servers [33] to industrial control systems [9] and autonomous driving systems [7]. Serving as the cornerstone for the entire system, any vulnerability in the kernel can not only lead to kernel crashes or privilege escalation attacks, but also cascade to affect all dependent applications, potentially causing irreversible damage [24]. For instance, resource management vulnerabilities in the kernel may lead to performance degradation or unavailability of user applications [38]; memory out-of-bounds access or privilege bypass vulnerabilities can be exploited to leak sensitive data from servers [43].

Kernel fuzzing has been regarded as an effective and successful testing technique, and attracted considerable research attention in recent years [20, 58]. To date, the most widely used tool is Syzkaller [17], a coverage-guided kernel fuzzer, introduced by Google in 2015. It leverages Syzlang [16], a domain-specific language, to declaratively model kernel syscall interfaces. This enables the automated generation, scheduling, and mutation of syscall sequences (also known as seeds). Through this mechanism, Syzkaller has become a core tool for discovering kernel vulnerabilities. Its automated vulnerability reporting system, Syzbot [14], has reported over 23,830 Linux kernel bugs to the community, among which 26.3% (6,271 bugs) of them have been identified as high-severity. Since then, various fuzzers [8, 10, 19, 27, 40, 50, 51, 55, 59, 62, 65, 66] have been proposed.

Problems. While these fuzzers have achieved significant success, their code coverage and bug-finding capability remain limited. A key bottleneck lies in the lack of seed diversity, affected by initial seed generation, scheduling, and mutation strategies, which results in redundant exploration of similar execution paths, limits the theoretical upper bound of code coverage, and ultimately reduces the chance of discovering vulnerabilities. This diversity bottleneck can be attributed to three problems.

P1. Overvaluation of Randomly Generated Seeds at Early Stage. Triggering kernel vulnerabilities often requires constructing complex contexts and preconditions. For example, Fig. 1 (a) shows a seed which triggers a general protection fault in the net subsystem. However, at the early stage of fuzzing, since the exploration space is far from saturated, any syntactically valid seed can lead to an increase in coverage (even without the complex contexts required to trigger vulnerabilities), causing the fuzzer to overestimate the seed's value and subject it to excessive mutation. This phenomenon is illustrated in Fig. 1 (b-1) and (b-2), where seed ① is an initial seed randomly generated based on Syzlang [16]. Although such a seed can produce syntactically valid programs (*e.g.,* the return value of `socket` is correctly used later) and calls a key syscall `sendmsg$nl_route_sched`, it still lacks the complex context and precondition required to actually trigger a vulnerability. Due to this overvaluation, seed ① is frequently mutated, leading to the repeated generation of locally similar seeds (*e.g.,* seeds ②-③) [17]. This overvaluation biases downstream strategies such as scheduling and mutation toward locally similar regions, thereby trapping the search and making it increasingly difficult to reach dissimilar seeds (e.g., the seed in Fig. 1 (a)) that require complex contexts and preconditions.

Some fuzzers [40, 51] attempt to mitigate this overvaluation problem by using syscall sequences collected from real-world programs or from Syzkaller after a period of execution as initial seeds. While these methods indeed provide more contextual information than randomly generated seeds (*e.g.,* seed ①), they are still essentially derived from seeds that have not triggered vulnerabilities, and hence the risk of overvaluation remains. Other fuzzers [8, 10, 19, 50, 62] automatically generate Syzlang specifications to supplement syscall interfaces missed during rapid kernel evolution, thereby enhancing the effectiveness of initial seeds. However, these methods still rely on generating seeds from scratch based on Syzlang, and thus cannot effectively address the problem of overvaluation.

P2. Limited Guidance in Leveraging Diverse Seeds in Scheduling Strategies. Existing scheduling strategies are largely driven by a single coverage-based heuristic and lack the guidance from seed diversity, and the problem **P1** further causes these strategies to favor exploration along frequently

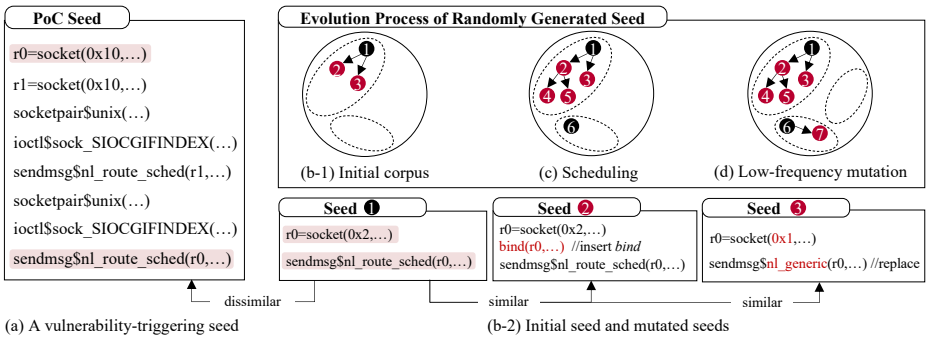


Fig. 1. Illustration of the lack of seed diversity (the red numbered circles denote the seeds after mutation)

executed paths. For example, Syzkaller-based fuzzers [27, 40, 51, 59, 66] typically select seeds for mutation using coverage-based probabilistic scheduling, while SyzVegas [55] prioritizes seeds that can achieve high coverage in a shorter period of time. Due to the problem **P1**, a high density of similar seeds are generated at the early stage, making these fuzzers more likely to redundantly schedule locally similar seeds. Although SyzGPT [66] periodically generates seeds targeting low-frequency syscalls (*e.g.,* seed ⑥ in Fig. 1(c)), these low-frequency seeds have a low probability of being selected for mutation, as the scheduling strategies do not provide a special favor for them. For example, seed ② is still likely to be selected to obtain seeds ④ and ⑤, as such simple seeds can yield coverage gains at the early stage of fuzzing, which limits the effective exploration of diverse seeds (*e.g.,* seed ⑥). Even when **P1** is partially mitigated, the lack of explicit diversity guidance in scheduling still drives the search toward frequently executed paths. For instance, although SUNFLOWER [65] leverages manually curated PoC seeds to alleviate the early-stage overvaluation, its subsequent scheduling provides no explicit diversity signal, leaving continuous seed diversity exploitation unexplored.

P3. Low Mutation Efficiency in Generating Semantically Diverse Seeds. In general, the theoretical upper bound of coverage is determined by the diversity of the explored seed space, namely the total number of semantically distinct seed clusters. Therefore, reaching or approaching this upper bound depends on whether the fuzzer can 1) discover new clusters (*e.g.,* the new cluster in Fig. 1 (d)), and 2) efficiently mutate seeds to populate sparse clusters (e.g., seed ⑦ in Fig. 1 (d)). Unfortunately, existing fuzzers [51, 59] fail to prioritize sparse clusters as mutation targets (owing to **P1** and **P2**), resulting in a low probability of discovering new clusters. In addition, they often adopt conservative mutation strategies for seeds in the corpus, typically generating only one mutated seed per cycle. If this new seed fails to increase coverage, the parent seed, even if it originates from a sparse cluster, is prematurely discarded, forfeiting the opportunity to further explore that cluster. As a result, the fuzzer becomes trapped in saturated local regions, repeatedly mutating seeds from high-density clusters, limiting both code coverage and bug-finding capability.

Approach. To address the above problems, we propose SyzDiversity, a kernel fuzzer guided by a novel diversity metric, community popularity rate (CPR), to steer and enhance the generation of diverse seeds. Specifically, to address problem **P1**, we leverage syscall sequences in 22,506 high-value proof-of-concepts (PoCs) collected as initial seeds to mitigate the early-stage overvaluation of randomly generated seeds. We further apply a community partitioning algorithm to group these seeds into multiple initial communities (*i.e.,* seed clusters), serving as the foundation for CPR computation. To address problem **P2**, we design a CPR-aware hierarchical Multi-Armed Bandit (MAB) algorithm, which leverages both CPR and code coverage as reward signals to prioritize the scheduling of diverse seed communities and seeds. To address problem **P3**, we propose a CPR-guided mutation strategy that adaptively allocates higher mutation frequencies to communities that are more conducive to the diversity evolution of the corpus. This strategy enhances the likelihood of breaking through

community boundaries and generating entirely new communities, while efficiently mutating seeds in sparse communities to rapidly populate these previously unexplored regions of the seed space.

Evaluation. We evaluate SyzDiversity on Linux kernels v5.15 and v6.14 by comparing it with several recent state-of-the-art kernel fuzzers, *i.e.,* Syzkaller [17], HEALER [51], SyzVegas [55], and SyzGPT [66]. SyzDiversity demonstrates substantial improvements in code coverage, achieving an average increase of 17.2% and a maximum increase of 41.9%, as well as in bug-finding capability, achieving a 6.4× average and 26× maximum improvement. It has successfully discovered 32 unique new vulnerabilities during a two-month fuzzing campaign, with 12 of them confirmed, 8 fixed, and 3 assigned CVE identifiers. Finally, we conduct an ablation study and parameter sensitivity analysis to assess the effectiveness of each module and to justify the choice of optimal hyperparameters.

Contributions. This work makes the following contributions.

- We propose and implement a diversity-guided Linux kernel fuzzer SyzDiversity based on a novel diversity metric, community popularity rate, aiming at improving the seed diversity.
- We conduct a comprehensive evaluation of SyzDiversity on the widely used Linux kernel v5.15 and the latest v6.14 by comparing it with state-of-the-art kernel fuzzers, which demonstrate its significant effectiveness improvements over existing fuzzers.
- We successfully detect 32 unique new vulnerabilities in the Linux kernel during a two-month fuzzing campaign, 12 of which have been confirmed by Linux kernel developers.

2 Background

2.1 Syscall-Based Kernel Fuzzing

Kernel fuzzers that treat syscall sequences as input seeds are typically composed of three core modules, *i.e.,* seed generation, scheduling, and mutation [20, 58]. Seed generation is responsible for providing the fuzzer with syntactically valid seeds. Seed scheduling assigns higher execution and mutation priorities to "high-value" seeds [52, 55, 60]. Seed mutation continuously transforms or perturbs the scheduled seeds through structural or fine-grained mutations. When a mutated seed leads to new coverage, it is preserved in the corpus as a foundation for subsequent fuzzing.

Seed generation relies on a manually crafted DSL, *i.e.,* Syzlang [16], which defines syscall templates, and a program syntax [15] that specifies how syscall sequences are organized. It is triggered (1) during the initial fuzzing stage when no seed is available, it generates seeds from scratch; and (2) during ongoing fuzzing, it injects new seeds at a fixed probability to prevent fuzzing stagnation.

The seed scheduling module randomly or heuristically selects seeds from the corpus for mutation and execution, determining which seeds are prioritized for subsequent evolution (*e.g.,* those with high coverage or high diversity), thereby serving as a key lever for maximizing fuzzing metrics.

The seed mutation module is divided into two strategies: regular mutation and smash mutation. Regular mutation operates on selected seeds, and applies one or more mutators, chosen based on empirical probabilities. These mutators perform syscall-level changes such as insertion, deletion, modification, or concatenation, as well as argument-level transformations including bit flipping, byte insertion/deletion, integer arithmetic, setting magic values, and endianness swapping. Each regular mutation generates only a single new seed. In contrast, smash mutation is a more aggressive exploitation strategy. When a seed generated by a regular mutation leads to new coverage, smash mutation is immediately triggered to perform multiple additional regular mutations on that seed, thereby amplifying the potential coverage gain (*e.g.,* up to 25 additional mutations in Syzkaller by default).

2.2 Upper Confidence Bound Algorithm

Seed scheduling involves a continuous trade-off in decision making and can be naturally modeled as a Multi-Armed Bandit (MAB) problem. Upper Confidence Bound (UCB) [1] is one of the

classical solutions to this problem. The MAB framework models the fundamental trade-off between *exploration* and *exploitation*, where each arm corresponds to a candidate action with an unknown reward distribution. The objective is to select arms over a finite time horizon to maximize the cumulative reward. Selecting the arm with the highest estimated value corresponds to *exploitation*, whereas selecting under-explored arms represents *exploration*. In the context of fuzzing, this trade-off manifests as follows: further mutating high-coverage seeds corresponds to *exploitation*, while prioritizing low-coverage or under-explored seeds reflects an *exploration* strategy.

Apart from UCB, the MAB problem has several classical solutions, e.g., ε-*Greedy* [57], Thompson Sampling [45], and EXP3 [2]. Compared to them, UCB requires no prior knowledge of the reward distribution, has a simple algorithmic structure with low computational overhead, and enjoys a logarithmic theoretical regret bound, balancing practicality and convergence. In the kernel fuzzing characterized by sparse feedback, high execution cost, and the need for fast decision-making, UCB offers notable practical advantages. The UCB family includes various algorithmic variants, among which UCB-1 is widely adopted for best-arm identification due to its structural simplicity and logarithmic regret convergence rate $O(logT)$, where T denotes the number of rounds [1].

$$UCB_i(t) = \underbrace{\bar{x}_i(t)}_{exploitation} + \underbrace{\sqrt{\frac{2ln(t)}{n_i(t)}}}_{exploration} \tag{1}$$

Eq. 1 presents the basic form of the UCB-1 algorithm, where $n_i(t)$ denotes the number of times arm i is selected before round t, $\bar{x}_i(t)$ denotes the average reward of arm i after being selected $n_i(t)$ times before round t, and $\sqrt{\frac{2ln(t)}{n_i(t)}}$ is the exploration bonus (larger for less frequently explored arms). The use of $ln(t)$ is standard in UCB-1 (allowing for a more concise mathematical derivation [1]), and it ensures that even arms with relatively low historical rewards will eventually receive more attention if they have not been selected for a long time. The denominator $n_i(t)$ also penalizes frequently selected arms, thus boosting the UCB scores of under-explored ones. This UCB-based strategy is naturally well-suited for seed scheduling in fuzzing, where the average reward $\bar{x}_i(t)$ must be carefully designed based on domain knowledge and specific fuzzing objectives.

3 Related Work

3.1 Generic Kernel Fuzzing

Syzkaller-Based Fuzzers. Benefiting from the expert knowledge and human efforts invested in Syzlang [16], various fuzzers [8, 10, 19, 27, 40, 50, 51, 55, 59, 62, 65, 66] have been proposed to enhance the performance of Syzkaller from the perspectives of seed generation, scheduling, and mutation. For seed generation, Moonshine [40] and Healer [51] improve the seed quality by extracting syscall sequences from real-world programs or from Syzkaller after a period of execution. Meanwhile, SUNFLOWER [65] focuses on constructing high-value initial seeds via a semi-automated workflow that combines manual analysis with automated syscall sequence extraction. However, the heavy reliance on manual curation makes it labor-intensive and hard to generalize to new syscalls. SyzGPT [66] periodically generates low-frequency syscall sequences during fuzzing. Other fuzzers [8, 10, 19, 20, 62] enhance the seed quality by generating more accurate syscall specifications. For seed scheduling, SyzVegas [55] improves the execution efficiency and coverage of Syzkaller by prioritizing seeds that can achieve high coverage in a shorter period of time. For seed mutation, Healer [51] and MOCK [59] capture dependencies between syscalls, while HFL [27] identifies critical syscalls and their arguments via symbolic execution and prioritizes their mutation. These approaches do not explicitly

consider seed diversity in seed generation, scheduling, or mutation, nor do they explore the coordinated optimization of these three modules to generate diverse seeds. As a result, the local optima issues arising in individual modules may be propagated and exacerbated across the entire workflow.

Non-Syzkaller-Based Fuzzers. Trinity [12] is an early kernel fuzzer that performs black-box fuzzing. In addition, existing non-Syzkaller-based kernel fuzzers are specifically designed based on libFuzzer [41] or AFL [29]. For example, FUZZNG [6] uses libFuzzer to interpret generated bytecode as syscalls and arguments, while TriforceAFL [18], X-AFL [31], and Unicorefuzz [35] implement bytecode generation and mutation for encoded syscalls and arguments on top of AFL. They aim to reduce reliance on syscall specifications. However, their seed space exploration remains limited.

3.2 Specific Kernel Fuzzing

Some fuzzers are often specifically designed for certain fuzzing targets. For example, some fuzzers are focused on specific vulnerability types, e.g., double-fetch bugs [46], memory bugs [13], and data race vulnerabilities [22, 23, 25, 60]. Some fuzzers target specific operating systems, e.g., complex enterprise-level kernel environments [49], and real-time operating systems [47, 48]. Some fuzzers are designed for specific subsystems, e.g., drivers [67], file systems [28, 32, 34, 61], eBPF [21], and specific lines of code [52]. These approaches differ from our work in the fuzzing objective and scope.

3.3 Diversity-Aware Seed Selection

Existing diversity-aware seed selection approaches [4, 5, 37, 42] mainly target user-space application fuzzing. These approaches typically guide seed selection either by fine-grained path/branch statistics or by structured input-space partitioning, aiming to enrich the corpus with broader behavioral diversity. However, directly applying these approaches to kernel fuzzing has several limitations. For fine-grained path/branch statistics-based approaches, their instability can be exacerbated for state-dependent syscall executions, whose paths are jointly sensitive to the kernel state and the rich syscall parameter space. Such instability may mistakenly treat seeds within the same subsystem as diverse, thus amplifying local path variations rather than global seed space diversity. For structured input-space partitioning-based approaches, their applicability is limited for large-scale and complex kernel executions, whose input space lacks a clear and embeddable partitioning basis. Such limitation may force semantically distinct seeds into the same partition, thus obscuring local partition boundaries.

3.4 MAB-Based Fuzzing

MAB-based fuzzers formulate key fuzzing tasks as a canonical *exploration vs. exploitation* problem. The key is to design appropriate reward functions that estimate the expected benefit of each strategy. For *non-kernel fuzzing*, Woo et al. [56] use the number of crashes as the reward for guiding seed scheduling in black-box fuzzing. EcoFuzz [63] adopts a variant of adversarial MAB (VAMAB) to alleviate over-allocation to frequently executed paths in AFL. SEAMFUZZ [30] dynamically tunes AFL's mutator selection. For *kernel fuzzing*, SyzVegas [55] uses the ratio of coverage growth to execution time as the reward to schedule both fuzzing tasks and seeds in Syzkaller, mitigating the risk of generating a large number of low-value seeds caused by fixed mutation strategies. MOCK [59] leverages the UCB algorithm to adaptively switch between static dependency-based and language model-based mutation strategies. These approaches often prioritize immediate effectiveness while neglecting the exploration of global diversity. As a result, the corpus tends to concentrate on code regions initially covered by early seeds, limiting the coverage upper bound.

4 Empirical Study

We first conduct an empirical study to examine whether continuous seed-diversity exploitation remains indispensable for fuzzing performance, even with high-value initial seeds provided.

Table 1. Comparison of different initial seed settings (↑/↓ are relative to *All Seed* within each subset)

	No Seed	Syzkaller-Generated Seed			Sampled PoC Seed		
		All Seed	Similar Seed	**Diverse Seed**	All Seed	Similar Seed	**Diverse Seed**
Num. of Seed	0	1,000	650	**650**	1,000	650	**650**
Coverage	235,927	241,252	240,266 (↓0.41%)	**247,312 (↑2.51%)**	250,699	246,149 (↓1.82%)	**262,030 (↑4.52%)**

Setup. We organize the experiments along two dimensions, *i.e.,* the source of initial seeds and the distribution of initial seeds, yielding seven configurations in total (see Table 1). First, we consider three groups: (i) *No Seed*, where no initial seed is provided; (ii) *Syzkaller-Generated Seed*, where the initial seeds are produced by Syzkaller itself; and (iii) *Sampled PoC Seed*, where the initial seeds are drawn from the public Syzbot corpus of 22,506 seeds. The third group is motivated by SUNFLOWER [65], which shows that PoC-quality seeds form a substantially more valuable starting corpus. Since its human curated process is too costly to scale, we instead obtain PoC seeds by simple random sampling from the same Syzbot corpus. Second, we further build three subsets according to the distribution of initial seeds: (a) *All Seed*, the unfiltered seed set of size 1,000; (b) *Similar Seed*, selected from the highest-density communities; and (c) *Diverse Seed*, sequentially selected from the lowest-density communities to higher-density communities to maximize cross-community coverage. The community is partitioned for each seed source via module ① (see Sec. 5). The size of *All Seed* is fixed to 1,000, which satisfies the requirement of a 95% confidence level and a 5% error margin when sampling PoC seeds from the 22,506-seed corpus via simple random sampling without replacement. The sizes of *Similar Seed* and *Diverse Seed* are uniformly set to 650, the largest common size that can be stably constructed under the same selection rules across both seed sources, so as to keep all similar/diverse subsets comparable. We use the latest Linux kernel v6.14 as the target, and adopt Syzkaller [17] as the state-of-the-art baseline fuzzer. Considering the randomness of fuzzing, we conduct five independent 24-hour runs for each configuration to obtain reliable results.

Results. As shown in Table 1, compared with No Seed, both Syzkaller-generated Seed w. All Seed and Sampled PoC Seed w. All Seed improve coverage by 2.26% and 6.26%, respectively, with PoC seeds yielding a larger gain. Across both initial seed settings, Diverse Seed achieves a higher coverage than the corresponding All Seed baseline, despite using fewer seeds, yielding gains of 2.51% under Syzkaller-generated Seed and 4.52% under Sampled PoC Seed. In contrast, Similar Seed even suppresses performance, with coverage dropping by 0.41% and 1.82% under the two seed sources. Moreover, the performance of Syzkaller-generated Seed w. Diverse Seed approaches that of the PoC-seed-based groups, while Sampled PoC Seed w. Diverse Seed achieves the largest improvement.

Findings. Fuzzing performance can be jointly shaped by initial seed *value* and seed *diversity*. Diverse Seed consistently outperforms All Seed (+2.51%/+4.52%), while Similar Seed even degrades it (-0.41%/-1.82%), and the diversity gain is markedly larger on top of the more valuable PoC seeds. These results indicate that a high-value corpus only sets a better starting point, and its potential remains locked without continuous seed diversity exploitation.

5 Methodology

Fig. 2 presents the approach overview of SyzDiversity, which is designed to address the three problems introduced in Sec. 1. We only present the three core modules, i.e., *corpus construction and partitioning*, *CPR-aware seed scheduling*, and *CPR-guided seed mutation*, of SyzDiversity in Fig. 2, while other modules like seed minimization, coverage collection, and bug monitoring are the same as Syzkaller [17] and thus are omitted from Fig. 2 for clarity. The module ① uses syscall sequences in proof-of-concepts (PoCs) of reported vulnerabilities as initial seeds. To avoid the instability and overhead of execution path-based similarity measurement in kernel fuzzing, the module ① leverages the structured syscall syntax encoded by Syzlang to characterize seed similarity. It further

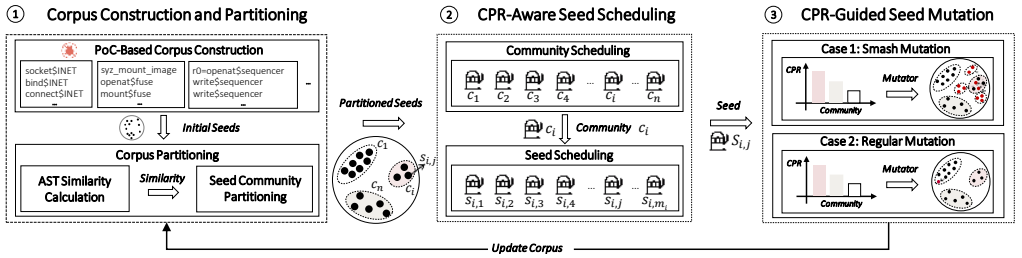


Fig. 2. The approach overview of SyzDiversity

partitions these initial seeds into communities based on their AST (abstract syntax tree) similarity, aiming to mitigate **P1**. Based on the partitioned seeds, the module ② leverages our hierarchical Multi-Armed Bandit (MAB) algorithm to adaptively schedule both the communities and the seeds, aiming to address **P2**. To avoid mistaking local path-level differences for global diversity, the module ② characterizes seed diversity through the distribution and exploration status of semantic communities in the global seed space. Based on this design, the scheduling is guided by our novel diversity metric, *i.e.,* community popularity rate (CPR), together with code coverage. Finally, to continuously maintain and further enhance seed diversity during fuzzing, the module ③ introduces a CPR-guided seed mutation strategy, which achieves adaptive selection between the smash and regular mutation strategy, aiming to addressing **P3**. Newly generated diverse seeds are fed back to the module ①, continuously updating and enriching the seed corpus.

5.1 Corpus Construction and Partitioning

We use the Linux kernel enriched corpus [39] as the initial set of seeds, which contains 22,506 seeds. This corpus is collected by crawling PoCs for confirmed and fixed bugs from Syzbot [14], thereby increasing the value of the seeds and preventing the fuzzing campaign from repeatedly triggering already discovered unfixed bugs. While these initial seeds are considered as higher-value than randomly generated seeds, some seeds can be similar to others. To model and distinguish such similarity, we propose to compute the semantic similarity between seeds based on their ASTs, and partition the initial seeds based on their semantic similarity into communities. Each community is a cohesive group of semantically similar seeds. These initial communities not only provide a solid starting point for the fuzzing campaign, but also offer the diversity guidance for the subsequent scheduling and mutation of seeds, mitigating the problem of seed overvaluation at the early stage of fuzzing.

5.1.1 AST Similarity Calculation. In application-level fuzzing, Lee et al. [30] combine static structural (syntactic) similarity with dynamic execution path (semantic) similarity to model behavioral similarity between seeds. However, this strategy is difficult to extend to kernel fuzzing due to the nature of kernel execution. Furthermore, kernel execution paths are often deeply nested and structurally complex, making execution path analysis more expensive than in user-space programs, which contradicts the design principles of kernel fuzzing, *i.e.,* high throughput and timely feedback.

Fortunately, Syzlang [16] encodes rich semantic information by annotating syscall variants (a set of behaviors for the corresponding syscall) using the `$` symbol, with each variant corresponding to distinct arguments. For example, `write$selinux_create(fd,...)` and `write$selinux_context(fd,...)` are used to trigger SELinux object creation and context setting, respectively. The behavioral differences between such semantic variants are primarily determined by their arguments; *e.g.,* their `fd` arguments point to different SELinux pseudo-device nodes `/selinux/create` and `/selinux/context`. Although both variants invoke the same `write` syscall, they activate distinct kernel behaviors and execution paths. Therefore, by structurally decomposing each seed in terms of syscall names, argument types, nested structures, and concrete values, we can estimate semantic

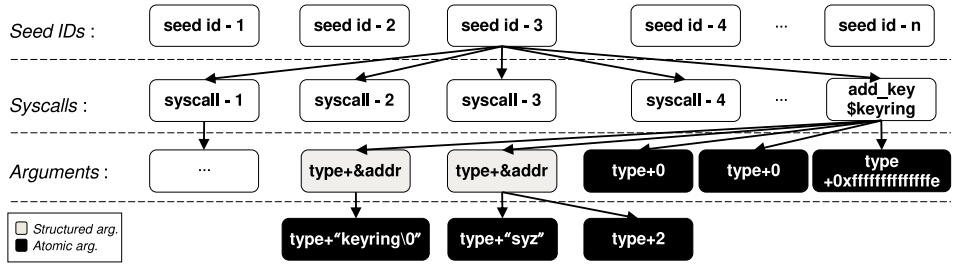


Fig. 3. An illustration of the form of seeds in corpus and the structure of a syscall variant (`add_key$keyring`)

similarity between seeds without requiring actual execution path analysis, while maintaining low overhead, which is suitable for high-efficiency kernel fuzzing.

To achieve this, we structure seeds into their ASTs. Fig. 3 illustrates the structure of the customized AST. Each node in the first layer is hashed to generate a unique identifier for a syscall sequence. Each node in the second layer represents a single syscall variant. For example, the last syscall of the first-layer seed "seed id-3" is `add_key$keyring(&(addr), &(addr)={'syz', 0x2}, 0x0, 0x0, 0xfffffffffffffffe)`. The third layer corresponds to the arguments of each syscall variant, which can be categorized into two types, *i.e.,* structured and atomic (*e.g.,* constants, enums, strings, *etc.*). For structured arguments, the node value represents a memory address automatically managed by Syzkaller, and its child nodes recursively represent the nested structure of the corresponding data type. For atomic arguments, the node value directly denotes the actual value, and does not contain any nested structure. Starting from the third layer, each argument node encodes both its type and value as a concatenated string, in order to avoid ambiguity (such as identical values with different types, or identical variable names with different values) during seed similarity computation.

We compute the similarity between seeds by Eq. 2. Specifically, we use the tree edit distance (TED) algorithm [64] to measure the dissimilarity between two seeds by computing the minimum edit distance required to transform one AST into another ($TED(T_1, T_2)$). To normalize the dissimilarity to a consistent scale and thereby enable a fair similarity computation and comparison, we adopt $NTED_2$ [44] (*i.e.,* the maximum number of nodes in the two trees, $max(|T_1|, |T_2|)$) as our normalization scheme. $NTED_2$ is specifically designed to mitigate bias when dealing with tree structures whose sizes are highly imbalanced, an issue we frequently encounter because syscall sequences often differ markedly in shape and length. Finally, the similarities between seeds are stored in the similarity matrix $M \in \mathbb{R}^{n \times n}$, where $M_{p,q}$ denotes the similarity score between seed p and q.

$$Similarity(T_1, T_2) = 1 - \frac{\mathrm{TED}(T_1, T_2)}{\max(|T_1|, |T_2|)} \tag{2}$$

In the basic TED algorithm, the edit cost for each node is uniformly set to 1, meaning all nodes are treated with equal weight. However, in syscall sequences, the variant differences at the second level largely determine the kernel's behaviors. Thus, we assign a higher edit cost to the second level using a hyperparameter $Cost_s$ for adjustment. Moreover, for argument fields that do not affect the execution path (*e.g.,* memory address values randomly allocated by Syzkaller or randomly generated file system images), we set their edit cost to 0. Similarly, argument fields labeled as placeholders ("`AUTO`" and "`ANY`") [15] are also assigned an edit cost of 0, as they typically represent uncertain types that may be randomly generated by Syzkaller. Such argument fields have minimal impact on code path and constitute only a negligible fraction of the entire corpus. Fields with known types and values (*e.g.,* integers and enums) are assigned an edit cost of 1, serving as the baseline for the edit cost.

5.1.2 Seed Community Partitioning. The similarity matrix M is inherently a weighted graph where seeds are vertices and similarities are edge weights. We adopt a community partition algorithm [3] on

this graph to partition the initial seeds into communities. To suppress the noise introduced by near-zero weights in this graph, we sparsify M by retaining only the top-k strongest edges per node.

5.2 CPR-Aware Seed Scheduling

We model the seed scheduling problem as a hierarchical MAB problem that first schedules communities and then schedules seeds in the selected community, balancing the trade-off between exploration and exploitation. To further achieve diversity-guided scheduling, we design a novel diversity metric, *i.e.,* community popularity rate (CPR), and propose a CPR-aware hierarchical UCB algorithm by defining community- and seed-level rewards to solve the MAB problem.

5.2.1 Community-Level Scheduling. To implement community-level scheduling, we first define the reward function (*i.e.,* the *exploitation* term in Eq. 1). Let the community set be $\{C_1, C_2, \ldots, C_n\}$, where each community C_i has m_i seeds. We define the seed corpus as a two-dimensional array S, where $S_{i,j}$ denotes the j-th seed in the i-th community C_i. The density of each community before round t of the fuzzing process is approximated as the ratio of its size to the total size of all communities (*i.e.,* Eq. 3).

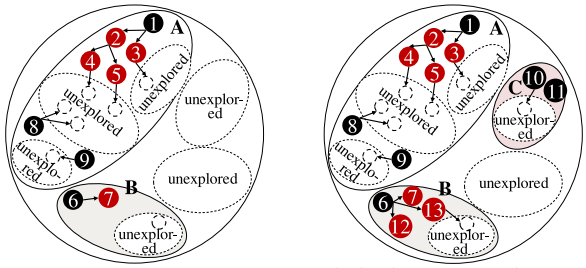
$$density_{C_i}(t) = \frac{|C_i(t)|}{\sum_{j=1}^{n} |C_j(t)|} \tag{3}$$

Intuitively, one scheduling strategy is to prioritize low-density communities, *i.e.,* the community-level reward is naively modeled as the inverse of community density, which may help diversify the seeds [36, 66]. However, if scheduling decisions are made solely based on this, it may introduce scheduling bias as low-density communities do not inherently represent diversity potential. Specifically, even if certain high-density communities contain high-value seeds (especially at the early stage of fuzzing), they may be prematurely ignored or remain unscheduled for extended periods. As the initial seeds constructed in the module ① often exhibit high diversity potential and can rapidly yield high coverage gains, they should not be excluded solely based on their density.

Community Popularity Rate. To achieve diversity-oriented community scheduling, we propose a more discriminative diversity metric, *i.e.,* community popularity rate (CPR), in Eq. 4, where $n_{C_i}(t)$ denotes the number of times community C_i has been scheduled before round t of the fuzzing.

$$CPR_{C_i}(t) = \frac{n_{C_i}(t)}{density_{C_i}(t)} \tag{4}$$

A higher CPR indicates that the community has been underexplored relative to its density, and should therefore be regarded as a potential source of diversity. It is important to note that diversity is a relative concept. Higher CPR values can be observed in both low-density and high-density communities that have been scheduled more frequently. This is because of the significant differences in the amount of code in the different kernel subsystems covered by the community. For example, the `bpf` and `bcachef` subsystems, which have a large codebase, have fixed 303 and 200 bugs respectively, while the smaller `autofs` subsystem has fixed only 4 bugs [14]. Thus, even high-density communities may occupy a small proportion of the overall seed space and still hold value for further scheduling. As illustrated in Fig. 4 (a), the community A has a high density but also leaves a large uncovered region; and a new community C is formed from round t to $t+1$ in Fig. 4 (b). At round $t+1$, the community A has a density of 0.538 and has been scheduled 4 times, resulting in a CPR of $4/0.538 = 7.435$. The community B has a density of 0.308 and has been scheduled twice, with a CPR of $2/0.308 = 6.494$. The newly formed community C has a density of 0.154 and has been scheduled once, with a CPR of $1/0.154 = 6.494$. Although community A has a higher density, its higher CPR indicates that it still holds value for scheduling. This design effectively avoids the misinterpretation that "high density implies low diversity potential", thereby improving the capability of our scheduling strategy to adapt to global diversity.



(a) Seed space at round t (b) Seed space at round $t+1$

Fig. 4. An illustration of seed space evolution

Community-Level Reward. To effectively schedule communities, we compute the scheduling reward of a community at each round as a weighted sum of its average *new* coverage and its CPR (*i.e.,* Eq. 5). Specifically, $cov_{S_{i,j}}$ denotes the new coverage contributed by seed $S_{i,j}$, and $\delta_{S_{i,j}}$ denotes whether seed $S_{i,j}$ triggers a crash. $1+\delta_{S_{i,j}}$ amplifies the new coverage if it further triggers a crash. The weighting factor α ($\alpha \geq 1$, which is introduced because CPR and code coverage are measured at different scales) modulates the contribution of CPR in the scheduling strategy.

$$r_{C_i}(t) = \text{Avg}_{S_{i,j} \in C_i(t)}(cov_{S_{i,j}} \cdot (1+\delta_{S_{i,j}})) + \alpha \cdot CPR_{C_i}(t) \tag{5}$$

Based on this, we compute the average reward of community C_i by Eq. 6, where $R_{C_i}(t)$ denotes the rounds when C_i is selected (*i.e.,* $|R_{C_i}(t)| = n_{C_i}(t)$), which serves as the *exploitation* term in Eq. 1. A higher exploitation value indicates that the community has a greater potential value and thus a higher probability of being scheduled.

$$\bar{r}_{C_i}(t) = \frac{\sum_{t' \in R_{C_i}(t)} r_{C_i}(t')}{n_{C_i}(t)} \tag{6}$$

Finally, the community-level UCB score is computed and updated by Eq. 7, balancing the trade-off between exploration and exploitation. A higher score indicates a higher priority for scheduling.

$$UCB_{C_i}(t) = \bar{r}_{C_i}(t) + \sqrt{\frac{2ln(t)}{n_{C_i}(t)}} \tag{7}$$

5.2.2 Seed-Level Scheduling. After a community is scheduled, seed-level scheduling is performed within the community. Selected seeds are passed to the mutation module. For each seed $S_{i,j}$, we compute its scheduling reward as the average *new* coverage growth rate of all the seeds mutated from $S_{i,j}$, as formulated in Eq. 8. Specifically, $K_{S_{i,j}}(t)$ denotes the set of all seeds mutated from $S_{i,j}$ up to round t. For a mutated seed $S^k_{i,j} \in K_{S_{i,j}}(t)$, $cov_{S^k_{i,j}}$, $T_{S^k_{i,j}}$ and $\delta_{S^k_{i,j}}$ respectively denote its new coverage, its execution time, and whether it triggers a crash. Following SyzVegas [55], $cov_{S^k_{i,j}}/T_{S^k_{i,j}}$ is the new coverage growth per unit time of $S^k_{i,j}$, which favors a seed that has high new coverage and short execution time. Similar to Eq. 5, $1+\delta_{S^k_{i,j}}$ amplifies the new coverage growth rate if it triggers a crash.

$$r_{S_{i,j}}(t) = \text{Avg}_{S^k_{i,j} \in K_{S_{i,j}}(t)}((\frac{cov_{S^k_{i,j}}}{T_{S^k_{i,j}}}) \cdot (1+\delta_{S^k_{i,j}})) \tag{8}$$

We compute the average reward of seed $S_{i,j}$ by Eq. 9, where $R_{S_{i,j}}(t)$ denotes the rounds when $S_{i,j}$ is selected (*i.e.,* $|R_{S_{i,j}}(t)| = n_{S_{i,j}}(t)$).

$$\bar{r}_{S_{i,j}}(t) = \frac{\sum_{t' \in R_{S_{i,j}}(t)} r_{S_{i,j}}(t')}{n_{S_{i,j}}(t)} \tag{9}$$

Algorithm 1 CPR-Aware Seed Scheduling

Require:
C: Community Set; S: Seed Array.
1: **for** $i = 1$ **to** n **do** ▷ **Initialization Phase**
2: $n_{C_i}(t=1) \leftarrow 1$, Compute $density_{C_i}(t=1)$ and $CPR_{C_i}(t=1)$
3: **for** $j = 1$ **to** $|C_i|$ **do**
4: $cov_{S_{i,j}}, \delta_{S_{i,j}} \leftarrow$ Execute($S_{i,j}$)
5: $n_{S_{i,j}}(t=1) \leftarrow 1$
6: Compute $UCB_{S_{i,j}}(t=1)$ ▷ Eq. 10
7: **end for**
8: Compute $UCB_{C_i}(t=1)$ ▷ Eq. 7
9: **end for**
10: **for** all $t = 1, 2, 3, \ldots$ **do** ▷ **Scheduling Phase**
11: $C_i \leftarrow \arg\max(UCB_{C_i}(t))$
12: $n_{C_i}(t+1) \leftarrow n_{C_i}(t) + 1$
13: $S_{i,j} \leftarrow \arg\max(UCB_{S_{i,j}}(t))$
14: $n_{S_{i,j}}(t+1) \leftarrow n_{S_{i,j}}(t) + 1$
15: $Mutate(S_{i,j}, t)$ ▷ **Mutation Phase (See Algorithm 2)**
16: **end for**

Finally, the seed-level UCB score is computed and updated using Eq. 10. A higher score indicates a higher priority for scheduling.

$$UCB_{S_{i,j}}(t) = \bar{r}_{S_{i,j}}(t) + \sqrt{\frac{2ln(t)}{n_{S_{i,j}}(t)}} \tag{10}$$

5.2.3 The Overall Scheduling Algorithm. With the reward functions defined for communities and seeds, our CPR-aware UCB algorithm (Algorithm 1) is applied to determine which community and seed are scheduled in each fuzzing round. Its inputs are a community set C, and a seed array S.

During the initialization phase (Lines 1–9), all the statistical variables required for the reward functions of communities and seeds at round $t = 1$ are computed. The scheduling count n_{C_i} of each community C_i is first initialized to 1, enabling the initialization of $density_{C_i}$ and CPR_{C_i} at round 1 (Line 2). For each seed $S_{i,j}$ in the community, its coverage gain $cov_{S_{i,j}}$ and crash indicator $\delta_{S_{i,j}}$ are obtained by executing the seed (Line 3), and its scheduling count $n_{S_{i,j}}$ is initialized to 1 (Line 5). After all the seeds in the community are processed, the initial seed-level and community-level scheduling scores $UCB_{S_{i,j}}$ and UCB_{C_i} at round 1 are computed (Lines 6 and 8).

Once these variables are initialized, the algorithm proceeds to the scheduling phase (Lines 10–16). The fuzzing campaign starts from round t = 1, and the community C_i with the highest UCB_{C_i} is scheduled (Line 11), and then its scheduling count is incremented (Line 12). In the scheduled community, the seed $S_{i,j}$ with the highest $UCB_{S_{i,j}}$ is scheduled (Line 13), and its scheduling count is incremented (Line 14). Notably, since seeds $S_{i,j}$ has not been mutated in the first round (*i.e.*, $t = 1$), both $cov_{S^k_{i,j}}$ and $\delta_{S^k_{i,j}}$ are zero. As a result, early seed scheduling is determined by the exploration term and thus more random. The scheduled seed is then mutated by Algorithm 2 (Line 15).

To mitigate the cold-start issue of UCB caused by its sensitivity to sparse early rewards, SyzDiversity employs a lightweight warm-up strategy in the first four hours. It selects seeds from highest-CPR communities with a small probability, and then switches to the full MAB-based scheduling.

5.3 CPR-Guided Seed Mutation

After scheduling a seed for mutation, we design a mutation strategy to adaptively adjust the mutation frequency based on CPR to accelerate exploration of uncovered regions within communities and enhance the likelihood of generating entirely new communities.

Algorithm 2 CPR-Guided Seed Mutation

Require: $S_{i,j}$: j-th Seed in Community C_i; t: Current MAB Round.

1: Initialize mutated seeds $queue = \emptyset$ ▷ **Mutation Scheduling Phase**
2: $rank_{C_i} \leftarrow \text{DescendingRanking}(CPR_{C_i}(t))$
3: **if** $rank_{C_i} \leq threshold_r$ **then**
4: $queue \leftarrow \text{smash\_mutation}(S_{i,j})$
5: **else**
6: $queue \leftarrow \text{regular\_mutation}(S_{i,j})$
7: **end if**
8: **for** $k = 1$ **to** $|queue|$ **do** ▷ **Seed Execution Phase**
9: $cov_{queue_k}, \delta_{queue_k}, T_{queue_k} \leftarrow \text{Execute}(queue_k)$
10: **if** $cov_{queue_k} > 0$ **or** $\delta_{queue_k} > 0$ **then**
11: $sim \leftarrow \max_{p \in [1,n]} \text{Similarity}(queue_k, C_p)$
12: **if** $sim \geq threshold_C$ **then** ▷ **Assign to a Community**
13: Assign $queue_k$ to C_p
14: **else** ▷ **Establish a New Community**
15: $C_{n+1}(t+1) \leftarrow \{queue_k\}$
16: $n_{C_{n+1}}(t+1) \leftarrow 1$
17: **end if**
18: **end if**
19: **end for**
20: Compute UCB scores at round $t+1$ ▷ **Compute UCB Scores**

5.3.1 CPR-Guided Mutation Scheduling. In the original design of Syzkaller [17], the scheduled seed only undergoes regular mutation (*i.e.,* the low-frequency strategy) to generate a single new seed. Only if this new seed improves coverage will the scheduled seed be further subjected to smash mutation (*i.e.,* the high-frequency strategy) to generate multiple new seeds. Otherwise, although the scheduled seed still remains in the corpus, its chance of being mutated again becomes much smaller as the corpus keeps growing (the corpus already contains around 20K seeds initially). Thus, the effectiveness of regular mutation largely determines whether the scheduled seed can be fully explored. Even for the scheduled seed from a high-CPR community, its potential value may be wasted if regular mutation fails to explore new paths. Intuitively, customizing regular mutation strategies for different seeds is a way to improve mutation effectiveness.

However, there are over 3,000 syscall variants [17], with different parameter combinations and nested structures being extremely complex. This high complexity imposes a significant barrier in terms of expert knowledge and requires substantial human efforts to design customized strategies. In contrast, directly adjusting mutation frequency (*i.e.,* prioritizing smash mutation) to generate multiple new seeds for the scheduled seed from a high-CPR community can alleviate this problem to some extent. This method not only avoids wasting the potential value of the scheduled seed, but also significantly reduces human efforts while maintaining simplicity and efficiency.

Therefore, we make improvements from two aspects. First, we refactor Syzkaller's mutation module to integrate smash and regular mutation into a unified framework as two distinct strategies. Second, we design a CPR-guided mutation scheduling mechanism without modifying the mutation operators. The rationale is that Syzkaller already has a well-designed set of mutation operators, and our focus is on dynamically regulating the mutation frequency using CPR to efficiently populate high-CPR communities and generate new communities, thereby approaching the coverage upper bound.

5.3.2 The Overall Mutation Algorithm. Our CPR-guided seed mutation algorithm is shown in Algorithm 2, whose inputs consist of the scheduled seed $S_{i,j}$, and the current MAB round t. In the mutation scheduling phase (Lines 1–7), the queue for storing seeds mutated from $S_{i,j}$ is initialized to empty (Line 1), and the CPR values for all communities at round t are ranked in descending order (Line 2). If

the CPR of the community containing $S_{i,j}$ is among the top $threshold_r$, the community is considered as having higher diversity potential, and the high-frequency smash mutation is applied on $S_{i,j}$ to explore the potential (Lines 3–4); otherwise, the low-frequency regular mutation is used (Lines 5–6).

In the seed execution phase (Lines 8–20), each mutated seed in the queue is executed to record its new coverage, whether it triggers a crash, and its execution time (Line 9). If a mutated seed leads to new coverage or triggers a crash, it is considered as high-value, and the community C_p whose centroid seed has the maximum similarity sim to this mutated seed is identified (Line 11). If $sim \geq threshold_C$, this mutated seed is assigned to the most similar community C_p (Line 12-13). Otherwise, no similar community is found, and a new community is established (Line 14-16). Finally, the UCB scores for all communities and seeds are updated for the next round, *i.e.,* $t + 1$ (Line 20). Regarding the engineering design of this dynamic partitioning, this choice is essentially a performance trade-off. As seeds accumulate rapidly during fuzzing, continuously repartitioning seeds and repeatedly determining new community centroids would incur significant computational overhead. Therefore, throughout a single fuzzing campaign, we reuse the initial partitioning results and their centroid seeds to exploit seed diversity under an acceptable performance overhead.

6 Evaluation

We evaluate the effectiveness and efficiency of SyzDiversity by answering five research questions.

- **RQ1: Diversity Expansion Evaluation.** Can SyzDiversity mitigate the early-stage overvaluation problem by continuously expanding the diversity of the explored seed space?
- **RQ2: Effectiveness Evaluation.** What is the effectiveness of SyzDiversity?
- **RQ3: Efficiency Evaluation.** How efficient is SyzDiversity in generating and executing seeds?
- **RQ4: Ablation Study.** How do the different modules in SyzDiversity affect its effectiveness?
- **RQ5: Parameter Sensitivity.** How do the parameters affect the effectiveness of SyzDiversity?

6.1 Evaluation Setup

State-of-the-Arts. We selected four kernel fuzzers, *i.e.,* Syzkaller [17] (the de-facto standard, and basis for most later work), Healer [51] (generating semantically meaningful seeds via syscall-dependency analysis, serving as a baseline for effective comparison with the initial seeds constructed in Module ①), SyzVegas [55] (also employing a MAB scheduler similar to SyzDiversity), and SyzGPT [66] (the latest kernel fuzzer, and extensively evaluated against prior work).

Seed Corpora. State-of-the-art fuzzers like Syzkaller, SyzVegas, and SyzGPT do not incorporate initial seeds by default. To more precisely evaluate the impact of diverse initial seeds on fuzzing effectiveness, as well as the overall effectiveness of SyzDiversity, we use two sets of seed corpora with the same size. The first is the 22,506 initial seeds (†) generated by Healer, and the second consists of 22,506 initial seeds (‡) from the Linux kernel enriched corpus [39] (covering all available seeds except those linked to disclosed but unfixed bugs). Both corpora are partitioned by SyzDiversity.

Experiment Configuration. We use the long-term Linux kernel, *i.e.,* Linux v5.15, and the latest Linux kernel, *i.e.,* Linux v6.14. To alleviate the influence of the inherent randomness, each experiment is repeated five times, and each experiment has a time budget of 72 hours. All the kernels are compiled under the same configuration with the common features enabled (*e.g.,* KCOV [54] and KASAN [26]). All experiments were conducted on a machine running Ubuntu 20.04 with 48 CPUs and 64GB memory, powered by AMD EPYC 9754 128-Core Processor.

6.2 Diversity Expansion Evaluation (RQ1)

Early-stage overvaluation makes fuzzers to repeatedly schedule and mutate seeds that obtain coverage gains in the early stage, even when these seeds lead only to locally similar communities. To investigate the impact of the early-stage overvaluation issue on seed space evolution, and to evaluate

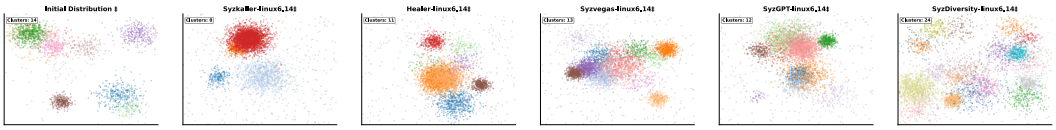


Fig. 5. Visualization of seed distribution evolution across different fuzzers after 72 hours

Table 2. Comparison results († and ‡ denote the seed corpus generated by Healer and SyzDiversity, and std and cv denote the standard deviation and the coefficient of variation, respectively)

Kernel	Metric	Syzkaller†	Healer†	SyzVegas†	SyzGPT†	SyzDiversity †	Syzkaller‡	Healer‡	SyzVegas‡	SyzGPT‡	SyzDiversity ‡
v5.15	Cov.	109,314(-7.4%)	111,019(-5.7%)	115,141(-1.9%)	82,706(-41.9%)	**117,351**	122,120(-20.1%)	136,831(-7.2%)	136,712(-7.3%)	105,515(-39.0%)	**146,644**
	Cov. (std, cv)	(1079.6, 0.99%)	(1471.9, 1.33%)	(828.3, 0.72%)	(1439.6, 1.74%)	(1670.4, 1.42%)	(1656, 1.36%)	(1728.6, 1.26%)	(1755.8, 1.28%)	(1619.7, 1.54%)	(2551.4, 1.74%)
	Bug	33(1.4×)	19(2.4×)	29(1.6×)	18(2.6×)	**46**	25(2.0×)	4(12.8×)	26(2.0×)	20(2.5×)	**51**
	Bug (std, cv)	(0.45, 1.36%)	(0.25, 1.32%)	(0.6, 2.06%)	(0.38, 2.10%)	(1.03, 2.23%)	(0.51, 2.04%)	(0.08, 2.15%)	(0.56, 2.16%)	(0.42, 2.12%)	(0.91, 1.79%)
	New	2(2.0×)	0(-)	1(4.0×)	0(-)	**4**	4(1.0×)	1(4.0×)	2(2.0×)	2(2.0×)	**4**
	Thpt.	**16,644,334**	7,279,467(2.3×)	4,095,875(4.1×)	2,237,345(7.4×)	2,860,243(5.8×)	**20,904,104**	3,948,172(5.3×)	4,332,282(4.8×)	2,568,419(8.1×)	3,234,712(6.5×)
	Cov./Thpt.	0.007	0.015	0.028	0.037	**0.041**	0.006	0.035	0.032	0.041	**0.045**
v6.14	Cov.	326,780(-6.7%)	258,662(-34.9%)	344,963(-1.1%)	268,015(-30.2%)	**348,826**	424,184(-9.8%)	375,922(-23.9%)	413,291(-12.7%)	370,365(-25.8%)	**465,824**
	Cov. (std, cv)	(3926.9, 1.20%)	(1754.1, 0.68%)	(3276, 0.95%)	(4462.4, 1.66%)	(1178.6, 0.34%)	(3494.7, 0.82%)	(2971.5, 0.79%)	(6582.8, 1.59%)	(1150.5, 0.31%)	(2480.9, 0.53%)
	Bug	68(1.4×)	4(23.8×)	19(5.0×)	14(6.8×)	**95**	82(1.6×)	5(26.0×)	18(7.2×)	34(3.8×)	**130**
	Bug (std, cv)	(0.88, 1.29%)	(0.08, 2.08%)	(0.33, 1.72%)	(0.29, 2.08%)	(1.34, 1.41%)	(1.8, 2.20%)	(0.08, 1.54%)	(0.38, 2.10%)	(0.49, 1.45%)	(1.59, 1.22%)
	New	5(1.4×)	0(-)	3(2.3×)	0(-)	**7**	5(1.8×)	0(-)	3(3.0×)	2(4.5×)	**9**
	Confirmed	39(1.1×)	2(21.0×)	11(3.8×)	6(7.0×)	**42**	46(1.3×)	1(62.0×)	10(6.2×)	16(3.9×)	**62**
	Fixed	8(2.1×)	1(17.0×)	3(5.7×)	1(17.0×)	**17**	10(2.5×)	0(-)	4(6.3×)	5(5.0×)	**25**
	Thpt.	5,393,904(2.1×)	2,505,250(4.5×)	8,346,779(1.3×)	**11,201,609**	2,549,262(4.4×)	8,332,785(1.3×)	2,788,207(4.0×)	8,245,985(1.4×)	**11,143,423**	3,862,436(2.9×)
	Cov./Thpt.	0.061	0.103	0.041	0.024	**0.137**	0.051	**0.135**	0.05	0.033	0.121

whether diversity-guided fuzzing can mitigate this phenomenon, we visualize the seed distributions of all fuzzers in the v6.14-‡ setting after 72 hours of fuzzing, using the community partitioning method in module ① together with a dimensionality reduction technique [11]. As shown in Fig. 5, the initial PoC seeds form multiple relatively separated semantic communities. Since dimensionality reduction may introduce visual distortion, colors are only used to indicate cluster membership.

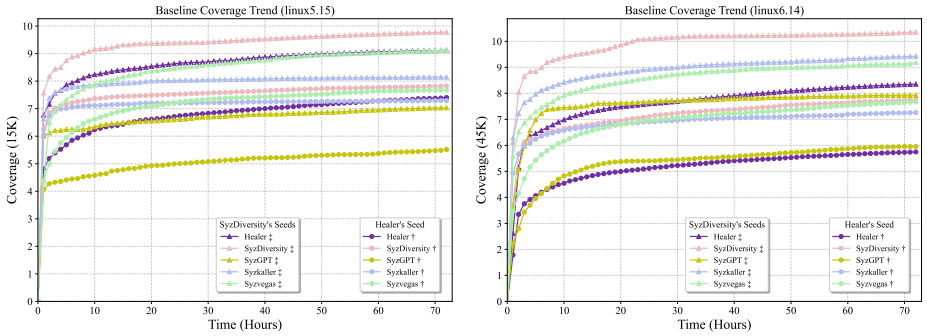
After 72 hours of fuzzing, baseline fuzzers exhibit varying degrees of seed space contraction. Quantitatively, Fig. 5 shows that, compared with the initial distribution containing 14 semantic communities, the four baseline fuzzers produce only 8, 11, 13, and 12 communities, respectively. On average, the number of semantic communities decreases by 21.4% relative to the initial corpus. This phenomenon suggests that seeds obtaining coverage gains at the early stage are repeatedly exploited during subsequent fuzzing, reflecting the early-stage overvaluation issue.

In contrast, SyzDiversity produces more communities than all baseline fuzzers, improving the number of semantic communities by 200.0%, 118.2%, 84.6%, and 100.0% over Syzkaller, Healer, SyzVegas, and SyzGPT, respectively. This indicates that diversity-guided scheduling and mutation can effectively prevent the search process from prematurely converging to locally dense communities. Specifically, module ① establishes the foundation for modeling seed diversity through semantic community partitioning. Based on this representation, module ② continuously guides the scheduling strategy toward diversity expansion rather than repeatedly exploiting locally dense communities. Meanwhile, module ③ increases the probability of generating semantically diverse seeds and exploring previously unseen communities through the adaptive high-frequency mutation mechanism, thereby effectively mitigating the seed-space contraction caused by early-stage overvaluation.

6.3 Effectiveness Evaluation (RQ2)

Table 2 shows the results of SyzDiversity compared to the state-of-the-arts on two kernel versions.

6.3.1 Code Coverage. The **Cov.** column in Table 2 reports the total number of basic blocks covered by each fuzzer during the 72-hour period, and Fig. 6(a) and (b) illustrate the growth of code coverage. Overall, SyzDiversity achieves statistically significant improvements in code coverage over all the state-of-the-art fuzzers (p-value < 0.01 under the Mann–Whitney U test). Specifically, on v5.15, SyzDiversity achieves an average coverage improvement of 13.8%, 6.5%, 4.6%, and 40.5% over Syzkaller, Healer, SyzVegas, and SyzGPT, respectively. On v6.14, the average improvements are



(a) v5.15 kernel (b) v6.14 kernel

Fig. 6. Growth of coverage achieved by SyzDiversity and baselines for Linux kernels v5.15 and v6.14

8.3%, 29.4%, 6.9%, and 28.0%. This owes to the synergy between our CPR-aware scheduling and CPR-guided mutation modules in generating diverse seeds.

Moreover, the choice of initial seeds plays a critical role in the achieved coverage of all fuzzers. The initial seeds constructed by SyzDiversity significantly improve the coverage of all fuzzers by an average of 21.3% and 33.3% on v5.15 and v6.14, respectively. As a result, all fuzzers can start fuzzing from a higher initial coverage level. Building on this stronger starting point, SyzDiversity further leverages seed diversity at an early stage through CPR-aware scheduling and CPR-guided mutation, thereby accelerating coverage growth in subsequent fuzzing. Specifically, as shown in Fig. 6 (a) and (b), within the initial 5 hours, SyzDiversity achieves a faster coverage increase, as it is able to quickly populate seeds for scheduled diverse communities based on partitioned intial PoC seeds. This trend reappears during the 6–10 hour or 10–20 hour intervals, where the coverage upper bound is broken, allowing coverage growth to consistently remain at the best level.

Notably, the overall coverage of all fuzzers decreases after switching from v6.14 to v5.15. The reason is that the v5.15 kernels have nearly 70% less basic blocks (536,000 vs. 1,760,000) compared to the v6.14, and the number of supported syscall variant interfaces is correspondingly close to 1,700 less (2,176 vs. 3,862). This difference in the number of basic blocks and the interface drift issue explains the overall coverage degradation in v5.15. Moreover, the low coefficients of variation (cv < 3%) across all runs indicate that the results are stable and consistent.

6.3.2 Bug-Finding Capability. The **Bug** column in Table 2 reports the total number of vulnerabilities discovered by each fuzzer, while the **New** column indicates the number of previously unreported vulnerabilities, confirmed through manual inspection of authoritative sources such as LKML [53] and Syzbot [14]. Additionally, we further classify only the vulnerabilities found in v6.14 into **Confirmed** and **Fixed** categories, since vulnerability reports are typically validated and maintained on the latest kernel version, making reliable status confirmation infeasible for v5.15.

On v5.15, SyzDiversity finds 46 and 51 vulnerabilities in the † and ‡ settings, respectively, along with a higher number of new bugs, *i.e.,* 4 and 4, respectively. In the † setting, it detects 1.4×, 2.4×, 1.6×, and 2.6× more vulnerabilities than Syzkaller, Healer, SyzVegas, and SyzGPT, respectively. In the ‡ setting, SyzDiversity similarly outperforms the same baselines by 2.0×, 12.8×, 2.0×, and 2.6×.

On v6.14, SyzDiversity still discovers more vulnerabilities in the † and ‡ settings, reporting 95 and 130 vulnerabilities, respectively. In the † setting, it outperforms Syzkaller, Healer, SyzVegas, and SyzGPT by 1.4×, 23.8×, 5.0×, and 6.8×, respectively. In the ‡ setting, SyzDiversity similarly achieves improvements of 1.6×, 26.0×, 7.2×, and 3.8× over the same baselines. Moreover, it detects a higher number of new, confirmed and fixed vulnerabilities than all baselines.

Table 3. Vulnerabilities discovered by SyzDiversity

Module	Crash Type	Operation	Status	Module	Crash Type	Operation	Status
lib	use-after-free	__bitmap_weight	$C(\nabla)$	driver/sound	slab-out-of-bounds	snd_seq_oss_synth_sysex	$F^C(\nabla)$
scheduler	slab-use-after-free	try_to_wake_up	$F(\nabla)$	fs/hfs	slab-out-of-bounds	hfsplus_bnode_read	$F^C(\nabla)$
fs/udf	use-after-free	udf_add_fid_counter	$F(\nabla)$	mm	slab-out-of-bounds	isolate_migratepages_block	$R(\nabla)$
fs/jfs	read_message failed	LogSyncRelease	$R(\nabla)$	fs/udf	use-after-free	udf_statfs	$R(\nabla)$
fs	slab-out-of-bounds	__bh_read	$F^C(\nabla)$	fs/ntfs3	warning	ntfs_get_block_vbo	$F(\nabla)$
fs/gfs2	task hung	gfs2_glock_nq	$R(\nabla)$	kernel/rcu	soft lockup	note_gp_changes	$R(\nabla)$
fs/exfat	soft lockup	exfat_clear_bitmap	$F(\nabla)$	mm	task hung	shmem_swapin_folio	$R(\nabla)$
fs/jfs	possible deadlock	jfs_rsync	$F(\nabla)$	fs/bfs	task hung	bfs_lookup	$R(\nabla)$
fs/ocfs2	use-after-free	ocfs2_block_group_alloc	$R(\nabla)$	fs/jfs	null-ptr-deref	txLazyCommit	$R(\nabla)$
fs/gfs2	slab-out-of-bounds	gfs2_pin	$R(\nabla)$	fs	slab-use-after-free	chrdev_open	$R(\nabla)$
fs/bcachefs2	shift-out-of-bounds	bch2_trans_iter_init_outlined	$R(\nabla)$	driver/block	soft lockup	loop_configure	$R(\nabla)$
net/dccp	BUG: hc exception	ccid3_update_send_interval	$R(\triangle)$	fs/bcachef	missing-bounds-check	validate_bset_keys	$R(\triangle)$
mm	slab-use-after-free	find_lock_entries	$R(\triangle)$	driver/usb	warning	usb_lbniturb	$C(\triangle)$
kernl/time	spinlock bad magic	lock_timer_base	$C(\triangle)$	fs/netfs	tash hung	netfs_writepages	$R(\triangle)$
kernel/bpf	general protection fault	alloc_bulk	$R(\triangle)$	kernel	possible deadlock	smp_call_function_many_cond	$R(\triangle)$
driver/block	soft lockup	loop_configure	$R(\triangle)$	arch	vmalloc-out-of-bounds	copy_from_buffer	$C(\triangle)$

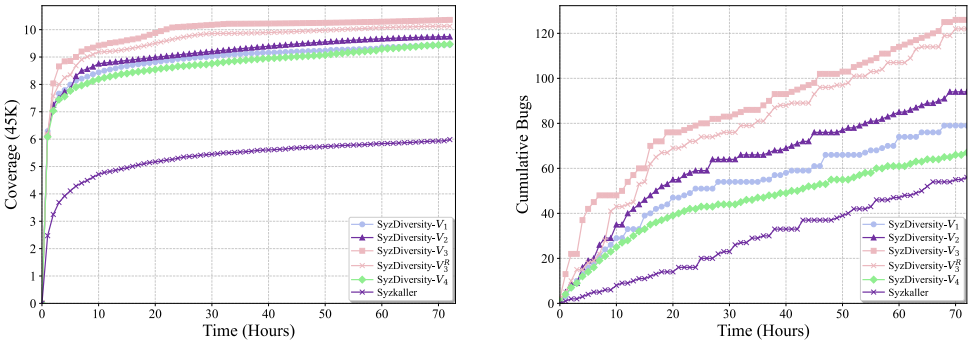
Abbreviations — R: Reported, C: Confirmed, F: Fixed. F^C: Received a CVE identifier. ∇: Linux v6.13, $\triangle$: Linux v6.14.

SyzDiversity demonstrates a superior bug-finding capability, primarily due to its tendency to use high-frequency mutation strategy to rapidly populate high-potential communities. The low coefficients of variation further confirm the robustness of this performance. It is important to notice that the initial seeds are derived from PoCs that have already triggered vulnerabilities, and the corresponding communities, along with their mutation-generated descendants, are often not accurately retained or managed by existing state-of-the-art fuzzers. In contrast, SyzDiversity captures critical context and preconditions that can trigger these vulnerabilities, and actively prioritizes these communities by allocating substantial smash mutation opportunities to their constituent seeds. As a result, it is able to explore deeper execution paths and find a larger number of new vulnerabilities.

6.3.3 Two-Month Fuzzing. Before the experiments reported in Table 2, we conducted a long-term fuzzing with SyzDiversity, and Table 3 provides detailed information about the new vulnerabilities detected during this two-month period. Specifically, SyzDiversity discovered 32 new vulnerabilities spanning kernel versions v6.13 to v6.14 (as the kernel version transitioned during this period). Among them, 12 vulnerabilities have been confirmed; and 8 of the confirmed vulnerabilities have already been fixed and have either been merged or are in the process of being merged into the mainline kernel, while 4 of the confirmed vulnerabilities are currently in the fixing queue. Three of the fixed vulnerabilities have been assigned CVE identifiers, which are omitted for anonymity. These vulnerabilities pose significant risks, as they can be exploited by attackers to perform privilege escalation, cause denial-of-service (DoS), or corrupt data, thereby compromising system security.

6.3.4 Case Study. SyzDiversity discovered a slab-out-of-bounds write vulnerability in GFS2 (*i.e.,* Global File System 2). It occurs when toggling the `GFS2_DIF_JDATA` flag of a file, which controls whether file data is subject to journaling. Depending on the flag's value, the kernel uses different data structures within the folio, *i.e.,* either `buffer_head` or `iomap_folio_state`. Due to the failure to properly clear the existing folio state during the flag transition, the system incorrectly interprets `iomap_folio_state` as `buffer_head`, ultimately leading to a slab out-of-bounds access and memory corruption. This vulnerability also triggers a separate issue along a different execution path; *i.e.,* a kobject name collision occurs when remounting with a reused locktable name, causing the mount operation to fail. Under certain erroneous user behaviors, it may further evolve into a use-after-free (UAF) vulnerability, *e.g.,* if a registration is attempted on a partially unmounted device.

This vulnerability was triggered by a structurally complex and semantically diverse syscall sequence, which not only involves GFS2 mounting operations, but also spans multiple subsystems, including TCP socket initialization and HID input handling, forming a complex I/O context. The key sequence is `syz_mount_image$gfs2` → `socket$inet_tcp` → `openat$sw_sync` → `select`



(a) Coverage growth of SyzDiversity variants (b) Cumulative bugs of SyzDiversity variants

Fig. 7. Ablation study of SyzDiversity variants

→ `setsock- opt$inet_tcp_int` → `chdir` → `openat` → `write$UHID_INPUT` (invoked twice consecutively). This sequence does not originate from the initial seed corpus, but is instead evolved from a newly formed community with a high CPR. By prioritizing aggressive mutation within such high-CPR communities, SyzDiversity quickly explores new execution paths and exposes hidden vulnerabilities under complex conditions. Prior fuzzers often fail to promptly identify and intensively mutate these high-CPR communities, being ineffective in finding such deep vulnerabilities.

6.4 Efficiency Evaluation (RQ3)

The **Thpt.** column in Table 2 shows the total throughput (*i.e.,* the total number of seeds executed) of all fuzzers over 72 hours. Overall, SyzDiversity exhibits lower throughput, mainly due to three reasons. First, each time a seed is scheduled, SyzDiversity globally updates the reward function values for all communities and seeds, which incurs overhead similar to that of SyzVegas [55]. Second, each time a new seed is mutated and leads to new coverage improvement, SyzDiversity immediately calculates the similarity between the new seed and the communities to determine whether to assign it to an existing community or create a new one. This is the key factor of the decreased throughput, but it is essential for implementing CPR-aware mutation. Third, throughput is also influenced by the effectiveness of seeds. When a mutated seed leads to a coverage improvement, these fuzzers further verify the stability of the new coverage, which requires repeated execution. Once a seed is successfully verified, it is minimized and stored in its simplest form. During the verification and minimization process, subsequent seeds are queued. In contrast, when the newly generated seeds are mostly ineffective (*i.e.,* do not lead to a coverage improvement), the verification and minimization process is reduced, the seeds execute faster, and the throughput is gained to some extent. For example, in † and ‡ settings on v5.15, Syzkaller exhibits a higher throughput, but the coverage does not increase proportionally.

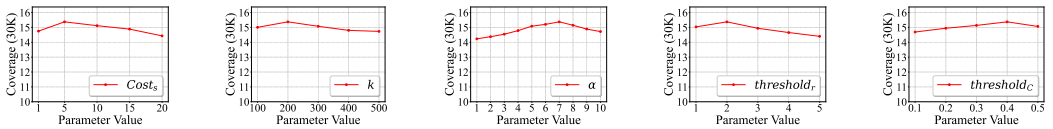
To fairly assess the trade-off between coverage and throughput, we calculate the average coverage per seed for each fuzzer, which is reported in the **Cov./Thpt.** column in Table 2. SyzDiversity achieves the highest average coverage per seed in the † and ‡ settings on v5.15 as well as in the † setting on v6.14. This indicates that the seeds generated by SyzDiversity are generally more effective over the 72-hour campaign. Despite executing fewer seeds, it still yields a higher overall coverage gain. However, in the ‡ setting on v6.14, Healer achieves a higher average coverage per seed, primarily due to its smaller number of executed seeds. Notably, **Cov./Thpt.** should be interpreted as a complementary metric to the coverage and throughput, rather than a standalone indicator. Otherwise, it may lead to a biased conclusion. Overall, SyzDiversity strikes a more favorable balance between effectiveness and efficiency.

6.5 Ablation Study (RQ4)

We conducted an ablation study on the v6.14 kernel to evaluate the impact of the three core modules in SyzDiversity on the code coverage and bug-finding capability, using Syzkaller as the baseline. Specifically, we create four ablated versions. First, we only enable module ① (*i.e.,* corpus construction and partitioning), but disable module ② and ③, which is denoted as SyzDiversity-V_1. Second, we further enable module ② (*i.e.,* CPR-aware seed scheduling), but still disable module ③, which is denoted as SyzDiversity-V_2. Third, we further enable module ③ (*i.e.,* CPR-guided seed mutation), which is actually the complete version of SyzDiversity, here denoted as SyzDiversity-V_3. To further evaluate the impact of the warm-up strategy on the overall SyzDiversity framework, we create an additional variant based on SyzDiversity-V_3, denoted as SyzDiversity-V_3^R, where "R" indicates random warm-up, *i.e.,* replacing the highest-CPR community selection in the first 4-hour warm-up phase with random selection. Finally, to evaluation the effectiveness of our metric CPR, we use community density to guide scheduling and mutation, denoted as SyzDiversity-V_4.

Fig. 7 shows the coverage growth and cumulative bugs of these ablated versions over 72 hours. SyzDiversity-V_1 achieves the largest improvement over Syzkaller in both coverage and bug discovery within the first 20 hours, demonstrating that initial PoC seeds constructed by SyzDiversity substantially boost early-stage fuzzing effectiveness. Compared to SyzDiversity-V_1, SyzDiversity-V_2, equipped with our CPR-aware seed scheduling, tends to prioritize communities with higher UCB scores (Eq. 10), which is associated with a higher likelihood of triggering bugs (Eq. 8). This advantage becomes more evident at approximately 14 and 27 hours of fuzzing. In terms of coverage, SyzDiversity-V_2 also achieves a peak coverage increase of 3.9% during the 10–59 hour interval. At other times, the improvement is relatively modest. This can be attributed to the fact that, although V_2 is able to schedule communities with a higher potential, the mutations of seeds within these communities occur at a consistently low frequency. Once a randomly mutated seed fails to yield a new coverage, it is unlikely to be further scheduled in subsequent stages, thereby missing the opportunity to quickly populate these high-potential communities. Moreover, even if a mutated seed does provide a new coverage, the low frequency at which new seeds are generated remains insufficient for populating community rapidly. SyzDiversity-V_3 addresses this problem via dynamically adjusting the mutation strategy for communities with different levels of CPR. As a result, V_3 significantly improves the coverage of V_2, with substantial increase of 15.4% and 10.8% observed around the 3th and 22nd hour, and also consistently outperforms other ablated variants in terms of cumulative bugs. This demonstrates that V_3 can quickly populate high-potential communities in the early phase of fuzzing (*i.e.,* 4th hour), and continue to be effective in later phases (*i.e.,* 22nd hour). For the warm-up comparison, SyzDiversity-V_3^R exhibits a certain performance loss relative to SyzDiversity-V_3 during the first five hours, while its performance gradually improves in the later stage. This is because random scheduling may select seeds with lower CPR, and the low-frequency mutation strategy further slows down the early generation of diverse seeds (i.e, Algorithm 2, Lines 3-7). Therefore, random warm-up weakens the early performance advantage. Interestingly, once the scheduling switches to CPR-aware scheduling and sufficient historical information has been accumulated, SyzDiversity can again explore high-CPR communities more efficiently and apply high-frequency mutation, so its overall performance can still improve to a certain extent.

Besides, as shown in Fig. 7, the coverage of SyzDiversity-V_4 is even lower than that of SyzDiversity-V_1 in the early stage, and only starts to improve in the middle and later stages. Similarly, the cumulative number of discovered bugs of SyzDiversity-V_4 remains below that of SyzDiversity-V_1 over the full fuzzing period. Nevertheless, there is still a large gap between SyzDiversity-V_4 and SyzDiversity-V_3. This is because, the low-density metric cannot effectively reflect seed diversity.



(a) $Cost_s$ (b) k (c) α (d) $threshold_r$ (e) $threshold_C$

Fig. 8. Results of our parameter sensitivity analysis

Without the guidance of CPR, SyzDiversity lacks the ability to focus on high-potential communities and may become trapped in local optima similar to other state-of-the-art fuzzers.

6.6 Parameter Sensitivity (RQ5)

SyzDiversity has five configurable parameters, namely the edit cost of AST ($Cost_s$), number of nearest neighbors in the similarity matrix (k), CPR weighting factor (α), CPR rank threshold for smash mutation ($threshold_r$), and minimum threshold for community creation ($threshold_C$), with default values of 5, 200, 7, 2, and 0.4. To evaluate their impact on coverage, we conducted 24-hour fuzzing for each parameter, varying one while keeping the others fixed.

Fig. 8 presents our sensitivity analysis results. In module ①, a high syscall edit cost over-penalizes structural changes by focusing only on syscall names, while a low edit cost adds noise by blurring behavioral distinctions. The parameter k determines the sparsity of the k-NN graph for seed partitioning; *i.e.,* large k values reduce discriminability by making the graph too dense, while small k values hinder semantic aggregation by making the graph too sparse. Both extremes weaken partition cohesion, and degrade SyzDiversity's performance in CPR computation, scheduling, and mutation. Specifically, $Cost_s = 5$ and $k = 200$ provide the best trade-off between structure and semantics.

In module ②, coverage shows a clear turning point when the CPR weighting factor α is set to 7. A smaller α reduces CPR's influence, causing the scheduling to favor immediate local coverage and possibly neglect high-CPR communities. In contrast, a larger α makes the scheduling overly dependent on CPR, leading to the neglect of communities with high overall coverage and those that have not been scheduled for a long time. Empirically, 7 is the best choice for α.

In module ③, SyzDiversity achieves the optimal coverage when $threshold_r = 2$ and $threshold_C = 0.4$. This allows aggressive smash mutation in the top 2 communities, whereas 0.4 serves as the best threshold for effective community creation.

7 Conclusions

We propose a diversity-guided Linux kernel fuzzer, SyzDiversity, which advocates for sustained construction and generation of sufficiently diverse seeds. aimed at effectively exploring boundary conditions across kernel subsystems and uncovering deep kernel vulnerabilities. Our evaluation demonstrates that SyzDiversity outperforms state-of-the-art kernel fuzzers in terms of code coverage and bug-finding capability.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 62332005 and 62372114).

References

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47 (2002), 235–256.

[2] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. 2002. The Nonstochastic Multiarmed Bandit Problem. *SIAM J. Comput.* 32 (2002), 48–77.

[3] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008 (2008), P10008.

[4] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[6] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2023. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In *Network and Distributed System Security Symposium*.

[7] Long Chen, Yunqing Zhang, Bin Tian, Yunfeng Ai, Dongpu Cao, and Fei-Yue Wang. 2022. Parallel Driving OS: A Ubiquitous Operating System for Autonomous Driving in CPSS. *IEEE Transactions on Intelligent Vehicles* 7, 4 (2022), 886–895.

[8] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *2024 IEEE Symposium on Security and Privacy*.

[9] Mauro Conti, Denis Donadel, and Federico Turrin. 2021. A Survey on Industrial Control System Testbeds and Datasets for Security Research. *IEEE Communications Surveys & Tutorials* 23, 4 (2021), 2248–2294.

[10] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[11] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407.

[12] D.Jones. 2015. Linux system call fuzzer. https://github.com/kernelslacker/trinity.

[13] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. 2023. {ACTOR}:{Action-Guided} Kernel Fuzzing. In *32nd USENIX Security Symposium*. 5003–5020.

[14] Google. 2025. Syzbot. https://syzkaller.appspot.com/upstream.

[15] Google. 2025. Syzkaller - Program Syntax. https://github.com/google/syzkaller/blob/master/docs/program_syntax.md.

[16] Google. 2025. Syzkaller - Syscall Descriptions Syntax. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[17] Google. 2025. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[18] NCC Group. 2017. FL/QEMU Fuzzing with Full-system Emulation. https://github.com/nccgroup/TriforceAFL.

[19] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. 2023. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *2023 IEEE Symposium on Security and Privacy*. 3262–3278.

[20] Kun Hu, Qicai Chen, Zilong Lu, Wenzhuo Zhang, Bihuan Chen, You Lu, Haowen Jiang, Bingkun Sun, Xin Peng, and Wenyun Zhao. 2025. A Survey of Fuzzing Open-Source Operating Systems. *arXiv preprint arXiv:2502.13163* (2025).

[21] Hsin-Wei Hung and Ardalan Amiri Sani. 2024. BRF: Fuzzing the eBPF Runtime. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1152–1171.

[22] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy*. 754–768.

[23] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. SEGFUZZ: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing. In *2023 IEEE Symposium on Security and Privacy*. 2104–2121.

[24] Muhui Jiang, Jinan Jiang, Tao Wu, Zuchao Ma, Xiapu Luo, and Yajin Zhou. 2024. Understanding Vulnerability Inducing Commits of the Linux Kernel. *ACM Trans. Softw. Eng. Methodol.* 33, 7 (2024), 170.

[25] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security Symposium 2022*.

[26] Linux kernel document. 2019. Kernel AddressSanitizer. https://github.com/google/kasan/wiki.

[27] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*.

[28] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2020. Finding bugs in file systems with an extensible fuzzing framework. *ACM Transactions on Storage* 16, 2 (2020), 1–35.

[29] lcamtuf. 2013. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[30] Myun Sub Lee, Sooyoung Cha, and Hakjoo Oh. 2023. Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), 384–396.

[31] Hongliang Liang, Yixiu Chen, Zhuosi Xie, and Zhiyi Liang. 2020. X-afl: A kernel fuzzer combining passive and active fuzzing. In *Proceedings of the 13th European workshop on Systems Security*. 13–18.

[32] Wenqing Liu and An-I Andy Wang. 2023. LFuzz: Exploiting Locality-Enabled Techniques for File-System Fuzzing. In *European Symposium on Research in Computer Security*. 507–525.

[33] Thouraya Louati, Heithem Abbes, and Christophe Cérin. 2018. LXCloudFT: Towards high availability, fault tolerant Cloud system based Linux Containers. *J. Parallel and Distrib. Comput.* 122 (2018), 51–69.

[34] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. 2024. MONARCH: a fuzzing framework for distributed file systems. In *Proceedings of the 2024 USENIX Conference on Usenix Annual Technical Conference*. 529–543.

[35] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies*.

[36] Hector D Menendez and David Clark. 2021. Hashing fuzzing: introducing input diversity to improve crash detection. *IEEE Transactions on Software Engineering* 48, 9 (2021), 3540–3553.

[37] Hector D. Menendez and David Clark. 2022. Hashing Fuzzing: Introducing Input Diversity to Improve Crash Detection. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3540–3553.

[38] Morteza Noferesti and Naser Ezzati-Jivan. 2024. Enhancing empirical software performance engineering research with kernel-level events: A comprehensive system tracing approach. *Journal of Systems and Software* 216 (2024), 112117.

[39] Palash B. Oswal. 2025. Linux Kernel Enriched Corpus. https://github.com/cmu-pasta/linux-kernel-enriched-corpus.

[40] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium*. 729–743.

[41] LLVM Project. 2018. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html

[42] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, and Zhenyu Chen. 2023. DiPri: Distance-Based Seed Prioritization for Greybox Fuzzing (Registered Report).

[43] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1852–1867.

[44] Juan Ramón Rico-Juan and Luisa Micó. 2003. Comparison of AESA and LAESA search algorithms using string and tree-edit-distances. *Pattern Recognit. Lett.* 24 (2003), 1417–1426.

[45] Daniel J. Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. 2018. A Tutorial on Thompson Sampling. *Found. Trends Mach. Learn.* 11 (2018), 1–96.

[46] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. 2018. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 587–600.

[47] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. 2021. Rtkaller: State-aware task generation for RTOS fuzzing. *ACM Transactions on Embedded Computing Systems* 20, 5s (2021), 1–22.

[48] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. 2022. Tardis: Coverage-guided embedded operating system fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4563–4574.

[49] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. Industry practice of coverage-guided enterprise linux kernel fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 986–995.

[50] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. {KSG}: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference*. 351–366.

[51] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 344–358.

[52] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1630–1644.

[53] Linus Torvalds. 2025. Linux Kernel Mailing List. https://lkml.org/.

[54] Dmitry Vyukov. 2018. kernel: add kcov code coverage. https://lwn.net/Articles/671640/.

[55] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium*. 2741–2758.

[56] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013).

[57] Michael Wunder, Michael L. Littman, and Monica Babes-Vroman. 2010. Classes of Multiagent Q-learning Dynamics with epsilon-greedy Exploration. In *International Conference on Machine Learning*. 1167–1174.

[58] Jiacheng Xu, Shihao Jiang, He Sun, Qinying Wang, Mingming Zhang, Xiang Li, Kaiwen Shen, Charles Zhang, Shouling Ji, Peng Cheng, and Jiming Chen. 2025. A Survey of Operating System Kernel Fuzzing. *ACM Trans. Softw. Eng. Methodol.* (2025).

[59] Jiacheng Xu, Xuhong Zhang, Shouling Ji, Yuan Tian, Binbin Zhao, Qinying Wang, Peng Cheng, and Jiming Chen. 2024. Mock: optimizing kernel fuzzing mutation with context-aware dependency. In *Proceedings of the Network and Distributed System Security Symposium*.

[60] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy*. 1643–1660.

[61] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy*. 818–834.

[62] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. *arXiv preprint arXiv:2401.00563* (2023).

[63] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *USENIX Security Symposium*.

[64] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18 (1989), 1245–1262.

[65] Qiang Zhang, Yuheng Shen, Jianzhong Liu, Yiru Xu, Heyuan Shi, Yu Jiang, and Wanli Chang. 2025. SUNFLOWER: Enhancing Linux Kernel Fuzzing via Exploit-Driven Seed Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[66] Zhiyu Zhang, Longxing Li, Ruigang Liang, and Kai Chen. 2025. Unlocking Low Frequency Syscalls in Kernel Fuzzing with Dependency-based RAG. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[67] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. {StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing. In *31st USENIX Security Symposium*. 3273–3289.

Received 2026-01-29; accepted 2026-06-25