Artifact Evaluated NDSS
Available
Functional
Reproduced

SYSYPHUZZ: the Pressure of More Coverage

Zezhong Ren[‡§], Han Zheng[§], Zhiyao Feng[§], Qinying Wang[§], Marcel Busch[§]
Yuqing Zhang[‡✉], Chao Zhang[†], Mathias Payer[§]
[‡]University of Chinese Academy of Sciences [§]EPFL [†]Tsinghua University

***Abstract*—Kernel fuzzing effectively uncovers vulnerabilities. While existing kernel fuzzers primarily focus on maximizing code coverage, coverage alone does not guarantee thorough exploration. Moreover, existing fuzzers, aimed at maximizing coverage, have plateaued. This pressing situation highlights the need for a new direction: code frequency-oriented kernel fuzzing. However, increasing the exploration of low-frequency kernel code faces two key challenges: (1) Resource constraints make it hard to schedule sufficient tasks for low-frequency regions without causing task explosion. (2) Random mutations often break context dependencies of syscalls targeting low-frequency regions, reducing the effectiveness of fuzzing.**

In our paper, we first perform a fine-grained study of imbalanced code coverage by evaluating Syzkaller in the Linux kernel and, as a response, propose SYSYPHUZZ, a kernel fuzzer designed to boost exploration of under-tested code regions. SYSYPHUZZ introduces *Selective Task Scheduling* to dynamically prioritize and manage exploration tasks, avoiding task explosion. It also employs *Context-Preserving Mutation* strategy to reduce the risk of disrupting important execution contexts. We evaluate SYSYPHUZZ against the state-of-the-art (SOTA) kernel fuzzers, Syzkaller and SyzGPT. Our results show that SYSYPHUZZ significantly reduces the number of under-explored code regions and discovers 31 unique bugs missed by Syzkaller and 27 bugs missed by SyzGPT. Moreover, SYSYPHUZZ finds five bugs missed by Syzbot, which continuously runs on hundreds of virtual machines, demonstrating SYSYPHUZZ's effectiveness. To evaluate SYSYPHUZZ's enhancement to SOTA fuzzers, we integrate it with SyzGPT, yielding SyzGPTsysy, which finds 33% more exclusive bugs, highlighting SYSYPHUZZ' potential. All discovered vulnerabilities have been responsibly disclosed to the Linux maintainers. We release the source code of SYSYPHUZZ at https://github.com/HexHive/Sysyphuzz and are trying to upstream it to Syzkaller.

I. INTRODUCTION

The operating system remains a foundational and highly security-sensitive element of modern computing. Its responsibilities in resource management and enforcing isolation between applications make it an attractive target for attackers aiming to compromise its guarantees. Given the importance of OS security, industry [1], [2], [3] and academic researchers [4], [5], [6], [7], [8], [9], [10], [11], [12] are actively looking for effective testing techniques. Among these techniques, fuzzing is unarguably the most promising approach for discovering kernel bugs.

✉Corresponding author. Email: zhangyq@ucas.ac.cn

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230921
www.ndss-symposium.org

Code coverage is a key success metric for kernel fuzzing and has been subject to several academic optimization efforts [13], [14], [10], [15]. Meanwhile, simply replaying the Syzbot corpus [2], accumulated through years of continuous fuzzing, enables kernel fuzzers to reach over 90% of their total coverage attainable during the fuzzing campaign [16], [17]. Building on this solid foundation of mature seed corpus and continuous optimization techniques, *kernel fuzzers' coverage plateaued* [18]. Additionally, *high coverage does not directly indicate sufficient exploration*. Often, these metrics fail to represent testing distributions throughout various kernel regions, potentially overlooking vulnerabilities hidden within less-explored regions. These two observations raise the need for additional optimization directions that focus on low-frequency areas to drive further bug discovery.

However, increasing exploration in the low-frequency areas of the Linux kernel, to boost hit frequency, remains a significant challenge. While hit frequency has been extensively studied in user-space fuzzing [19], [20], [21], [22], these solutions do not directly scale to kernel fuzzing. We outline two key challenges that limit improvements in hit frequency, with a specific focus on the Linux kernel.

Challenge 1. Targeted Exploration of Low-Frequency Regions Under Resource Constraints. A key challenge in kernel fuzzing is exploring low-frequency regions without overwhelming the fuzzer's task scheduler. While expanding code coverage remains a primary goal, naively injecting large numbers of exploration tasks targeting enormous-scale low-frequency regions may lead to task explosion, where the fuzzer is overloaded with too many pending tasks. This is particularly problematic in kernel fuzzing, where each test case incurs significant execution overhead. This overload of the execution queue can delay or even suppress other valuable tasks, ultimately reducing the effectiveness of fuzzing. Additionally, low-frequency regions are not static; the hit count of basic blocks (BBs) evolves as fuzzing progresses. Blindly triggering new tasks for low-frequency regions without considering the current coverage state may waste resources or duplicate effort. Therefore, effectively exploring low-frequency regions requires an adaptive mechanism that tracks execution frequency, determines when sufficient baseline coverage has been achieved, and selectively schedules new tasks to rebalance attention toward low-frequency regions.

Challenge 2. Context-Aware Mutation for Dependency-Preserving Fuzzing. Kernel fuzzers use sequences of system calls (syscalls) as test inputs. Unlike user-space programs, these syscall sequences often involve complex contextual dependencies. For example, a file descriptor returned by one syscall (*e.g.,* `open`) can be used in subsequent calls (*e.g.,* `read`, `write`). Similar dependencies exist for sockets and memory-mapping operations. When test cases are randomly mutated, these dependencies may inadvertently be broken, rendering the inputs semantically invalid or functionally meaningless. This presents a critical challenge when guiding fuzzing toward low-frequency regions. Due to the large and dynamic set of target regions, the mutator must determine syscall dependencies, which can vary across different target regions. Hence, the fuzzer must go beyond traditional mutation strategies and adopt context-aware mutation techniques that preserve the execution semantics of syscall sequences. This ensures that the necessary sub-sequences, serving as pre-contexts for the test cases, are preserved, enabling more reliable exploration of rarely executed code.

Proposed Solution. In our paper, we first study the nature of unbalanced hit frequency by evaluating Syzkaller on the Linux kernel and in response propose SYSYPHUZZ, a kernel fuzzer designed to enhance the exploration of overlooked code regions. SYSYPHUZZ builds upon Syzkaller's task-oriented fuzzing model, where a scheduled task sequence drives testing. During task execution, test cases are generated through carefully designed generation or mutation strategies. SYSYPHUZZ introduces a new task type, *the boost task*, along with a *Boost Delegator* responsible for task scheduling. The *Boost Delegator* schedules boost tasks based on runtime feedback. Each boost task performs pre-mutation checking to keep the task updated with the latest target regions and applies *Context-Preserving Mutation* to maintain execution semantics during targeted exploration.

To address the first challenge, SYSYPHUZZ adds boost tasks only after achieving a sufficient level of baseline coverage obtained by replaying the Syzbot corpus. Then, it introduces a *Selective Task Scheduling* strategy within the *Boost Delegator* to control when and how boost tasks are enqueued. To avoid overloading the task queue, SYSYPHUZZ continuously tracks BB hit counts, maps them to their corresponding seeds, and dynamically enqueues a small portion of boost tasks targeting under-explored regions. To mitigate task explosion, the *Boost Delegator* generally prioritizes executing boost tasks and schedules the coverage-guided tasks only when new coverage is observed. However, when a boost task is triggered, some regions may no longer qualify as low-frequency or may prove to be unreachable, yielding diminishing returns and potentially wasting resources. Thus, the *Boost Delegator* rechecks whether the associated task still targets a valid low-frequency region before applying boost mutations. Additionally, SYSYPHUZZ maintains a *denylist* of BBs that remain consistently inaccessible despite repeated attempts, further reducing unnecessary effort.

To address the second challenge, SYSYPHUZZ introduces *Context-Preserving Mutation* in the boost tasks, which identify syscalls associated with low-frequency regions and selectively mutate them to reduce the risk of breaking syscall dependencies. Specifically, SYSYPHUZZ introduces and maintains a mapping between syscalls and the BBs they cover, which enables fundamentally more fine-grained mutation strategies. During mutation, SYSYPHUZZ only focuses on mutating syscalls related to low-frequency regions, while preserving those necessary for maintaining the execution context. When targeting the mutation of the syscall sequence, SYSYPHUZZ keeps the necessary execution context unchanged to preserve reachability to the targeted regions. When focusing on a specific syscall, only the identified syscall is modified, either through argument mutation or by actively introducing runtime faults to check the error-handling codes.

We evaluate SYSYPHUZZ against state-of-the-art kernel fuzzers including Syzkaller [3] and SyzGPT [23]. Our results demonstrate that SYSYPHUZZ increases coverage of low-frequency regions by 2X, and reduces the proportion of surviving overlooked low-frequency regions that are initially underexplored and remain so after 72 hours of fuzzing campaigns by 12.7%. Furthermore, SYSYPHUZZ discovers 31% more unique bugs than Syzkaller (67 vs. 51), and 10% more bugs than SyzGPT (67 vs. 61). Among these, 20 bugs are exclusively found by SYSYPHUZZ. To understand the nature of these exclusive bugs, we analyze their correlation with low-frequency regions identified by SYSYPHUZZ. Almost twice as many bugs found exclusively by SYSYPHUZZ are located in low-frequency regions in comparison to those bugs exclusively found by Syzkaller, which underscores our approach's effectiveness in targeting low-frequency kernel regions. Additionally, SYSYPHUZZ identified five unique bugs that were missed even by Syzbot [2], a large-scale continuous fuzzing service operated by Google that runs hundreds of virtual machines (VMs) [24]. To evaluate whether SYSYPHUZZ's strategy can enhance the effectiveness of other kernel fuzzers, we integrate SYSYPHUZZ into SyzGPT (named SyzGPTsysy). We then compare SyzGPTsysy against the original SyzGPT using a saturated corpus. Overall, SyzGPTsysy discovers 16 exclusive bugs—33% more than the 12 exclusive bugs found by SyzGPT—demonstrating SYSYPHUZZ's potential to augment other fuzzers in uncovering previously overlooked vulnerabilities. All discovered vulnerabilities have been responsibly disclosed to the Linux kernel maintainers.

To sum up, we make the following contributions:

- We study the nature of unbalanced code coverage by evaluating Syzkaller targeting the Linux kernel, and identify low-frequency regions and context-destroying mutations as key limitations of existing kernel fuzzing campaigns.
- We develop SYSYPHUZZ, equipped with a *Boost Delegator* and *Context-Preserving Mutation* to address the challenges of developing a frequency-balanced kernel fuzzer.
- We conduct a thorough evaluation to demonstrate that SYSYPHUZZ boosts the hit frequency of underexplored regions and improves the kernel fuzzer's capability to discover overlooked bugs.
- We release SYSYPHUZZ at https://github.com/HexHive/S

ysyphuzz to support open science and work with Google to upstream it into Syzkaller.

II. Background

A. Linux Kernel Fuzzing

The kernel takes a sequence of system calls (syscalls) from user-space programs as input and returns corresponding resources as output. This interface allows adversaries to send crafted inputs aimed at compromising the kernel's integrity and confidentiality guarantees, potentially leading to an escalation of privileges or information leakage [25], [26], [27], [28]. Therefore, recent kernel fuzzers primarily target the syscall interface by generating valid syscall sequences to simulate the program behavior [29], [30], [31], [30], [14], [15], [17], [15], [10], [12], [11]. For instance, the kernel fuzzer may generate syscalls, including `socket_inet_tcp`, `bind_inet`, and `listen`. This sequence simulates a network communication program that starts a TCP connection. First, the `socket_inet_tcp` call creates a TCP socket. Then, `bind_inet` associates it with a specific address and port. Finally, `listen` sets the socket to listen for incoming connections.

B. Syzkaller and Syzbot

Syzkaller [3] has become the de facto standard for Linux kernel fuzzing and is being deployed on hundreds of VMs [2]. In contrast to user-space fuzzers such as AFL [32], Syzkaller adopts a task-oriented design, primarily including four types of tasks: **Generation**, **Mutation**, **Smash**, and **Triage**. These tasks generate test cases consisting of sequences of syscalls along with their arguments. The Syzkaller executor then runs these test cases by invoking the corresponding syscalls.

Generation. Syzkaller creates new test cases from scratch using a set of manually curated templates developed by domain experts (*e.g.,* kernel developers). These templates encode the argument types for each syscall and capture dependencies between them (*e.g.,* the return value of `open` being used by `read`). This allows Syzkaller to generate semantically meaningful syscall sequences and arguments, increasing the likelihood of triggering deep kernel logic.

Mutation. Syzkaller randomly selects a seed from the corpus, which consists of test cases that have previously triggered new coverage, and performs a series of random mutations. These mutations may involve inserting or removing syscalls, or modifying the arguments of existing calls, guided by built-in syscall templates.

Triage. When a generated or mutated test case triggers new coverage, Syzkaller performs a triage step. First, it conducts verification to ensure the coverage is reproducible and not affected by kernel state, nondeterminism, or concurrency. Then, it performs minimization to simplify the test case while preserving the newly found coverage. If a minimized variant triggers new coverage, it is marked for further triage. Successfully triaged test cases are added to the seed corpus for future fuzzing, and corresponding Smash tasks are generated and enqueued in the work queue.

Smash. Smash tasks aggressively mutate new seeds that have just been added to the corpus (after passing triage). Each such seed is mutated repeatedly for a fixed number of iterations (typically 100). This stage is designed to maximize the discovery of bugs in these coverage-introducing seeds.

Task Scheduling. By default, Syzkaller selects among the four primary fuzzing activities according to a fixed priority order. Triage tasks are given the highest priority and are executed first. If no triage task is available, Syzkaller invokes the Smash tasks. Smash tasks are polled intermittently based on a configurable skip interval. When both triage and smash queues are exhausted, Syzkaller performs either mutation or generation based on a probabilistic ratio (99% mutation + 1% generation).

Syzbot [2] is a Google service that runs continuous, large-scale kernel fuzzing across diverse configurations and architectures. It discovered and reported over 17,000 unique bugs, more than 6,000 of which are fixed [33]. During its fuzzing campaign, Syzbot collects a coverage-rich corpus, allowing the kernel fuzzer to start with a reasonable initial coverage [17], [16].

C. Imbalanced Coverage Problem

Fuzzing leverages random mutations to explore different program paths. While this strategy has proven effective for uncovering diverse behaviors, it often leads to coverage imbalance—a situation where certain regions of the code are exercised repeatedly, while others receive little execution after being covered. This issue has been well-documented when fuzzing user-space applications [20], yet it remains relatively underexplored in the domain of kernel-level fuzzing. In the case of complex targets like the Linux kernel, the problem can be even more pronounced due to its intricate control flows and extensive code base. As a result, some critical code paths may remain insufficiently tested, limiting the effectiveness of fuzzing in exposing deep or rare vulnerabilities.

Imbalanced Coverage in the Linux Kernel. The inherent complexity of the Linux kernel poses significant challenges to achieving balanced code coverage during fuzzing. With over 40 million lines of code [34], high cost of each execution, and limited testing resources, exhaustively exercising all possible execution paths becomes highly impractical, if not infeasible.

Moreover, kernel fuzzing introduces additional challenges due to the structured nature of its input. Unlike user-space fuzzers, which often operate on raw binary input, kernel fuzzers interact with the system through sequences of syscalls. These syscalls often exhibit complex dependencies—violating any dependency during mutation may prematurely trigger error-handling paths, hampering the exploration of functional logic. Furthermore, even carefully crafted syscall sequences that successfully reach deep kernel components (*e.g.,* networking protocols) can be easily disrupted by random mutations. As a result, these valuable code paths may be executed infrequently, further exacerbating the coverage imbalance.

Low-Frequency Region. To the best of our knowledge, there is no standardized definition for low-frequency regions in the context of fuzzing. Our straightforward approach ranks code regions based on their execution frequency and selects a fixed proportion, typically a small percentage of the least frequently executed code, as the threshold. This threshold is treated as a hyperparameter, which may vary depending on the characteristics of the target program.

In this work, we define low-frequency regions based on the hit count of BBs. Specifically, we study Syzkaller, from which we extract the execution frequency of kernel BBs. The resulting distribution is shown in Figure 2. Based on our observation in Section III, a *5% threshold* serves as a reasonable approximation, and we adopt it to define the boundary of low-frequency regions in our analysis and implement our prototype SYSYPHUZZ.

III. COVERAGE FREQUENCY IN THE KERNEL

In this section, we explore the *imbalanced coverage problem* in the Linux kernel fuzzing by studying the hit count distribution of code regions. Using a case study of Syzkaller, we first analyze the hit count of covered basic blocks (BBs) in Section III-A. Then, we examine the proportion of BBs that remain low frequency throughout the whole campaign (Section III-B). Finally, we revisit a past bug discussed in Section III-C, which Syzkaller failed to detect for six months, despite achieving high coverage, until it was later reported by directed fuzzing, which restricts fuzzing to focus on the file system. This case exemplifies how high overall coverage can still miss bugs residing in persistent low-frequency regions.

Benchmark and Corpus. We use the Linux kernel v6.12-rc6 (released on November 04, 2024) in our study, and import Syzbot's [2] corpus from November 13, 2024 [35].

Experimental Setup. All experiments were conducted on a server equipped with an Intel Xeon Gold 5218 processor and 64 GB of RAM. Each fuzzer was executed on 8 virtual machines (VMs), and each VM was assigned 2 CPU cores and 4 GB of memory. Each experiment consisted of a 72-hour fuzzing campaign, preceded by the corpus replay phase to warm up the fuzzing process (this phase was completed within one hour). Each experiment was repeated five times to minimize variance. Additionally, we select Syzkaller (commit 9750182a9) [3], the most widely deployed Linux Kernel fuzzer, for this study [1], [2].

Code Frequency Sampling. The BB ID serves as the key to store each BB's hit count, *i.e.,* the number of times executed by Syzkaller. In a single execution, the BB may be hit multiple times. Additionally, we rank the BBs according to their hit counts and consider the BBs whose hit counts belong to the bottom 5% as low-frequency regions.

A. Basic Block Hit Count Analysis in Syzkaller

We aggregate the execution traces from Syzkaller into a hit count map to analyze the hit distribution over time. Specifically, we record the BB hit count at six time points: immediately after corpus replay (t_0), and at 6, 12, 24, 48, and 72 hours post-t_0. The resulting frequency distributions are depicted in Figure 1. The results show that throughout the

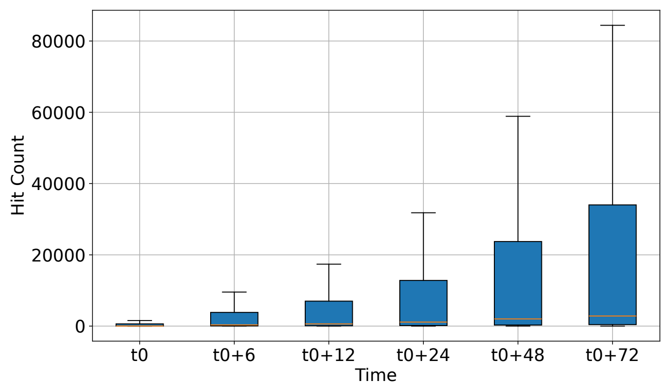


Fig. 1: Linux hit frequency of five trials of our 72 h-campaign. Syzkaller takes $t0$ time to finish corpus replay, $t0+Xh$ means X hours after $t0$. The y-axis is the BB hit count. The box spans the inter-quartile range (IQR), the horizontal orange line marks the median, and whiskers extend to 1.5 × IQR

entire fuzzing campaign, the average BB hit count remains low. In contrast, the maximum hit count exceeds 80.0K, revealing a significant imbalance in execution frequencies across BBs.

freq	t0	t0+6h	t0+12h	t0+24h	t0+48h	t0+72h
0%	1	1	1	1	1	1
5%	<10.0	<10.0	14.0	19.5	25.5	29.8
25%	<0.1K	<0.1K	<0.1K	0.2K	0.3K	0.4K
50%	<0.1K	0.3K	0.6K	1.1K	2.0K	2.8K
75%	0.6K	3.8K	7.0K	12.8K	23.7K	34.0K
100%	76.7M	0.6B	1.2B	2.5B	2.9B	3.6B
Num_{bb}	368.9K	403.9K	421.4K	446.4K	462.5K	470.9K

TABLE I: The hit frequency distribution during the 72-hour fuzzing campaign. The row at x% represents the frequency of the x%-th BB when sorting BBs by hit frequency in ascending order; K, M, and B stand for thousand, million, and billion.

Our proportion analysis in Table I further reveals that Syzkaller exhibits a pronounced imbalance in code coverage during the 72-hour fuzzing campaign. While the most frequently executed BB increases its hit count from 76 million at t_0 to 3.6 billion at t_0+72h, the least executed BBs are executed only once throughout the entire period. Moreover, even at the 72-hour mark, the bottom 5% of BBs had hit counts below 30, which is significantly lower than the median hit count of approximately 2,800.

To further characterize the coverage disparity, we analyze the cumulative distribution of BB execution counts over the course of the campaign, as illustrated in Figure 2. In the figure, we highlight the execution counts corresponding to the top and bottom 5% of covered code regions using dashed lines. Even after 72 hours of fuzzing, approximately 10% of the BBs have been executed fewer than 100 times. In contrast, the top 5% of BBs are executed more than one million times. This stark contrast highlights a substantial imbalance in execution

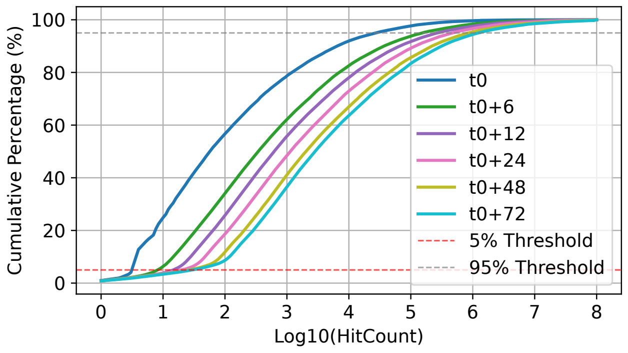


Fig. 2: Cumulative distribution function of coverage frequency (log scale) in Syzkaller campaign.

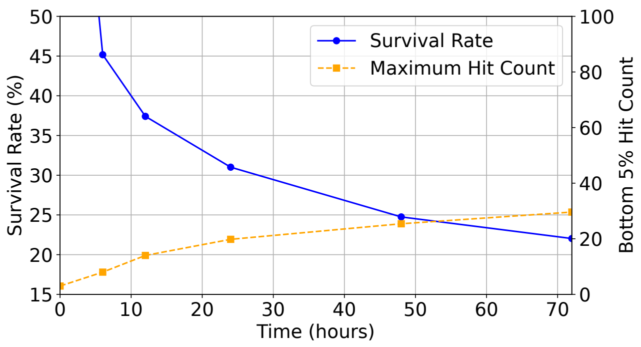


Fig. 3: Percentage and maximum hit count of low-frequency regions that remain low-frequency in 72-hour fuzzing campaign.

frequency across kernel code regions, indicating that a large portion of the code remains underexplored despite extended fuzzing.

Additionally, as shown in Figure 2, we observe that the 5th percentile of BB execution frequency closely aligns with the inflection point of the cumulative distribution function (CDF). This observation suggests that the bottom 5% of BBs, in terms of execution frequency, serves as a reasonable threshold for identifying low-frequency regions in kernel fuzzing. Thus, in subsequent analyses, we define low-frequency regions as those comprising the least frequently executed 5% of BBs.

Observation 1: Kernel fuzzers suffer from an imbalanced code frequency distribution. Some BBs are executed much fewer times than others.

B. *Low-Frequency Region Survival*

We further examine whether initially underexplored code regions remain insufficiently covered throughout the fuzzing campaign. Specifically, we define the bottom 5% of BBs at t_0 based on their hit count immediately after replaying the corpus as the initial low-frequency BBs, and the percentage of these initial low-frequency BBs that remain within the bottom 5% at each point in time (*i.e.,* t_0, t_0+6, $\dots$) is called the survival rate. Figure 3 summarizes our findings.

Within the first 6 hours, 54.8% of these low-frequency BBs receive enough executions and no longer fall within the bottom 5%. However, the pace of improvement slows considerably: from 6 to 12 hours, only an additional 7.8% are better explored. In the subsequent intervals, *i.e.,* 12 to 24h, 24 to 48h, and 48 to 72h, coverage improvements further diminish, with only 6.4%, 6.3%, and 2.7% of BBs, respectively, exiting the low-frequency set. Notably, even after 72 hours, 22.0% of the initially underexplored BBs remain within the bottom 5%, *i.e.,* being visited less than 30 times in the whole campaign, indicating persistent gaps in exploration. Moreover, BBs that persist in the bottom 5% continue to receive insufficient coverage, limiting effective bug discovery. For instance, the maximum hit count for these surviving BBs increases from 8.0 at 6 hours to 29.6 at 72 hours, while the average rises from 4.0 to 11.4, and the median from 3.6 to 9.6, respectively.

Observation 2: Over 22% of the initially underexplored BBs remain insufficiently explored after 72 hours of kernel fuzzing.

C. *Case Study for the Overlooked Low-Frequency Regions*

To further illustrate that bugs hide in the underexplored kernel code, we conduct a case study on a representative bug discovered in `bch2_trans_node_iter_init`, located in the Bcachefs file system, a modern copy-on-write (COW) file system for Linux, officially merged into the kernel in version 6.7 (released January 2024). The affected source file, `btree_iter.c`, had already been covered back in May 2024 (the earliest available coverage record) and showed 62.0% code coverage. Yet the bug was not reported until six months later by a customized Syzkaller build focused on file system testing.

In the Bcachefs file system, 1,340 BBs remain within the low-frequency region even after 72 hours of fuzzing following corpus replay. Among them, 73.7% exhibit a hit count of 11 or fewer, matching the average hit count for low-frequency regions as identified in Section III-B. Focusing on bug-related source files, `btree_iter.c` and `btree_iter.h`, which handle the B-tree traversal, we observe 95 low-frequency BBs in total, including 55 in `btree_iter.h`. Of these, 72.6% also have a hit count of 11 or fewer.

These low-frequency regions remain insufficiently tested due to stringent preconditions, particularly those associated with complex kernel logic such as B-tree manipulation. A representative example is the function `bch2_trans_revalidate_updates_in_node`, which is responsible for maintaining transactional consistency by validating pending updates within a B-tree node. We identify 7 BBs within this function that fall into the low-frequency region, each with a hit count no greater than 10.

We further observed that this function requires a precondition where a sufficient number of *insertion* and *modifica-*

tion operations have already been performed. Satisfying this condition involves issuing multiple, often repetitive, system calls that modify files within the Bcachefs file system. Because these system call sequences tend to be highly similar, they offer limited potential for additional coverage, causing Syzkaller to prioritize generating new test cases in other areas. As a result, the function remains underexplored in standard fuzzing campaigns. It was not until the development of a customized version of Syzkaller, specifically designed for file systems, that targeted generation of file operation test cases began to yield improved coverage in these regions, and finally reported this bug. This bug is not a unique case since we found another bug that occurred during network encryption, involving a use-after-free of the `sk_buff` structure within the function `scatterwalk_copychunks`, invoked by `bch2_encrypt` in the Bcachefs file system. This bug also goes through several low-frequency BBs and was reported by the same customized version of Syzkaller at the end of May 2024. These two cases suggest that low-frequency regions may conceal overlooked bugs, and motivate deeper investigation into the correlation between coverage frequency and bug prevalence, which we statistically analyze in Section Section VI-C

Observation 3: Overlooked and underexplored kernel regions *may* hide bugs, and limited coverage from random test cases is insufficient to expose them.

IV. SYSYPHUZZ DESIGN

Based on the observations in Section III, we propose the design of SYSYPHUZZ, which explores low-frequency regions by introducing novel boost tasks. The workflow of SYSYPHUZZ is illustrated in Figure 4. SYSYPHUZZ operates in two stages: corpus replay and targeted fuzzing via *Selective Task Scheduling*. In the first stage, SYSYPHUZZ replays the entire Syzbot corpus to establish a comprehensive baseline of code coverage. This stage ensures that the kernel fuzzer inherits the cumulative coverage achieved over years of CPU-intensive fuzzing campaigns. In the second stage, SYSYPHUZZ utilizes the boost delegator to allocate fuzzing energy with a priority on boost tasks, which are specifically designed to target low-frequency regions. At the same time, SYSYPHUZZ incorporates coverage tasks such as Syzkaller's native coverage-guided Triage and Smash tasks to assist in discovering previously unexplored code paths.

Specifically, the workflow of stage 2 is as follows. The *Boost Delegator* identifies low-frequency regions based on execution statistics and allocates additional energy to them. For each boost task, SYSYPHUZZ conducts pre-mutation checking, and the result is fed back to the *Boost Delegator* as part of its scheduling strategy to determine whether further mutation is necessary. If so, SYSYPHUZZ leverages the pre-mutation information to apply *Context-Preserving Mutation*, maintaining the syscalls' context necessary to reach the targeted regions. When a mutated test case triggers new edge coverage in boost tasks, additional coverage tasks are scheduled to further extend exploration. SYSYPHUZZ reuses Syzkaller's existing coverage-related tasks, which perform triage before passing the input to Syzkaller's mutator for further fuzzing.

A. Corpus Replay

SYSYPHUZZ begins with the corpus replay stage. A key prerequisite for discovering bugs is reaching the corresponding buggy code, which requires the kernel fuzzer to maximize code coverage [31], [30], [29]. However, the high execution cost of kernel fuzzing limits the total number of tasks that can be executed during a fuzzing campaign. As a result, solely focusing on coverage can leave a significant portion of low-frequency regions underexplored. Fortunately, the Syzkaller community has accumulated a rich fuzzing dataset. We leverage the Syzbot [2] corpus, which originates from over seven years of continuous fuzzing efforts. This allows SYSYPHUZZ to start with a reasonable initial coverage baseline, enabling it to dedicate more resources to boost tasks.

Specifically, we follow Syzkaller's approach by replaying the generated corpus while omitting time-consuming tasks to maintain efficiency. Our observations show that corpus replay alone achieves over 90% of the total code coverage observed during a full fuzzing campaign. Additionally, we record the hit count of each basic block (BB) during the replay process, which serves as the basis for identifying low-frequency regions.

B. Boost Delegator

After corpus replay, SYSYPHUZZ begins executing boost tasks, which are initiated by the *Boost Delegator*. The *Boost Delegator* identifies low-frequency regions at the BB level in real time, based on the hit counts of each BB. It then directs fuzzing efforts toward these underexplored regions, which are often overlooked by traditional coverage-guided fuzzing.

Hit Count Tracking. To track the hit count of each covered BB and select seeds that are more likely to explore low-frequency regions, SYSYPHUZZ maintains two maps in its seed corpus: one that stores seeds along with their associated BBs, and another that records the hit counts of these blocks. Figure 5 illustrates the structure of SYSYPHUZZ's seed corpus. Regarding the *seed map*, SYSYPHUZZ maps each seed to its covered BB IDs and updates the map when a new seed is generated. The *hit map* further stores the hit count for each covered BB and is updated in each execution.

Note that the seed corpus is initialized before the corpus replay stage. It is continuously updated during the replay, providing a starting point for identifying low-frequency regions and selecting seeds that cover BBs with low hit counts.

Hit Count Guided Task Enqueueing. Based on the seed corpus, SYSYPHUZZ employs a proportion-based selection strategy to choose seeds for further mutation. Specifically, SYSYPHUZZ sorts all covered BBs in ascending order by hit count and selects the bottom fraction—by default 5%—as low-frequency targets. This relative approach avoids relying on fixed thresholds (*e.g.,* BBs with hit count below 100), enabling

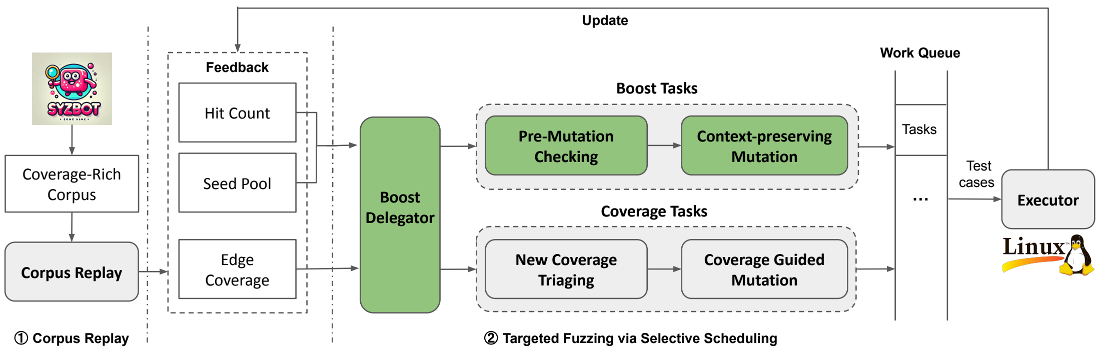


Fig. 4: SYSYPHUZZ consists of two stages: (1) corpus replay and (2) targeted fuzzing via selective scheduling.

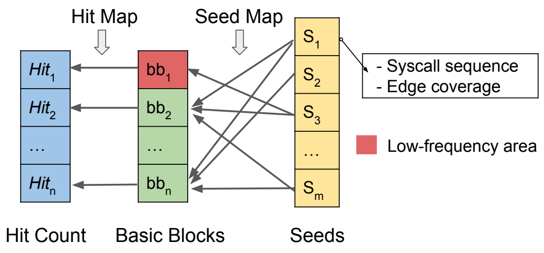


Fig. 5: Seed corpus of SYSYPHUZZ.

SYSYPHUZZ to consistently focus on the most overlooked regions during fuzzing. For instance, even if a BB's hit count increases from 1 to 100, it remains a low-frequency target as long as it stays among the least-covered BBs. This ensures that gradually explored but still under-covered regions continue to receive attention. By controlling the proportion size, SYSYPHUZZ limits the number of selected regions and ensures focus remains on code unlikely to be reached by standard fuzzing alone. If these regions are sufficiently explored by regular coverage tasks, no additional boost tasks are launched. Because hit counts evolve over time, the *Boost Delegator* periodically updates its target list to reflect current execution dynamics (*i.e.,* once per minute).

Based on the selected set of BBs, SYSYPHUZZ searches the seed map by iterating through each seed and comparing its covered BBs with the target set. If an overlap is detected, SYSYPHUZZ selects the corresponding seed and schedules a boost task to prioritize its execution.

Selective Task Scheduling. The *Boost Delegator* manages task scheduling by prioritizing boost tasks over general coverage tasks. Selected seeds for boosting are placed in a high-priority queue to ensure immediate task generation and execution. If new coverage is discovered during this process, the corresponding triage task is deferred to a secondary queue, allowing boost tasks to proceed without interruption. A third queue stores test cases generated by the mutator, including those from both boost and coverage tasks. The *Boost Delegator* monitors this queue, and once all test cases from the current boost round have been executed, it marks the round as complete—a cycle that includes identifying low-frequency regions, selecting seeds, generating test cases, and executing them. The Delegator then updates the target list and initiates a new round of boost tasks accordingly.

However, BB hit counts and seed coverage tend to be unstable and may vary across executions due to the non-deterministic nature of the kernel [36]. To address this, the *Boost Delegator* performs a pre-mutation validation step, where it replays each selected seed against the latest target list and filters out those that no longer cover any valid targets. During this step, it also identifies key syscalls whose execution overlaps with target BBs, enabling the fuzzer to guide mutation more precisely and retain meaningful execution context. As part of this validation process, SYSYPHUZZ also maintains a *deny list* to handle BBs that remain persistently uncovered. At the beginning of each boost round, the *Boost Delegator* records the hit counts of all selected BBs. If a BB shows no coverage improvement over 10 consecutive rounds, it is added to the *deny list* and temporarily excluded from future fuzzing rounds. Conversely, if its coverage improves later (*e.g.,* through an alternative path), it is removed and reinstated as a valid target.

C. *Context-Preserving Mutation*

The kernel fuzzer leverages mutations to vary exploration of the system calls. However, mutating kernel inputs is a non-trivial task. BBs in the target list are often missed during fuzzing because they are difficult to reach. This is largely due to execution paths that involve complex constraints, such as checksums or deep conditional logic, which are unlikely to be satisfied by random mutation alone. To overcome this, it is crucial to understand and preserve the necessary execution context and use the target BBs as entry points for further exploration. To this end, we propose a *Context-Preserving Mutation* strategy, which consists of two key steps: (1) determining the execution context, and (2) designing targeted mutation techniques.

Execution Context Determination. Since each system call can modify kernel state, test cases—comprising syscall sequences—must maintain the correct conditions to reach target BBs. During pre-mutation checking, the *Boost Delegator* identifies the key syscalls that lead to target BBs and builds a mapping (BB2Syscall Map) to track these dependencies. Because reaching a target BB depends on the kernel state established by prior syscalls, all calls before a relevant syscall are treated as its execution context. As shown in Figure 6, both sequences Sys_1 and Sys_1, Sys_2 can reach bb_1, while all syscalls before Sys_m form the context for bb_2. To ensure the test case's effectiveness, the mutator preserves this context when generating inputs for the target list, enabling more targeted and reliable exploration.

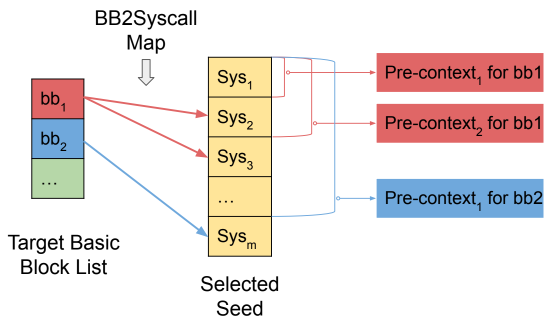


Fig. 6: The BB2Syscall map identifies the pre-context for each target BBs.

Mutator Designs. SYSYPHUZZ employs two distinct mutation strategies. The first strategy aims to explore code regions adjacent to targeted BBs, with the goal of uncovering new coverage or triggering latent bugs. This is achieved by either modifying the sequence of syscalls or altering variables involved in comparison instructions within key syscalls. When mutating syscall sequences, SYSYPHUZZ performs operations such as insertion, replacement, and deletion. To preserve the execution context required to reach the target BBs, key syscalls, along with their necessary pre-context, are marked as immutable during boost mutations. This approach ensures that the critical execution path remains intact, enabling the fuzzer to effectively explore neighboring logic without disrupting access to the target region.

The second strategy focuses on enhancing bug discovery by injecting faults into the pre-context of key syscalls, which is a default mutation strategy in Syzkaller. SYSYPHUZZ leverages Linux's built-in fault injection framework to simulate failures in various kernel subsystems. By inducing runtime faults, such as memory allocation failures, I/O errors, or abnormal control-flow conditions, SYSYPHUZZ is able to rigorously stress-test the kernel's error-handling mechanisms. This increases the likelihood of exposing bugs that are otherwise unlikely to manifest under normal execution conditions.

V. IMPLEMENTATION

We implement SYSYPHUZZ on top of Syzkaller. Our system includes the following components.

Hit Map. SYSYPHUZZ implements the *Hit Map* to support *boost tasks*. Specifically, SYSYPHUZZ first disables Syzkaller's default deduplication logic to preserve raw execution data across diverse task types and support accurate hit count collection. Furthermore, SYSYPHUZZ aggregates BB hit counts into a dedicated *hit map*, which is dynamically maintained using a copy-on-write strategy—similar to the one employed for the *deny list*—to ensure both consistency and low overhead. SYSYPHUZZ intentionally excludes the triage tasks to minimize the measurement noise.

Target List. SYSYPHUZZ implements a *target list* to record the low-frequency regions. By reading the current *hit map* and selecting BBs based on percentile thresholds, SYSYPHUZZ adds the least visited regions into the *target list*, which is further used for seed selection. For efficient seed selection, SYSYPHUZZ extends Syzkaller's in-memory corpus structure to map the BBs to corresponding seeds, *i.e.,* a BB-to-seed map (Figure 5). Subsequently, SYSYPHUZZ validates that the chosen seed can indeed cover the target BBs in the list.

Deny List. Not all BBs in the *target list* are reachable due to the indeterminism of kernel code execution. Thus, SYSYPHUZZ further extends a *deny list* to mark the covered BBs that remain persistently unreachable. During the *boost tasks*, if the BB continuously fails to be covered for a certain amount of time, we consider this BB related to non-deterministic kernel behavior and add it to the *deny list*. The *deny list* ensures that SYSYPHUZZ does not waste energy on non-reproducible low-frequency regions.

VI. EVALUATION

In this section, we study whether SYSYPHUZZ effectively boosts low-frequency regions and subsequently finds new bugs. Specifically, we evaluate SYSYPHUZZ to answer the following research questions:

1) RQ1: Can SYSYPHUZZ increase the hit counts of low-frequency kernel code?
2) RQ2: Can SYSYPHUZZ outperform SOTA kernel fuzzers?
3) RQ3: Is SYSYPHUZZ's bug discovery correlated with its focus on low-frequency regions?
4) RQ4: Can SYSYPHUZZ uncover bugs that Syzbot misses?
5) RQ5: How do SYSYPHUZZ's individual components contribute to its performance?
6) RQ6: Can SYSYPHUZZ improve other kernel fuzzers?

Compared Fuzzers. We evaluate SYSYPHUZZ against Syzkaller, the most widely used Linux kernel fuzzer [3], and SyzGPT [23], the latest state-of-the-art fuzzer. For RQ4, we compare the vulnerabilities discovered by SYSYPHUZZ over a 3-day period with those reported by Syzbot [2], the long-running Syzkaller instances deployed across hundreds of internal Google cluster nodes comprising nine years of kernel fuzzing [24].

Evaluation Setup. All experiments are conducted on a server running Ubuntu 22.04 equipped with a 16-core Xeon Gold 5218 and 64GB RAM. Each fuzzer instance was assigned 8 VMs, with each VM allocated two cores and 4GB of memory. Each evaluation trial was executed for a 3-day period after a corpus replay phase, whose duration depends on the number of generated inputs. This setup reflects real-world scenarios where replay is essential for triaging and validating generated seeds. All three fuzzers exhibit nearly identical corpus replay durations, with an average difference of no more than four minutes. The experiments described in the following sections aim to answer the research questions from above and are independently repeated five times to reduce statistical variance.

Initial Corpus. Regarding the initial corpus, both SYSYPHUZZ and Syzkaller use the same seed set (tag:ci-qemu-upstream-corpus-2024-11-13) retrieved from Syzbot [2]. As the SOTA SyzGPT enhances this corpus via LLM-assisted generation, we follow the guidelines and recommendations provided in the corresponding GitHub repository [37]. Specifically, we adopt GPT-3.5 Turbo as the LLM and use the Syzbot corpus as the base for seed generation. In line with SyzGPT, we reuse the syscall dependency data released by the authors, covering all system calls in the Linux kernel. After running the LLM for over three hours, SyzGPT produces 436 test cases, 79.82% of which are valid after applying the built-in repair mechanism. These validated seeds are merged with the base corpus and used as the initial input for the SyzGPT fuzzing process.

Benchmark. We chose Linux 6.12-rc6, the latest kernel version when SYSYPHUZZ's development started, as our benchmark. Following the recommendation [35], we import the Syzbot corpus from Nov. 13, 2024, as the initial seed corpus.

A. RQ1: Can SYSYPHUZZ *Increase the Hit Counts of Low-frequency Kernel Code?*

We first study whether SYSYPHUZZ boosts the low-frequency kernel code. Specifically, we analyze the hit count distribution of the covered basic blocks (BBs) and compare the distribution of SYSYPHUZZ and Syzkaller.

We analyze the differences in hit counts between the BBs explored by SYSYPHUZZ and Syzkaller. Overall, SYSYPHUZZ slightly increases the hit counts for the 50%- and bottom 25%-frequency BBs, with a 4.0% and 5.2% boost at the end of the campaign, respectively. While the hit counts in frequently executed regions remain largely unchanged, the distribution in low-frequency regions shows a notable divergence. We further look at the bottom 5% of BBs after corpus replay. We examine both their hit counts and the percentage of these initial low-frequency blocks that remain within the bottom 5% during the 72-hour fuzzing campaign (survival rate).

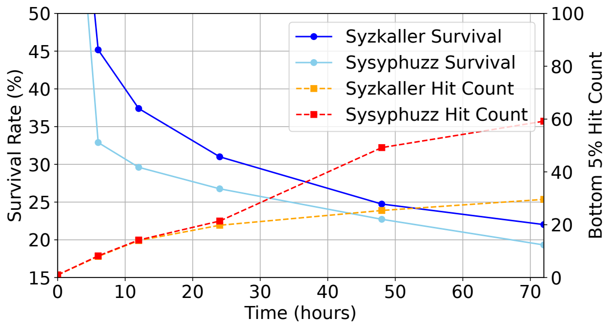


Fig. 7: The survival rate and hit count of low-frequency BBs remaining underexplored in the 72-hour campaign. The x-axis starts from t_0 (*i.e.,* point in time after the corpus replay).

Figure 7 illustrates the proportion of initial low-frequency regions that remain under-explored throughout the fuzzing campaign (referred to as the survival rate). We compare the performance of SYSYPHUZZ and Syzkaller in reducing these under-explored regions. Overall, SYSYPHUZZ reduces the amount of surviving low-frequency region from 22.0% (Syzkaller) to 19.3%. The most significant reduction occurs within the first 6 hours of fuzzing. During this period, SYSYPHUZZ dramatically decreases the percentage of remaining low-frequency regions from 45.2% to 32.9%. While Syzkaller gradually expands into low-frequency regions over time, a substantial portion of these regions remains underexplored. In contrast, SYSYPHUZZ demonstrates its ability to mitigate this imbalance.

Furthermore, even within the set of BBs that remain categorized as low-frequency, SYSYPHUZZ exhibits a significantly higher level of execution frequency compared to Syzkaller. As illustrated by the red dashed line in Figure 7, the BBs at the 5% threshold—*i.e.,* those with hit counts near the bottom of the distribution—are executed nearly 60 times by SYSYPHUZZ. In contrast, Syzkaller executes the same subset fewer than 30 times, indicating that SYSYPHUZZ delivers approximately twice the execution effort on the least-covered regions of the program. This highlights SYSYPHUZZ's capability, not only to reduce the amount of under-explored code, but also to increase the depth of exploration within these regions. Specifically, even when code is infrequently executed, the level of exploration can vary significantly, reinforcing the effectiveness of SYSYPHUZZ 's prioritization strategy. Noted, the hit-frequency increase in SYSYPHUZZ primarily originates from the *Context-Preserving Mutation* strategy, which generates thousands of test cases to boost the target BBs for each seed. In this stage, the *Pre-mutation validation* strategy may increase the hit count accordingly; however, this validation will only execute at the beginning of each job, and introduce only one hit count per job, while the following testcases should include up to thousands of hits. Thus, we consider the impact of *pre-mutation validation* negligible.

To understand the effectiveness of SYSYPHUZZ in managing low-frequency regions, we examine the number of BBs added to the *deny list* in SYSYPHUZZ over the entire campaign. The *deny list* is used to exclude unstable low-frequency regions from the fuzzing targets, thereby conserving energy. At the end of the corpus replay phase (t_0), the *deny list* is initially empty, and all BBs within the bottom 5% of hit counts are considered fuzzing targets. By the end of the fuzzing campaign, however, 17.5% of these initially low-frequency BBs are added to the *deny list*, as they are identified as being associated with non-deterministic kernel behavior. As a result, among the 19.3% of low-frequency BBs that remain at the end of the campaign (as shown in Figure 7), only 1.8% represent regions that SYSYPHUZZ was unable to improve. In other words, 80.7% of the initial low-frequency regions do not survive due to our system. The remaining majority was explicitly excluded via the *deny list*. This outcome highlights the effectiveness of SYSYPHUZZ in both identifying non-viable targets and focusing fuzzing efforts on promising regions. We further analyzed the BBs consistently appearing in the *deny list* across five independent fuzzing campaigns, as shown in Table IV in Appendix A-A, to demonstrate why the effectiveness of SYSYPHUZZ is limited. The *deny list* is a component of our design that tracks BBs that remain persistently unreachable. We found that these basic blocks are difficult to reach due to strict preconditions, which require long and specific syscall sequences that are challenging for mutation-based strategies to generate. We further evaluated SYSYPHUZZ by comparing it with Syzkaller in an extended 120-hour fuzzing campaign. SYSYPHUZZ achieved a 15.7% survival rate for low-frequency regions (approaching the 14.5% *deny list* addition rate), while Syzkaller reached 18.7%. Notably, SYSYPHUZZ consistently achieved approximately double the hit count within the bottom 5% of BBs, demonstrating superior exploration of low-frequency regions during extended campaigns.

RQ1: SYSYPHUZZ effectively balances the kernel code coverage by improving hit counts by 2X and reducing 80.7% of the initial low-frequency regions that remain low-frequency after the fuzzing campaign.

B. RQ2: Can SYSYPHUZZ *Outperform SOTA Kernel Fuzzers?*

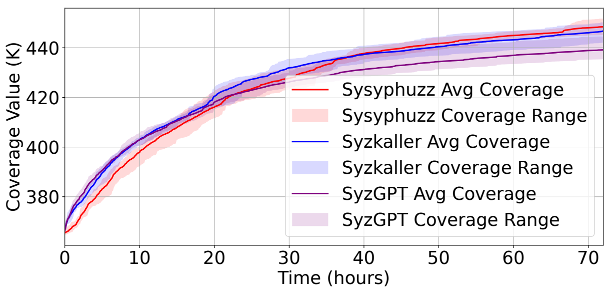


Fig. 8: Coverage achieved by SYSYPHUZZ, Syzkaller, and SyzGPT in 72 hours of fuzzing after corpus replay completion.

We evaluate SYSYPHUZZ against Syzkaller and SyzGPT for 3 days to study if SYSYPHUZZ outperforms SOTA kernel fuzzers, and calculate the *p-value* to test the statistical significance [38].

Figure 8 illustrates their coverage performance. Overall, SYSYPHUZZ and Syzkaller achieve comparable coverage, with Syzkaller exhibiting a more aggressive boost in the first 36 hours. However, SYSYPHUZZ gradually closes the gap and ultimately achieves the highest coverage. Specifically, SYSYPHUZZ outperforms Syzkaller by 8k *(p-value = 0.07)*, and SyzGPT by 20k edges *(p-value = 0.0025)*.

Figure 9 presents the number of bugs discovered by SYSYPHUZZ, Syzkaller, and SyzGPT during the whole campaign. In total, SYSYPHUZZ uncovered 67 bugs, outperforming Syzkaller by 31% with statistical significance *(p-value = 0.03)*. Compared to SyzGPT, SYSYPHUZZ also achieves a marginal improvement. SYSYPHUZZ finds 9.8% more bugs than the 61

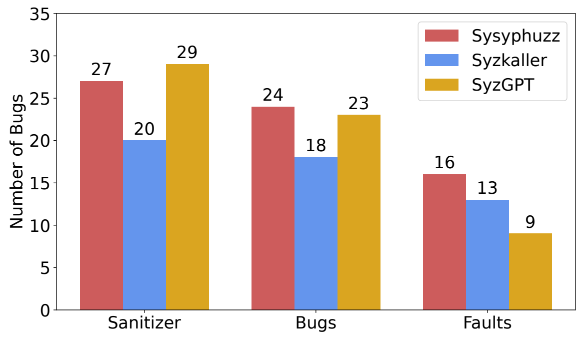


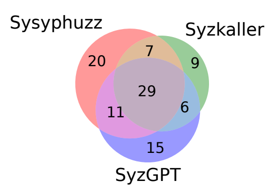
Fig. 9: Number of bugs discovered by three fuzzers. Sanitizer indicates KASan or UBSan errors, fault stands for a general protection fault, and bug means other kernel bugs. All of these bug types are security related [39].

identified by SyzGPT *(p-value = 0.08)*, highlighting SYSYPHUZZ potential. Furthermore, we categorize the discovered bugs into three major types: *Sanitizer Errors*, *General Protection Faults*, and other *Kernel Bugs*. Among all of these categories, SYSYPHUZZ excels in identifying General Protection Faults, discovering 23% more than Syzkaller and 78% more than SyzGPT. For Sanitizer Errors, both SYSYPHUZZ and SyzGPT outperform Syzkaller by 35% and 45%, respectively. Regarding Kernel Bugs, SYSYPHUZZ detects 33% more bugs than Syzkaller and 4% more than SyzGPT.

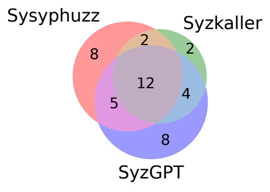
To further understand if these three fuzzers find the same set of bugs, we analyze their bug-finding overlap. Figure 10 illustrates the corresponding Venn diagram. The results indicate that the three fuzzers discover largely disjoint sets of vulnerabilities. Among the 97 unique bugs, only 30% are shared across all fuzzers. Notably, SYSYPHUZZ identifies 21% of the bugs exclusively, while Syzkaller and SyzGPT only find 9% and 15% exclusive bugs. Focusing on specific vulnerabilities, SYSYPHUZZ identifies the highest number of unique bugs exclusively across all three categories. In at least one category, it achieves a substantial lead over the others—for instance, discovering 5X more General Protection Faults than SyzGPT, and 4X more Sanitizer Errors than Syzkaller.

We also examined how consistently each fuzzer can rediscover the same bugs across multiple trials. For every unique bug that was triggered at least once, we counted how many of the five independent trials it appeared in. Then, we calculated the average number of times each bug was rediscovered. On average, SYSYPHUZZ discovers each bug 2.2 times out of five trials, while Syzkaller averaged 2.3 and SyzGPT 2.0 times. The results suggest that all three fuzzers exhibit a moderate level of stability in bug discovery.

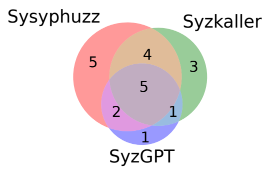
RQ2: SYSYPHUZZ discovers 20 unique bugs missed by both Syzkaller and SyzGPT, outperforming them in both coverage and vulnerability discovery.



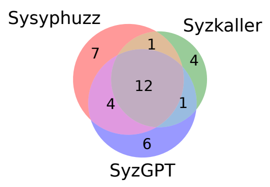
(a) All Bugs



(b) Sanitizer



(c) General Protection Fault



(d) Other Bugs

Fig. 10: Venn diagram for unique bugs discovered by SYSYPHUZZ, Syzkaller and SyzGPT.

C. RQ3: Is SYSYPHUZZ's Bug Discovery Correlated with its Focus on Low-Frequency Regions?

We further investigate the unique sanitizer bugs exclusively discovered by SYSYPHUZZ and Syzkaller to assess whether they are associated with the low-frequency regions. Specifically, we quantify the correlation by counting the number of low-frequency basic blocks (BBs) present in the sanitizer-reported stack trace. These include traces recorded at the bug trigger point, at memory allocation, and at memory deallocation. Each trace represents the active call chain at a specific moment in time that is directly related to the manifestation of the vulnerability. In the sanitizer-reported stack trace, each line represents a stack frame recording a function call and its location. For each frame, we extract its source code file, function, and line number to analyze the correlation in three categories: (i) same line, *i.e.,* the low-frequency BBs are located within the same line as the call stack frame (ii) same function but different line, and (iii) same source file but different function. Considering these granularities allows us to express a gradually decreasing intensity between the bug location and the role of low-frequency regions in the bug's discovery. To achieve this, we take the low-frequency BBs identified by SYSYPHUZZ in Section VI-A before the crash trigger time, and extract their source code locations using addr2line [40]. Finally, based on the extracted locations and the above classification, we categorize the low-frequency BBs into three distinct sets.

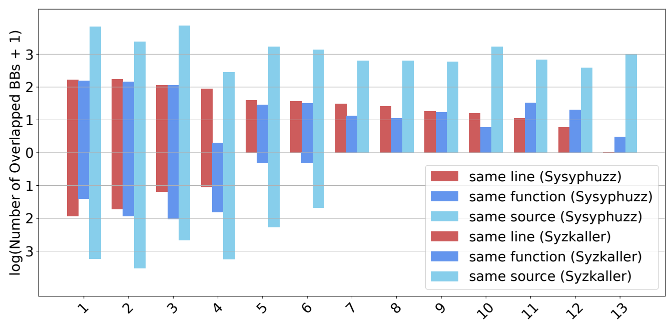


Fig. 11: The overlap between low-frequency BBs identified by SYSYPHUZZ and Sanitizer bugs' callstack frames, exclusively discovered by SYSYPHUZZ and Syzkaller. The x-axis represents the bug ID, while the y-axis shows the number of low-frequency BBs.

The results of this analysis are presented in Figure 11. In this figure, the overlap is categorized based on the granularity of the source code: same line, same function, or same source file. We label a vulnerability as low-frequency related if it covers at least one low-frequency BB in the same line category. Based on this criterion, SYSYPHUZZ identified 12 out of 13 such bugs, whereas Syzkaller detected 4 out of 6. Applying Fisher's Exact Test [41], we estimate that SYSYPHUZZ is about six times more likely to discover low-frequency bugs, but the difference is not statistically significant *(p = 0.22)*. We note that the limited number of bugs reduces the test's statistical power, making the results more sensitive to random fluctuations and less effective at detecting true differences. Additionally, the process of mapping BBs to source lines relies on sanitizer instrumentation, which can be imprecise, potentially affecting the labeling results.

To mitigate the above issues, we aim to understand whether the vulnerabilities discovered by Sysyphuzz are more strongly associated with low-frequency areas compared to the bugs discovered by Syzkaller, rather than arbitrarily counting low-frequency related bug numbers. To assess this, we calculate the average number of low-frequency BBs in the stack trace of each bug detected by the fuzzers in each round, which reflects the association with low-frequency areas. Across five rounds of evaluation, we construct two arrays (from Syzkaller and SYSYPHUZZ), each containing five average values. We then apply the *Bootstrap* method for p-value calculation following the recommendation [38]. On average, each vulnerability discovered by SYSYPHUZZ is associated with 1,000 times more low-frequency BBs than those found by Syzkaller *(p-value = 0.06)*. While this result does not meet the conventional 5% threshold for statistical significance, it suggests a potentially meaningful difference, deserving further investigation. Moreover, this discrepancy indicates a strong correlation between the low-frequency BBs targeted by SYSYPHUZZ and the bug-triggering call stack frames observed in SYSYPHUZZ. This insight helps to explain SYSYPHUZZ's ability to uncover certain unique vulnerabilities that Syzkaller misses, while also accounting for its failure to detect some bugs identified by Syzkaller. Furthermore, mere coverage is insufficient—execution frequency is critical for overlooked vulnerabilities.

For General Protection Faults and Other Kernel Bugs discovered by SYSYPHUZZ, the direct overlap with low-frequency regions is less apparent. None of the corresponding stack traces contains BBs clearly maps to the low-frequency regions identified earlier. However, further investigation reveals that the kernel design limits our study. While sanitizer reports provide clear execution contexts and stack traces, the General Protection Fault and Other Kernel Bug reports are highly abstract, and our analysis is limited by the lack of necessary information. Thus, it's possible that these bugs are directly related to the low-frequency regions, but due to the lack of detailed information, our experiments are not conclusive.

Case Study. To further demonstrate the critical role of low-frequency regions in vulnerability discovery, we present a case study of a use-after-free bug that SYSYPHUZZ exclusively uncovered. This bug arises during Bluetooth connection teardown, where a race condition in `l2cap_sock_teardown_cb` eventually reaches `__list_del_entry_valid_or_report`, a routine that verifies the integrity of a `list_head` before deletion. The vulnerability occurs when the `list_head` pointer already refers to freed memory, leading to a use-after-free error. This bug is triggered when a tight loop opens an L2CAP socket, initiates an accept operation, and repeatedly disables and re-enables the Bluetooth device, thereby causing teardown and reinitialization paths to race. SYSYPHUZZ highlights low-frequency BBs within this execution path, including those in `l2cap_sock_teardown_cb`, which invokes `bt_accept_unlink(sk)`, as well as those in the generic list deletion routine `__list_del_entry(entry)`. By treating system call sequences that traverse these rarely executed regions as valuable mutation targets, SYSYPHUZZ applies *Context-Preserving Mutation* to generate more test cases that perform Bluetooth device allocation and deletion, ultimately creating the conditions necessary to trigger this vulnerability.

RQ3: Focusing on low-frequency areas enables SYSYPHUZZ to uncover exclusive bugs missed by Syzkaller, while potentially missing some bugs Syzkaller finds.

D. RQ4: Can SYSYPHUZZ Uncover Bugs that Syzbot Misses?

As a state-of-the-art kernel fuzzer, Syzkaller is continuously running on hundreds of VMs [1], and their discoveries are publicly available in Syzbot [2]. To demonstrate SYSYPHUZZ's capabilities in discovering overlooked bugs, we compare a single SYSYPHUZZ's server campaign against the continuous campaign in hundreds of Google's VMs. Specifically, we deduplicate our discoveries and search our discoveries on the Syzbot dashboard. To date, SYSYPHUZZ has successfully identified five bugs that Syzbot failed to discover, one of which is confirmed to have security implications by the Linux maintainers and is currently being fixed. We responsibly disclosed all discovered vulnerabilities to the developers, and the full list can be found in Table V in Appendix A-B.

RQ4: SYSYPHUZZ finds 5 new bugs missed by the extensive fuzzing campaigns of Syzbot.

E. RQ5: How do SYSYPHUZZ's Individual Components Contribute to its Performance?

In this subsection, we conduct an ablation study to study how each SYSYPHUZZ component contributes to the final performance. We build SYSYPHUZZ variants with individual components removed. Specifically, we first construct SYSY-RANDMUTATE by replacing the *Context-Preserving Mutation* strategy with the random strategy. Then we build the SYSY-NODENY to assess the impact of the *deny list*. We compare these variants with SYSYPHUZZ and Syzkaller in Table II.

t0+72	Survival Rate	Edge coverage	Bug Count
SYSYPHUZZ	19.3%	448, 430.6	67
Syzkaller	+2.7%	-1,766.0	-16
SYSY-NODENY	-0.4%	-5,872.6	-10
SYSY-RANDMUTATE	+2.4%	-1,982.2	-9

TABLE II: The survival rate, edge coverage, and bug count achieved by SYSYPHUZZ and its variants. All results are presented as the diff compared to the SYSYPHUZZ.

Overall, SYSY-NODENY demonstrates the highest effectiveness in reducing low-frequency BBs, with only 18.9% of the initially identified low-frequency BBs that are still infrequently executed. This outcome is attributed to SYSY-NODENY 's strategy of persistently exploring low-frequency BBs that are difficult to reach reliably. However, this aggressive focus comes at the expense of overall coverage and bug discovery performance. Specifically, SYSY-NODENY yields the lowest edge coverage and the second-lowest number of unique bugs discovered among the evaluated approaches. In contrast, SYSY-RANDMUTATE achieves a survival rate of low-frequency BBs comparable to that of Syzkaller, at 21.7%. This suggests that the *Context-Preserving Mutation* employed by SYSY-RANDMUTATE is effective in reducing the persistence of low-frequency BBs. Notably, both SYSY-RANDMUTATE and SYSY-NODENY discover more unique bugs than Syzkaller, indicating that a targeted focus on low-frequency BBs can enhance the effectiveness of bug discovery.

RQ5: Both *Context-Preserving Mutation* and the *denylist* contribute to SYSYPHUZZ's performance.

F. RQ6: Can SYSYPHUZZ Improve Other Kernel Fuzzers?

	SyzGPTsysy Only	SyzGPT Only	SYSYPHUZZ and Syzkaller Only
Bug Numbers	16	12	29

TABLE III: Bugs exclusively discovered by SyzGPT, SyzGPTsysy, and SOTA kernel fuzzers in Section VI-B.

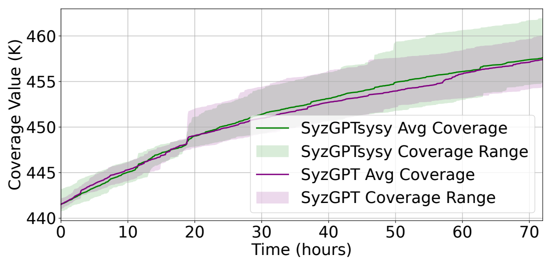


Fig. 12: Coverage achieved by two fuzzers in 3 days. SyzGPTsysy is an SYSYPHUZZ implementation over SyzGPT.

We further assess the potential of SYSYPHUZZ in enhancing other kernel fuzzers' performance. To this end, we combined SyzGPT with SYSYPHUZZ and constructed SyzGPTsysy. Using the same setup as SyzGPT, we run 5 * 72-hour campaigns with SyzGPTsysy and SyzGPT.

SYSYPHUZZ is designed to operate on a saturated corpus—a seed corpus that has been extensively exercised by fuzzers. However, SyzGPT leverages an LLM to generate untested novel inputs that unlock new paths aiming at deep code regions, thus its corpus no longer saturated. To simulate a saturated state, we use the output of a 3-day SyzGPT fuzzing campaign as the initial corpus. This corpus is then used as the starting point for both fuzzers in this evaluation.

Figure 12 illustrates the coverage performance. While both fuzzers achieve similar coverage, SyzGPTsysy performs notably better (600 more edges) compared to SyzGPT in the worst trial, demonstrating the potential of its strategy.

Regarding the bug discovery, we noted that SyzGPTsysy discovers 62 bugs, nearly the same number of bugs as SyzGPT *(SyzGPT discovered two more bugs, with p-value=0.96)*. Table III presents a more in-depth analysis of two fuzzers. Specifically, we compare the bugs discovered by SyzGPT, SyzGPTsysy, and two fuzzers from RQ2 (Syzkaller + SYSYPHUZZ). Overall, SyzGPTsysy discovers 16 unique bugs missed by all SOTA fuzzers, compared to 12 by SyzGPT, which is a 33% increase *(p-value = 0.11)*. This highlights SYSYPHUZZ's strength in uncovering overlooked vulnerabilities.

RQ6: SYSYPHUZZ strategy can help other kernel fuzzers to find overlooked bugs on a saturated corpus.

VII. DISCUSSION

While SYSYPHUZZ's approach significantly boosts kernel fuzzing, some factors may affect the results.

Low-Frequency Region Threshold. We define low-frequency regions as the bottom 5% of the least frequently executed code regions. While this threshold is selected based on the study of frequency distribution, we acknowledge that varying this hyperparameter could influence both the size of the identified low-frequency regions and the resulting behavior of SYSYPHUZZ. Nonetheless, we believe this setting captures a representative trend, and adjustments to this parameter do not undermine our core observations or the contributions of SYSYPHUZZ. Fine-tuning this threshold for performance optimization is left as future work.

Initial Corpus. To ensure a realistic and fair evaluation, we adopt the Syzbot [2] corpus—curated from a long-term, large-scale fuzzing campaign—following best practices outlined in prior work [42], [43]. This provides SYSYPHUZZ with a reasonable initial coverage baseline. For fairness, we initialize Syzkaller with the same corpus. Although beginning with an empty corpus might affect absolute performance, we argue that in practical, long-running fuzzing campaigns, the corpus tends to become saturated over time. Thus, using a coverage-rich initial corpus aligns well with real-world deployment scenarios and does not compromise the validity of our evaluation.

VIII. RELATED WORK

A. Linux Kernel Fuzzing

Linux kernel fuzzers typically operate by taking sequences of system calls (syscalls) as input and applying a variety of mutation strategies, such as inserting, deleting, splicing syscalls, or modifying their arguments, to generate new test cases. These mutated inputs are then executed in the kernel to explore new execution paths. Consequently, the quality of generated inputs directly influences the effectiveness of fuzzing. Substantial research has been dedicated to improving input quality. For instance, Difuze [13] enhances input construction by leveraging interface descriptions, while Moonshine [29], Healer [30], and Mock [31] focus on modeling syscall dependencies to guide meaningful mutations. In addition, symbolic execution techniques have been employed to satisfy complex constraints and generate high-quality seeds [15], [17].

Beyond input generation, feedback mechanisms have also been extensively explored to better guide fuzzing efforts. Traditional fuzzers rely on code coverage as the primary feedback signal, but alternative approaches have emerged. StateFuzz [10], for example, introduces state-space exploration to expose kernel states that coverage alone may miss. Meanwhile, Actor [11] and CountDown [12] adopt bug-guided fuzzing, directing the fuzzer towards specific vulnerability patterns to maximize the likelihood of uncovering critical bugs.

In contrast to existing work, SYSYPHUZZ introduces a novel and so far overlooked dimension in the kernel: code execution frequency. By leveraging code frequency as a feedback signal, SYSYPHUZZ identifies and targets rarely executed regions of the kernel. To this end, it further refines the mutation strategy to specifically address the imbalance in execution frequency, thus complementing and enhancing existing coverage-guided kernel fuzzing techniques.

B. Low-Frequency Region Fuzzing

The concept of code frequency has been explored in user-space coverage-guided greybox fuzzing for several years with the goal of guiding fuzzers towards uncovered areas. Beginning with AFLFast [20], researchers modeled the fuzzing process as a Markov chain to increase the probability of reaching rarely executed code paths. Building upon this, EcoFuzz [19] introduced a multi-armed bandit model to dynamically adjust energy allocation during fuzzing. Alternatively, FairFuzz [21] proposed a mask-based mutation strategy to bias the entire fuzzing campaign toward low-frequency regions. All these techniques, despite directing fuzzing to low-frequency regions, finally aim at finding new coverage.

Although these techniques have proven effective in user-space applications, they do not scale to kernel fuzzing due to the task explosion problem and destroyed context dependencies. To overcome these challenges, we introduce SYSYPHUZZ, a frequency-based kernel fuzzer customized for these challenges. By incorporating code frequency as a feedback signal and optimizing for context preservation, SYSYPHUZZ effectively uncovers overlooked bugs in the kernel that are often missed by traditional coverage-guided kernel fuzzers.

IX. CONCLUSION

The kernel is one of the most security-critical components in modern computing. However, current efforts to improve coverage in kernel fuzzing plateaued. Even worse, the focus on coverage fails to represent sufficient exploration, which necessitates a new approach to boost kernel fuzzing effectiveness. We propose SYSYPHUZZ, the first kernel fuzzer that targets already covered but rarely executed code. SYSYPHUZZ first introduces a *Selective Task Scheduler* to balance between coverage and code frequency. Subsequently, SYSYPHUZZ proposes *Context Preserving Mutations* to protect syscall dependencies. We evaluated SYSYPHUZZ against Syzkaller and found that SYSYPHUZZ discovered 31 unique bugs missed by Syzkaller. In addition, five bugs of these are not found by Syzbot.

X. ETHICS CONSIDERATIONS

We have responsibly disclosed all the discovered vulnerabilities to the Linux developer and worked with them to fix. We release the SYSYPHUZZ at https://github.com/HexHive/Sysyphuzz to support open science.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful and constructive feedback. We also express our gratitude to Chibin Zhang, Florian Hofhammer, Luca Di Bartolomeo, Ming Yuan, Nicolas Badoux, Philipp Mao, Qiang Liu, Rafaila Galanopoulou, Shangru Zhao, Tao Lyu, Xudong Cao, Yishun Zeng, Zheyu Ma, and the rest of the HexHive group for their invaluable support and suggestions throughout this work.

This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2 186974, the State of Upper Austria in the frame of the COMET Module DEPS (grant no. 888338) managed by FFG, the China Scholarship Council, the National Natural Science Foundation of China under grant U24A20337, and Joint Research Center for System Security, Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd, National Key Research and Development Program of China (No. 2023YFB3106400, 2023QY1202), the National Natural Science Foundation of China (U2336203, U1836210), and Beijing Natural Science Foundation (L242015, 4242031).

REFERENCES

[1] Google, "clusterfuzz," https://github.com/google/clusterfuzz, 2024.
[2] ——, "syzbot," https://github.com/google/syzkaller/blob/master/docs/syzbot.md, 2024.
[3] ——, "syzkaller," https://github.com/google/syzkaller, 2024.
[4] D. Liu, Z. Lu, S. Ji, K. Lu, J. Chen, Z. Liu, D. Liu, R. Cai, and Q. He, "Detecting kernel memory bugs through inconsistent memory management intention inferences," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4069–4086.
[5] Y. Yang, W. Shen, X. Xie, K. Lu, M. Wang, T. Zhou, C. Qin, W. Yu, and K. Ren, "Making memory account accountable: Analyzing and detecting memory missing-account bugs for container platforms," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 869–880.
[6] Q. Zhou, Q. Wu, D. Liu, S. Ji, and K. Lu, "Non-distinguishable inconsistencies as a deterministic oracle for detecting security bugs," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3253–3267.
[7] Q. Wu, Y. Xiao, X. Liao, and K. Lu, "Os-aware vulnerability prioritization via differential severity analysis," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 395–412.
[8] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi, "Linkrid: Vetting imbalance reference counting in linux kernel with symbolic execution," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 125–142.
[9] G. Lee, D. Xu, S. Salimi, B. Lee, and M. Payer, "Syzrisk: A change-pattern-based continuous kernel regression fuzzer," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1480–1494.
[10] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "Statefuzz: System call-based state-aware linux driver fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3273–3289.
[11] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna, "Actor: Action-guided kernel fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5003–5020.
[12] S. Bai, Z. Zhang, and H. Hu, "Countdown: Refcount-guided fuzzing for exposing temporal memory errors in linux kernel," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1315–1329. [Online]. Available: https://doi.org/10.1145/3658644.3690320
[13] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3133956.3134069
[14] W. Chen, Y. Hao, Z. Zhang, X. Zou, D. Kirat, S. Mishra, D. Schales, J. Jang, and Z. Qian, "Syzgen++: Dependency inference for augmenting kernel driver fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4661–4677.
[15] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel." in *NDSS*, 2020.
[16] P. B. Oswal, *Improving linux kernel fuzzing*. Carnegie Mellon University, 2023.
[17] Y. Lan, D. Jin, Z. Wang, W. Tan, Z. Ma, and C. Zhang, "Thunderkaller: Profiling and improving the performance of syzkaller," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1567–1578.
[18] Syzbot, "Syzkaller linux kernel coverage," https://syzkaller.appspot.com/upstream/coverage?period=month, 2025.
[19] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2307–2324.

[20] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[21] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 475–485.

[22] D. She, A. Storek, Y. Xie, S. Kweon, P. Srivastava, and S. Jana, "Fox: Coverage-guided fuzzing as online stochastic control," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 765–779.

[23] Z. Zhang, L. Li, R. Liang, and K. Chen, "Unlocking low frequency syscalls in kernel fuzzing with dependency-based rag," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 848–870, 2025.

[24] A. Nogikh, "Syzbot: 7 years of continuous kernel fuzzing," https://lpc.events/event/17/contributions/1521/attachments/1272/2698/LPC'23_%20Syzbot_%207%20years%20of%20continuous%20kernel%20fuzzing.pdf, 2023.

[25] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1749–1766.

[26] S. Han, "Breaking hardware-assisted kernel control-flow integrity with page-oriented programming," https://i.blackhat.com/BH-US-23/Presentations/US-23-Han-Lost-Control-Breaking-Hardware-Assisted-Kernel.pdf, 2023.

[27] Y. Liu, J. Yao, and X. Wang, "Usma: Share kernel code with me," https://i.blackhat.com/Asia-22/Thursday-Materials/AS-22-YongLiu-USMA-Share-Kernel-Code.pdf, 2022.

[28] M. Bogaard and D. Geist, "Achieving linux kernel code execution through a malicious usb device," https://i.blackhat.com/EU-21/Thursday/EU-21-Bogaard-Geist-Achieving-Linux-Kernel-Code-Execution-Through-A-Malicious-USB-Device.pdf, 2021.

[29] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing os fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 729–743.

[30] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "Healer: Relation learning guided kernel fuzzing," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 344–358.

[31] J. Xu, X. Zhang, S. Ji, Y. Tian, B. Zhao, Q. Wang, P. Cheng, and J. Chen, "Mock: optimizing kernel fuzzing mutation with context-aware dependency," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2024.

[32] M. Zalewski, "American fuzzy lop: Technical whitepaper," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016.

[33] A. Nogikh, "Syzbot: 7 years of continuous kernel fuzzing," in *Linux Plumbers Conference*, 2023. [Online]. Available: https://lpc.events/event/17/contributions/1521/attachments/1272/2698/LPC%2723_%20Syzbot_%207%20years%20of%20continuous%20kernel%20fuzzing.pdf

[34] stackscale, "The linux kernel surpasses 40 million lines of code: A historic nilestone in open-source software," https://www.stackscale.com/blog/linux-kernel-surpasses-40-million-lines-code/, 2025.

[35] P. B. Oswal, "Improving linux kernel fuzzing," Ph.D. dissertation, 2023. [Online]. Available: https://www.proquest.com/dissertations-theses/improving-linux-kernel-fuzzing/docview/2812311865/se-2

[36] A. Nogikh, "corpus progs with non-reproducible coverage," https://github.com/google/syzkaller/issues/4639, 2024.

[37] Z. Zhang, L. Li, R. Liang, and K. Chen, "syzgptgenerator," https://github.com/QGrain/SyzGPT/blob/main/generator/README.md, 2025.

[38] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1974–1993.

[39] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "Grebe: Unveiling exploitation potential for linux kernel bugs," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2078–2095.

[40] R. H. Pesch and J. M. Osier, "The gnu binary utilities," *Free Software Foundation*, 1993.

[41] G. J. Upton, "Fisher's exact test," *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, vol. 155, no. 3, pp. 395–402, 1992.

[42] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[43] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 230–243.

Appendix A
Additional Results

A. *Table IV: Basic Blocks in the Deny List*

TABLE IV: BBs in the deny list at t0 + 72h shared by 5 rounds of SYSYPHUZZ

ID	Function Name	Preconditions to Reach the Function	Line Numbers and Preconditions to Reach the Lines
1	vma_merge_new_range	The `mmap_lock` is held in write mode, the range `vmg->start`, `vmg->end` is empty, and the surrounding VMAs are identified and provided	930 (In an `if` conditional statement) 981 (Need to pass an `if` condition)
2	queue_folios_hugetlb	Huge pages are mapped, memory scanning follows the Non-Uniform Memory Access policy, and the virtual memory region is migratable	651 (Need to pass two `if` conditions)
3	nested_vmx_check _guest_state	A nested hypervisor must be emulated, causing control to traverse deep nested virtualization logic	3120 (In an `if` conditional statement)
4	blkdev_write_iter	A block device is ready for writing and a write system call or `io_uring write`, which go through raw I/O paths is performed	715 (Need to pass two `if` conditions)
5	io_sqe_buffers_register	A valid `io_uring` instance already exists and `io_uring_register` syscall is called	1050 (Need to pass five validations, including base pointer and length checks)
6	usb_fill_int_urb	A simulated USB device and driver are present, and a corresponding URB structure is allocated	1758 (Need to pass an `if` condition)
7	ld_usb_abort_transfers	Specific USB hardware and drivers are emulated, a device disconnect or error handling path is triggered, and the active URBs are configured with the correct flags.	198 (Need to pass two `if` conditions)
8	__refcount_add_not _zero	A shared kernel object (e.g., `file` or `socket`) exists with a non-zero refcount, and logic to increment the refcount is triggered	147 (Need to pass a `compare-and swap (CAS)` loop) 152,155 (Need to pass a `CAS` loop and 2 `if` conditons)
9	vivid_vid_cap_s_ctrl	The vivid driver is loaded, `/dev/videoX` is opened, and a `V4L2 ioctl` command is issued with a valid control ID	611 (In one of the 31 `switch` branches)
10	nf_tables_commit	The caller has sufficient privileges, changes are submitted via the Netlink Netfilter interface, and the commit function is invoked at the end of a valid netfilter operation batch	10325, 10326, 10327, 10328 (In one of the three `switch` branches Need to pass a strict `if` condition)
11	maybe_get_net	A net struct exists, and operations are triggered inside the namespace	269 (In an `if` condition)
12	xfrm_flowi_sport	XFRM policies are active, packets match specific IPsec rules, and flow lookups are triggered accordingly.	924 (In one of the nine `switch` branches)
13	do_ipv6_setsockopt	IPv6 socket is setup properly first	533 (In one of the 16 `switch` branches and need to pass eight `if` conditions)

B. *Table V: Unique Bug Findings by* SYSYPHUZZ

TABLE V: New bugs exclusively discovered by SYSYPHUZZ compared to Syzkaller. +: found during development.

ID	Crash Type	Root Cause	SyzGPT	Syzbot
1	Slab-out-of-bounds Read	`crc32c_pcl_intel_update` accessed buffer using oob index	✗	✓
2	Slab-out-of-bounds Read	`hex_dump_to_buffer` allocate too small buffer	✗	✓
3	Slab-out-of-bounds Read	`hfsplus_uni2asc` accessed the buffer using oob index	✓	✓
4	Slab-out-of-bounds Read	`journal_entry_dev_usage_to_text` accessed buffer using oob index	✗	✓
5	Slab-use-after-free Read	`List_del` accessed freed buffer in L2CAP teardown	✓	✓
6	Slab-use-after-free Read	`l2cap_sock_ready_cb` accessed a destroyed socket in L2CAP set	✓	✓
7	Slab-use-after-free Read	`scatterwalk_copychunks` accessed a freed sk_buff during network encryption	✓	✓
8	Slab-use-after-free Read	`set_powered_sync` accessed a freed mgmt_pending struct during synchronous handling	✗	✓
9	Slab-use-after-free Read	`device_release` accessed a freed object during a netdev is being unregistered	✗	✓
10	Slab-use-after-free Read	`dccp_ackvec_runlen` accessed a freed dccp_ackvec during the processing of DCCP packets	✗	✓
11	Slab-use-after-free Read	`cmd_complete_rsp` accessed a freed mgmt_pending struct during Bluetooth device shutdown	✗	✓
12	Null-ptr-deref Write	`dst_release` accessed freed struct dst during network namespace destroy	✓	✓
13	Stack-out-of-bounds Read	`iov_iter_revert` access the stack array out of bound	✗	✓
14	Kernel Bug in GFS2	The assertion failed in the `gfs2_qd_dispose`, when mounting GFS2 filesystem	✓	✓
15	Kernel Bug in JFS	The assertion in `txEnd`, when creating a directory	✗	✓
16	Kernel Bug in Mmap	The assertion failed in the `__age_table_check_zero` when freeing USB buffer	✗	✓
17	Kernel Bug in JFS	The assertion failed in the `LogSyncRelease`, when renaming a file	✓	✓
18	Kernel Bug in Bcachefs	The assertion failed in the `bch2_trans_node_iter_init`, when recovering filesystem	✗	✓
19	Kernel Bug in Bcachefs	The assertion failed in the `bch2_btree_path_level_init`, when traversing and locking a B-tree node	✗	✗
20	Kernel Bug in Bcachefs	The assertion failed in the `bch2_btree_node_iter_init`, when processing a B-tree node write	✗	✗
21	Kernel Bug in Bcachefs	The assertion failed in the `bkey_s_c_to_backpointer` during the recovery process	✗	✓
22	Kernel Bug in Fsnotifys	The assertion failed in the `dnotify_free_mark` when fsnotify was trying to destroy a mark	✗	✓

(continued on next page)

(continued from previous page)

ID	Crash Type	Root Cause	SyzGPT	Syzbot
23	Bug: Soft Lockup	The kernel deadlock in the networking subsystem, when processing socket operation.	✓	✓
24	Bug: Null-ptr-deref	A NULL pointer dereference in OverlayFS's `__lookup_slow`, when traversing directory	✓	✓
25	General Protection Fault	A wild memory access in the `__lock_acquire` during Btrfs shutdown, in btrfs_destroy_workqueue	✓	✓
26	General Protection Fault	Null pointer dereference in `fuse_read_args_fill`, when EROFS mount over FUSE	✓	✓
27	General Protection Fault	A null pointer dereference in `iov_iter_revert` when processing `netfs_write_collection_worker` task	✗	✗
28	General Protection Fault	`percpu_counter_add_batch` accessed a freed CPU counter when cleanup the network namespace	✗	✗
29	General Protection Fault	`rhashtable_lookup_fast` use uninitialized key, during the hashtable lookup in the Identifier Locator Addressing	✗	✓
30	General Protection Fault	A dangling pointer dereference in `write_special_inodes` during JFS unmount	✗	✓
31	General Protection Fault	`__list_del_entry_valid_or_report` accessed an invalid memory	✗	✓
32^{+}	Slab-use-after-free Read	`cgwb_release_workfn` accessed a freed blkcg structure in `blkcg_unpin_online`	✗	✗

Appendix B
Artifact Appendix

In this artifact, we provide a detailed guidance to deploy Sysyphuzz and reproduce the experiments presented in the paper.

A. Description and Requirements

This artifact provides the necessary materials to reproduce the results for the following research questions:

- **RQ1:** Can SYSYPHUZZ increase the hit counts of low-frequency kernel code? The corresponding results are presented in Section VI, Figure 7 of the manuscript.
- **RQ2:** Can SYSYPHUZZ outperform SYZKALLER in terms of coverage and bug discovery? The results are reported in Section VI, Figures 8–10 and Table V.
- **RQ3:** Is the bug discovery performance of SYSYPHUZZ correlated with low-frequency code regions? The supporting evidence is presented in Section VI, Figure 11.

1) How to access: The code is available in zendo https://doi.org/10.5281/zenodo.15961011 or in the GITHUB repository https://github.com/HexHive/Sysyphuzz/tree/main.

2) Hardware dependencies: This artifact requires at least 16 CPU cores, 64 GB of RAM, and a 128 GB disk.

3) Software dependencies: This artifact requires a Linux environment with all dependencies configured as required by Syzkaller.

B. Artifact Installation & Configuration

Run the *deploy.sh* script with sudo privileges. This script creates a new user *fuzz* and sets up the environment. You can modify the default password in the shell file if needed. Once executed, Sysyphuzz is ready for use.

C. Experiment Workflow

Our experimental workflow consists of two steps. First, we run SYSYPHUZZ and SYZKALLER for over three days to collect the necessary data. Due to hardware constraints, the server supports running only one fuzzer at a time. As a result, users must manually alternate between executing SYSYPHUZZ and SYZKALLER during the experiment.

```bash
# Make sure you are the user "fuzz"
# and in the sysyphuzz directory
su fuzz
cd ~/code/sysyphuzz/
# Start Sysyphuzz
sudo bin/syz-manager \
-config sysyphuzz.cfg 2>&1 \
| tee ./workdir_sysy/"$(date +"%Y_%m_%d").log"
# Useing "ctrl + c" to stop.
# Change to syzkaller.cfg and workdir_syzk
```

Listing 1: Running Bash Cmds

After the first step, two types of logs will be generated: *fuzzing logs* and *hit count logs*. The former are stored in the `workdir_sysy` or `workdir_syzk` directories, while the latter are saved in the corresponding `sysyphuzz_bb` or `syzkaller_bb` directories.

All directory paths can be customized in the respective `*.cfg` configuration files. The above experiment should be repeated to ensure that four rounds of results are collected for both SYSYPHUZZ and SYZKALLER. For each round, the same corpus should be used by renaming `ci-qemu-upstream-corpus.db` to `corpus.db`, ensuring consistency across all runs.

```
# Fuzzing logs
-/home/fuzz/code/sysyphuzz/workdir_sysy #_syzk
  -YYYY-MM-DD.log
  -crashes
# Hit count logs
-/home/fuzz/code/sysyphuzz/sysyphuzz_bb #
    syzkaller_bb
  -YYYYMMDD_hh
    -HitBBPerc_hhmm.json
    -UnderCoveredBBs_hhmm.json
```

Listing 2: Log Structure

Run the following analysis scripts to reproduce the evaluation results and support the main claims of the paper.

D. Major Claims

- (C1): SYSYPHUZZ effectively balances the kernel code coverage by improving hit counts and reducing the survival rate of initial low-frequency regions. This is proven by the experiment (E1) whose results are illustrated in VI-Fig7.
- (C2): SYSYPHUZZ discovers more unique bugs missed by SYZKALLER and slightly outperforms SYZKALLER in code coverage. This is proven by the experiment (E2) whose results are illustrated in VI-Fig8 to Fig10.
- (C3): SYSYPHUZZ's bugs discovered by SYSYPHUZZ are largely related to underexplored low-frequency regions. This is proven by the experiment (E2) whose results are illustrated in VI-Fig11.

E. Evaluation

1) Experiment (E1): [Low-frequency areas analysis] [60 human-minutes + 1 compute-hour]: For the six observation time points, we extract the hit counts of the low-frequency regions and compute the survival rate of the initially identified low-frequency regions across subsequent time points.

[How to] In each experimental trial, collect hit count logs at six time points. Run two scripts: one to extract low-frequency regions at each point, and another to compute the survival rate of the regions identified at the first time point.

[Preparation] For each trial, collect hit count logs at six time points: t0, t0+6h, t0+12h, t0+24h, t0+48h, and t0+72h. The reference point t0 marks the completion of corpus replay and the start of fuzzing, which is the creation time of the first hit count log.

```
# Example of the Hit count log create time.
-/home/fuzz/code/sysyphuzz/sysyphuzz_bb
  -YYYYMMDD_hh
    -UnderCoveredBBs_hhmm.json
```

Listing 3: Get Hit Count Logs

The timestamp in the format YYMMDD_hhmm corresponds to the creation time of each UnderCoveredBBs JSON file. We identify the first available file as the t0 time point and manually compute the subsequent five time points. If a required file is missing, we substitute it with the nearest available file from an earlier time. Each of the selected files should be copied to a new directory and renamed as t0.json, t0+6h.json, etc. This process is repeated for all trials in the experiment.

[Execution] For each trial of the experiments, get the low-frequency areas in six time points by using script *get_low_area.py*. The input files are the six JSON files created in the preparation step. There will be six JSON files in the OUTPUT_DIR, namely filtered_t0.json_1, filtered_t0+6h.json_2, etc.

```
# --per is 5 in our experiments.
python3 get_low_area.py -h
usage:
--input_files INPUT_FILES [INPUT_FILES ...] \
--per PER --output_dir OUTPUT_DIR
```

Listing 4: Get Low-Frequency Areas

Repeat the above step for the rest of the trials of the experiments. For four trials of the experiments, there will be four filtered JSON files for each time point, e.g., trial1_filtered_t0.json_1, trial2_filtered_t0.json_1, etc. Using the script *get_average_hit.py* to get the average hit counts of the low-frequency areas at t0, repeat this step for the rest of the time points.

```
python3 get_average_hit.py -h
usage:
--input_files INPUT_FILES ...  \
--output_file OUTPUT_FILE
```

Listing 5: Get Average Hit Counts

Then, using *get_consist.py* to compute the survival rate of the low-frequency areas identified at t0, the script processes each trial by reading the BBs from trial*_filtered_t0.json and evaluating their presence in the subsequent time points logs, e.g., trial*_filtered_t0+6h.json, trial*_filtered_t0+12h.json, etc.

```
python3 get_consist.py -h
usage:
--input_files INPUT_FILES ...  \
--output_file OUTPUT_FILE
```

Listing 6: Get Survival Rate

Repeat the above steps for the remaining trials to generate four survival rate JSON files in total. Then, use the script *get_average_survival.py* to compute the average survival rate of the initial low-frequency areas.

```
python3 get_average_survival.py -h
usage:
--input_files INPUT_FILES [INPUT_FILES ...]  \
--output_file OUTPUT_FILE
```

Listing 7: Get Average Survival Rate

[Results] The average hit counts and survival rates will be used in the VI-Fig7.

2) *Experiment (E2):* [Coverage and bug analysis] [hundred human-hours + tens compute-hours]: Based on the fuzzing logs under workdir, running the script *plog_coverage.py* to draw the coverage, bug performance, and bug overlap figures.

```
# MAX_HOURS is hours after corpus replay,
# e.g., 72 means t0+72.
python3 plog_coverage.py -h
usage:
# Run logs of fuzzer1; in multi-round
   experiments, logs are separated by spaces.
--boost_logs BOOST_LOGS ... \
# Run logs of fuzzer2; in multi-round
   experiments, logs are separated by spaces.
--coverage_logs COVERAGE_LOGS ... \
--output OUTPUT [--max_hours MAX_HOURS]
```

Listing 8: Coverage Bash Cmds

[Results] The output figure will be used in the VI-Fig8.

Running the script *plog_coverage_batch_totoalcrash_1.py* to get overall bug performance, and use the script *classify.py* to classify the crashes based on different bug types.

```
plog_coverage_batch_totoalcrash_1.py -h
usage:
--boost_logs BOOST_LOGS  \
--coverage_logs COVERAGE_LOGS \
--output OUTPUT \
--crash_report CRASH_REPORT \
[--max_hours MAX_HOURS]
#  Classify Bugs.
python3 classify.py -h
usage:
--input_json INPUT_JSON \ # CRASH_REPORT
--output_dir OUTPUT_DIR
```

Listing 9: Bug Analysis Bash Cmds

[Results] The data will be used to draw the VI-Fig9,10.

For each unique bug, extract the crash stack trace from the crash log (e.g., `report0`) and the first trigger time from the fuzzing log. Identify and merge all low-frequency areas identified by SYSYPHUZZ prior to the bug trigger time. Finally, compute the overlap between these merged low-frequency areas (e.g., `merged_low_bb.csv`) and the bug's crash stack trace (e.g., `crash_stack.csv`).

```
python3 extract_crash_trace_segment.py -h
usage:
-i INPUT [-o OUTPUT]
# Getting the merged low-frequency areas.
python3 low_bb_bef_crash_time.py -h
usage:
-syzlog SYZLOG \ # Sysyphuzz fuzzing log
-bblog BBLOG \ # related hit count log
   directory
-vmlinux VMLINUX \ # vmlinux binary
-time_stamp TIME_STAMP \ # first trigger time
-o O
# Find the overlap.
python3 overlap_analysis.py -h
usage:
-crash_csv CRASH_CSV  -lf_csv LF_CSV [-o O]
```

Listing 10: Bug Overlap Bash Cmds

[Results] The data will be used in the VI-Fig11.