École Polytechnique Fédérale de Lausanne

Syzkaller Profiling

by Nicolas Raulin

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Zhiyao Feng
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 7, 2024

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Zhiyao Feng, for her invaluable guidance and support throughout my Master's project. I am particularly thankful for her prompt and helpful responses to all my Syzkaller-related questions, as well as for the insightful brainstorming sessions discussing the results I obtained during each week.

I also acknowledge the use of the LaTeX template created by Prof. Mathias Payer from the HexHive laboratory at EPFL, where I conducted my Master's project. The template greatly facilitated the writing process of this thesis.

Lausanne, June 7, 2024 Nicolas Raulin

Abstract

Kernel fuzzing is a powerful concept to test ever-changing and rapidly growing codebases such as the Linux kernel. These OS fuzzers mutate sequences of system calls and execute them in a controlled environment hoping to trigger new bugs, or at least discover new coverage points in the kernel under test. Syzkaller, a state-of-the-art kernel fuzzer, plays a crucial role in identifying vulnerabilities within the Linux kernel. Profiling this tool can reveal bottlenecks and facilitate improvements in its codebase, leading to increased fuzzing efficiency, which in turn may help discovering new bugs in the target kernel. Existing studies base their work on an outdated version of Syzkaller, hence profiling the latest version might yield interesting insights. In this paper, we profiled Syzkaller and identified its queue of programs awaiting execution as the main bottleneck. We developed two experiments aiming to improve Syzkaller's efficiency. We propose a first experiment based on Syzkaller's `generate` mode that marginally improves its early stage discoveries by 2.5%. We also developed a new mutation strategy to could yield promising results if adequate system calls dependency analysis is performed during its execution.

Résumé

Le fuzzing d'un noyau d'exploitation est un puissant concept lorsqu'il s'agit de tester des bases de code toujours grandissantes et en évolution permanente comme le noyau Linux. Ces fuzzers modifient des séquences d'appels système au moyen de mutations pour ensuite les exécuter dans un environnement contrôlé dans l'espoir de déclencher de nouveaux bugs, ou au moins découvrir de nouveaux points de couverture (coverage points) dans le noyau sous test. Syzkaller est un fuzzer de noyaux dernier cri qui joue un rôle crucial dans l'identification de vulnérabilités dans le noyau Linux. Le profilage de cet outil peut révéler des goulots d'étranglement (bottlenecks) et ainsi faciliter les améliorations de sa code base, pouvant ainsi augmenter l'efficacité du fuzzing et donc aider à trouver de nouveaux bugs dans le noyau. Les études précédentes basent leur travail sur une version dépassée de Syzkaller ; le profilage de la dernière version du logiciel pourrait révéler de précieuses informations. Dans ce travail de recherche, nous avons profilé Syzkaller et identifié sa file de programmes attendant leur exécution comme étant le principal bouchon. Nous avons développé deux expériences ayant pour but d'améliorer l'efficacité de Syzkaller. Nous proposons une première expérience basée sur le mode `generate` de Syzkaller qui a légèrement augmenté les découvertes dans les premières étapes de 2,5% une fois testée. Nous avons aussi développé une nouvelle stratégie de mutation qui pourrait bien renvoyer des résultats prometteurs pour autant que de bonnes analyses de dépendance entre les appels système soient effectuées pendant son exécution.

Contents

Chapter 1

Introduction

Fuzzing has emerged as a highly effective technique for uncovering software bugs, especially within the vast and complex codebases of operating system kernels. Given the critical role that kernels play in managing system resources, ensuring their reliability is paramount. The sheer size and complexity of kernel code present significant challenges for traditional testing methods, making automated tools like fuzzers invaluable. Among these tools, Syzkaller [6] stands out for its ability to create diverse test cases. Syzkaller operates as a gray box fuzzer, leveraging knowledge about the internal structure of the kernel to produce more targeted test inputs. The programs it generates are essentially sequences of system calls, which allow it to explore and test a wide range of kernel behaviours and interactions.

The main challenge in Syzkaller profiling lies in balancing several critical factors. Performance overheads present a concern, as profiling tools inevitably introduce additional load on the system. It is essential to ensure that this overhead does not substantially degrade the performance of the fuzzer or the kernel. Finding a balance between the frequency of profiling data and maintaining an acceptable performance impact is crucial. Additionally, the security and stability of the kernel must not be compromised by the profiling process. Profiling code should be accurately designed to avoid becoming a source of bugs or vulnerabilities. Furthermore, the final step of interpreting the collected data requires analytical tools to transform raw logs into meaningful insights in order to identify areas for improvement.

Profiling plays a vital role in assessing the coverage and effectiveness of a fuzzer, providing critical insights that can guide further improvements. However, this aspect tends to be overshadowed by other areas in the literature. Additionally, much of the profiling research that does exist is based on outdated versions of Syzkaller, lacking central features like jobs and requests added very recently [7]. Consequently, the current research fails to fully capture the capabilities and nuances of the latest Syzkaller versions, underscoring the need for more focused and updated profiling studies.

We profiled Syzkaller using simple timers and counters to measure most metrics, ensuring minimal overhead and straightforward implementation. We first implemented basic metrics to track various aspects of Syzkaller's on `syz-manager`'s dashboard. Based on these initial results, we identified areas of interest, particularly the `smash` mode and the `squashAny` mutator, which are frequently used. We conducted an ablation study and a single mode study to isolate and understand the impact of isolated modes of operation. Additionally, we introduced a new mutator called `shuffle` and experimented with the `generate` mode to observe its effects. We also examined the removal of dependency analysis within Syzkaller. To facilitate thorough analysis, we used Jupyter notebooks to document and evaluate each study and experiment comprehensively.

This approach was chosen due to its simplicity and low overhead, making it possible to get detailed profiling data without significantly impacting Syzkaller's performance. We focused on the `smash` mode because it is the source of most requests, making it a critical area for optimisation. The `squashAny` mutator was also a priority since it is the only mutator that can fail depending on the nature of the input program. Understanding the impact of individual modes and their removal through single mode and ablation studies was crucial because these modes are fundamental to Syzkaller's operation. Finally, creating Jupyter notebooks for each run allowed for easier analysis, making it simple to interpret results over multiple runs.

This research project examines the evolution of Syzkaller's behaviour over time, identifies critical bottlenecks, and proposes three targeted experiments to mitigate these issues.

In our profiling of Syzkaller, we observed that mutation requests triggered by the `smash` job account for 97.2% of the overall basic requests (`generate`, `mutate`, mutations from `smash`, and `hints`). On average, these requests remain in Syzkaller's priority queue for 11.64 seconds, suggesting a likely saturation of the queue, given that a single mutation request requires approximately 0.053 second for complete execution. Additionally, our probabilistic generate experiments show promising results for short fuzzing sessions; however, the effectiveness diminishes as the duration of the fuzzing session increases. These findings highlight potential areas for optimisation in Syzkaller's request handling and fuzzing strategies.

To the best of my knowledge, this study represents the first profiling of the updated version of Syzkaller. This new version, pushed in mid-March 2024 [7], includes, as previously mentioned, three types of jobs ("smash jobs", "hints jobs", and "triage jobs") and requests such as `generate`, `mutate`, `minimize`, and collide. Additionally, it features an updated version of the request priority queue. These modifications fundamentally alter the fuzzer's architecture, making the profiling particularly noteworthy.

Chapter 2

Background

2.1 Kernel Fuzzing

First proposed in 1988 [3], fuzzing is a technique used to discover bugs by providing pseudo-random data as input to a program and monitoring for crashes or any unexpected behaviours. The main idea behind fuzzers can be summarised in three steps. First, generate inputs via various methods (modes of operation in Syzkaller's case). Then, feed the generated inputs to the program under test. Finally, monitor the program's behaviour and vital signs (e.g., errors and crashes) in the hope of triggering a bug.

Kernel fuzzers, which are fuzzers specifically designed to target system calls can be traced as far back as 1991 [16]. Operating systems, such as the Linux kernel with its 29.6 million lines of code as of today according to `cloc` [1] on version `6.7` of the kernel, present a vast surface area for fuzzing.

2.2 Syzkaller

2.2.1 Syzkaller's Hierarchy

Syzkaller is currently one of the most widely used system call fuzzers for various kernels. It has been instrumental in finding or fixing more than 6500 bugs in the Linux kernel [26]. The structure of Syzkaller is illustrated in Figure 2.1. It comprises three main components: `syz-manager`, `syz-fuzzer`, and `syz-executor`. The kernel under test runs on a set of QEMU virtual machines, with `syz-fuzzer` and `syz-executor` operating inside these VMs. `syz-manager` is running on the host machine. Each of these components plays a crucial role in the fuzzing process, working together to discover and report kernel bugs effectively.

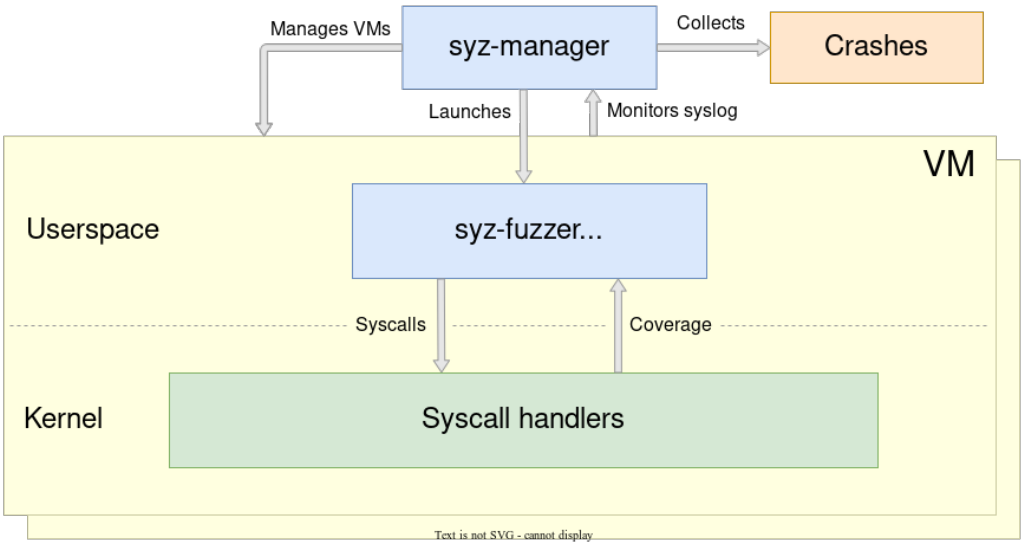
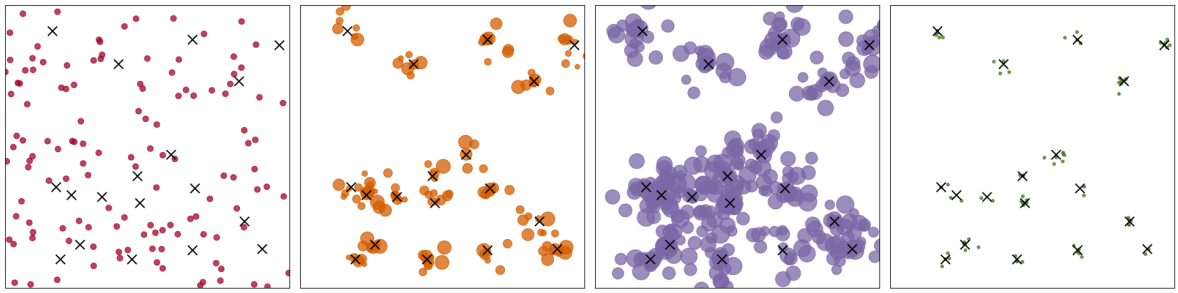


Figure 2.1: Link between Syzkaller's three main components: `syz-manager`, `syz-fuzzer`, and `syz-executor` by Andrey Konovalov [2], one of Syzkaller's top contributors according to Syzkaller's repository metrics

The `syz-manager` is the central coordinating entity in Syzkaller. Its primary roles include orchestrating the entire fuzzing operation, managing multiple instances of `syz-fuzzer`, and ensuring they work together efficiently (e.g., sharing test cases that improve coverage). The manager allocates resources such as the virtual machines where the kernel under test is located, and ensures they are correctly configured for fuzzing. Additionally, `syz-manager` collects crash reports, various stats later displayed on the dashboard, and execution feedback from fuzzer instances. It also keeps detailed logs of all activities happening in Syzkaller overall. This provides a clear overview of the fuzzers' progress at any point in time via Syzkaller's online dashboard collecting crashes, warnings, bugs, and various interesting stats. `syz-manager` also balances the workload among multiple fuzzers. It is setup on the host machine and is written in Go.

`syz-fuzzer` instances are responsible of the core fuzzing process. It generates new test cases using various strategies such as generations, mutations, smashing, and guided mutations described in detail below. When the fuzzer created a sequence of syscalls to experiment with, it sends it to `syz-executor` for execution within the unstable kernel inside of the virtual machine. After inputs are executed, the fuzzer collects feedback on the execution, such as coverage data and crash reports, which are crucial for refining future inputs. Based on this feedback, `syz-fuzzer` generates a new test case from scratch or mutates existing inputs using various mutators to explore new code paths and potential bugs. Additionally, `syz-fuzzer` communicates with `syz-manager` to report progress, receive new tasks, and update the overall status of the fuzzing process. `syz-fuzzer` is written in Go and runs in the VM itself.

`syz-executor` is the component that directly interacts with the kernel under test. It executes



(a) `generate` mode illustration (b) `mutate` mode illustration (c) `smash` mode illustration (d) `hints` mode illustration

Figure 2.2: Coverage discovery for each mode of operation visualised

the inputs generated by `syz-fuzzer` within the kernel environment, handling the actual system calls and monitoring their effects. The executor detects crashes or any abnormal behaviour during execution and captures detailed information about the state of the kernel at the time of the crash. This component provides detailed execution feedback to `syz-fuzzer`, including coverage information, crash logs, and other relevant data that can help in refining future test cases. Furthermore, `syz-executor` ensures that the execution of potentially harmful inputs is contained within a controlled environment, preventing any adverse effects on the host system. The executor is written in C++ and runs inside the VMs alongside `syz-fuzzer`

These components form a robust hierarchical system for kernel fuzzing: `syz-manager` coordinates and oversees the entire fuzzing process from the host stable machine; `syz-fuzzer` generates and alters programs, running within VMs to create and refine inputs based on execution feedback; and `syz-executor` executes the inputs in the kernel environment, detects crashes, and provides detailed feedback for further analysis and input refinement.

2.2.2 Description of `syz-fuzzer`

Candidates

At the beginning of a fuzzing campaign, Syzkaller loads a set of initial known programs called "candidates". These candidates are carefully selected to provide a robust starting point for the fuzzing process. They are pre-generated test cases that cover a variety of system calls and input patterns used to bootstrap `syz-manager` with an initial corpus. The diversity of the candidates helps in covering a wide range of code paths early in the fuzzing process.

Modes of Operation

Syzkaller makes use of four modes of operation to generate and refine inputs: `generate`, `mutate`, `smash`, and `mutate with hints`. Each mode has specific characteristics and excels in different scenarios, contributing to the overall effectiveness of the fuzzing process. Figure 2.2 shows a visualisation of the coverage discovery strategies for all four modes. The kernel under test's basic blocks are conceptualized as being mapped within a 2D square. Black "×" denote candidates, while circles depict a simplified illustration of the coverage achieved through this mode given said candidates.

The `generate` mode is responsible for creating new inputs from scratch. This mode is particularly useful when exploring uncharted areas of the codebase, as it helps in discovering new execution paths. The illustration from Figure 2.2a shows the strategy in detail: `generate` creates new programs from scratch that could end up almost anywhere in the 2-dimensional kernel representation. In `generate` mode, Syzkaller uses predefined man-made templates [25] (also called syscall descriptions) to produce valid programs.

`mutate` mode takes existing inputs and makes slight modifications to create new ones. This approach is based on the idea that small changes to an input might trigger different behaviours in the kernel, potentially uncovering bugs that are closely related to the original input as shown in Figure 2.2b. Syzkaller employs various mutation strategies, such as altering the values of arguments or modifying the sequence of operations. `mutate` mode is efficient when there is already a set of known good inputs that have been proven to reach interesting code paths.

The `smash` mode is an aggressive fuzzing strategy that involves making extensive changes to existing test cases. Unlike the more controlled modifications in `mutate`, `smash` introduces significant alterations through 100 mutations, often resulting in inputs that are drastically different from the originals as in Figure 2.2c. This mode is particularly effective in scenarios where simpler mutations have failed to produce new crashes or interesting behaviour, suggesting that more drastic changes are necessary to trigger hidden vulnerabilities.

`mutate with hints` mode combines the principles of mutation with additional information (called "hints") that guide the mutation process. These hints are derived from various sources, such as previous coverage data, crash reports, or pre-defined edges cases. By adding these hints, Syzkaller can make more informed and targeted mutations as shown in Figure 2.2d.

By leveraging all of these modes of operation together, Syzkaller can adapt its fuzzing strategy to the needs of the testing campaign, ensuring a thorough and effective exploration of the kernel's vulnerabilities. This phenomenon is visualised in Figure 2.3. In reality, each new coverage discovery would trigger a `smash` job at the new location. The visualisations are a means to understand modes used in a vacuum.

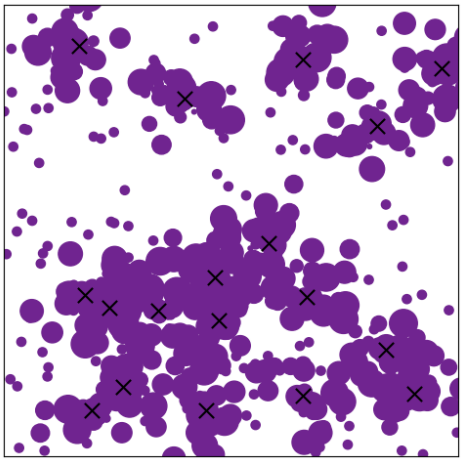


Figure 2.3: Coverage discovery using all four modes combined

Triaging Stage

The triaging stage in Syzkaller is a crucial process that occurs after a potential new coverage discovery, or potential bugs is detected during the fuzzing campaign. The goal of triaging is to filter out false positives, and make sure that only genuine bugs are reported to `syz-fuzzer`.

Internal Mutators

Mutators are used during mutations to alter existing inputs to create new ones. Each mutator performs a specific type of modification to the input, and collectively, they help create a diverse set of test cases.

The `insertCall` mutator adds a new system call anywhere in an existing program. This mutator helps exploring new code paths that involve additional system interactions.

The `removeCall` mutator deletes an random existing system call from the input sequence. This helps in simplifying the input and identifying the minimal set of operations needed to trigger a bug. By iteratively removing calls, Syzkaller can isolate the specific call or sequence of calls responsible for the observed crash or exceptional behaviour.

The `mutateArg` mutator changes the arguments of an random existing system call from the sequence. Arguments can be modified in various ways, such as changing their values, types, or even replacing them with randomly generated data. It helps in uncovering bugs that occur due to invalid or unexpected argument values.

The `splice` mutator combines two existing programs together. It takes the head of the program

currently being mutated, and sticks a random test case from the corpus at the end. This mutator is effective for generating new, potentially interesting input sequences that might not be produced by other mutation strategies.

The `squashAny` mutator applies a combination of mutations to the input sequence. It can involve changing multiple system calls and their arguments in a single operation. The `squashAny` mutator helps in uncovering subtle bugs that might only be triggered by specific combinations of changes.

Jobs and Requests

In `syz-fuzzer`, "jobs" and "requests" are designed to help manage the fuzzing tasks. Jobs are started via goroutines and represent asynchronous tasks, dividing the workload into manageable chunks, that can run independently of one another. There are three types of jobs: "smash jobs", "hints jobs", and "triage jobs". Throughout this paper, we will interchangeably use the terms "smash jobs" and "smash mode", as well as "hints jobs" and "hints mode". A triage job categorises crashes discovered during the fuzzing campaign. It also simplifies programs and incorporates them into `syz-manager`'s corpus, where they can later be used as seeds for new mutations. Requests are smaller tasks that need to be completed (e.g., execute a program on `syz-executor` to test for new coverage). Unlike jobs, requests are blocking operations. Jobs often have to wait on request execution to continue their job.

Pending requests awaiting execution are stored in a priority queue, ensuring that tasks are handled based on their importance. Each request is assigned a priority level, which determines its position in the queue. Higher priority requests are placed at the front of the queue, while lower priority requests are placed further back. For instance, a `generate` request initiated by `syz-fuzzer` has a low priority, whereas a `mutate` request ordered by a `smash` job is given top priority.

Chapter 3

Design

3.1 Overview

Syzkaller's workflow involves several components: `syz-manager`, `syz-fuzzer`, and `syz-executor`. Figure 2.1 illustrates Syzkaller's overall architecture, highlighting how the three components interact during the fuzzing campaign. `syz-manager` coordinates the fuzzing process, `syz-fuzzer` generates test cases, and `syz-executor` runs these test cases on the unstable kernel to detect anomalies. In this paper, we perform two types of experiments that we will call "profiling levels".

The first profiling level, is the "strategy level". It focuses on the profiling of `syz-fuzzer`. We will start with a simple profiling of the vanilla Syzkaller detailed in section 3.3 to establish a baseline and identify bottlenecks. Subsequently, based on the previous results, we will dig deeper in the modes of operation by conducting experiments where we selectively disable modes one by one (i.e., ablation studies in section 3.5) and another set of experiments where we enable only a single mode for each run (i.e., single modes studies in section 3.4). The metrics used to evaluate these experiments include code coverage and the number of crashes detected. These metrics are crucial for understanding the efficiency and effectiveness of different fuzzing strategies, and their detailed descriptions are provided in section 3.2. Each of the experiment and basic profiling we do will be based on 48 hours or 72 hours runs depending on the experiment. We will run each experiment 4 times and aggregate the obtained data before plotting the results using the Jupyter Notebooks we created.

The second profiling level, named "implementation level", is an experiment on `syz-executor`. Unlike the first level, where we profiled by instrumenting the Go code, this experiment involves directly logging into the QEMU virtual machine hosting `syz-executor`. We will perform a performance analysis using the Linux `perf` command [9] to gain insights into the execution behavior of `syz-executor`. This method allows us to capture detailed performance metrics and identify

potential bottlenecks in the execution process.

With these two levels of profiling, we aim to provide a solid base for the analysis of Syzkaller's performances and effectiveness. The insights we will gain with these experiments will help us design experiments aiming to enhance Syzkaller's original toolkit.

3.2 Profiling metrics

We used the notion of coverage to compare different runs since it is a direct indicator of how thoroughly the fuzzer explores the program. Code coverage indicate how much of the code is explored during testing. Higher coverage implies that a larger portion of the code has been tested, increasing the likelihood of identifying bugs and vulnerabilities. It also allows for easy comparison between different fuzzing tools.

The coverage is computed using KCOV [13] by tracing coverage points inserted into the object code by the compiler [24]. The particular metric we are using to track the fuzzer's performance is the number of individual basic blocks discovered in the kernel under test during the experiment.

We also monitored the number and source of the bugs triggered during our profiling and experiments to ensure that we were not consistently identifying the same type of bug. A bug cannot be triggered without being covered first by a system call, hence why we chose coverage as the main metric.

3.3 Profiling Vanilla Syzkaller

We aim to understand Syzkaller's inner workings and identify which parts of the code are resource-intensive. To achieve this, we will analyse several key aspects: the evolution of the fuzzer over time (e.g., differences in behaviour between 1 hour and 48 hours into the fuzzing campaign), Syzkaller's modes, mutators, and the new additions we made (the `shuffle` mutator and the probabilistic generate experiment). We expect the modes and mutators to be the largest bottlenecks since they are omnipresent in Syzkaller. Mutators, in particular, are used in three out of the four modes of operation, making them critical to the fuzzer's performance. By examining how these features perform, how they combine with other components, and how much each contributes, we hope to gain a comprehensive understanding of Syzkaller's efficiency and effectiveness.

We start by profiling each of Syzkaller's four modes of execution: `generate`, `mutate`, `mutate with hints`, and `smash`. In Figure 3.1 is shown a high-level flow of how `syz-fuzzer` works. At each iteration, the fuzzer checks the priority queue of pending requests to decide which mode to activate. If the queue is empty, the fuzzer chooses probabilistically between the `mutate` mode (95%

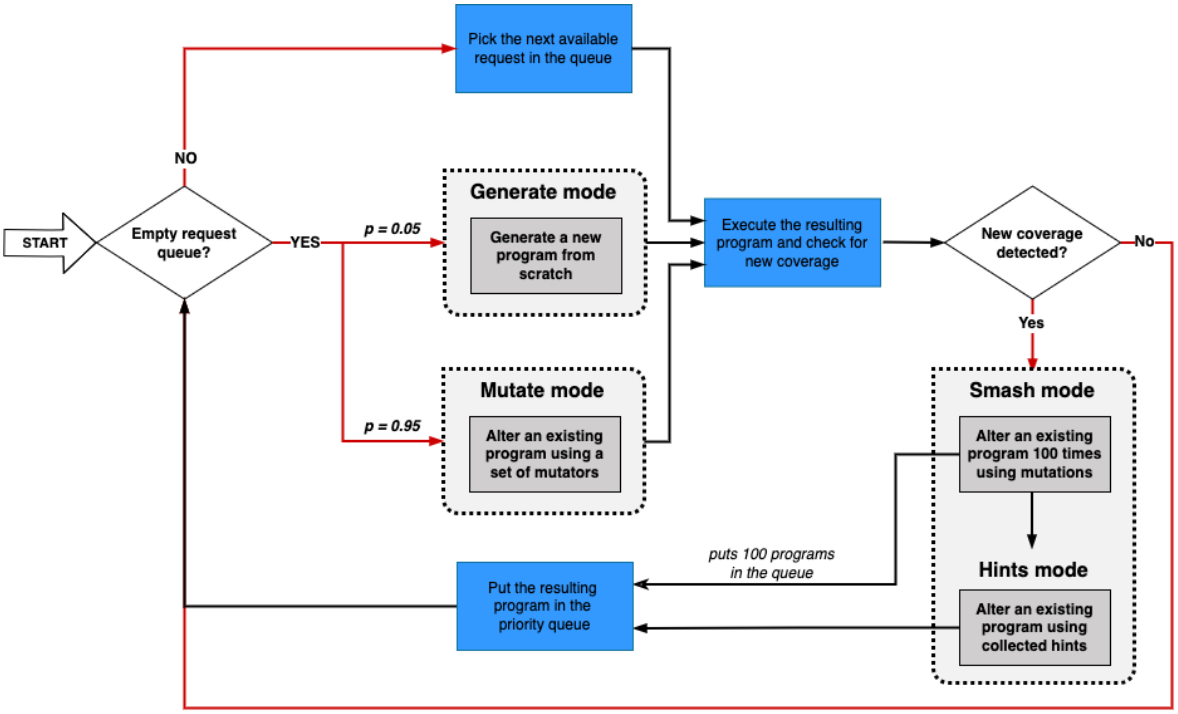


Figure 3.1: Representation of the interactions between `syz-fuzzer`'s modes of operation

probability) and the `generate` mode (5% probability). The resulting program is then executed by the fuzzer and checked for new coverage. If no new coverage is discovered, the fuzzer starts again, checking the queue for the next request to execute. On the other hand, if new coverage is found, the resulting program is smashed to find new paths, ultimately adding 100 new mutations to the queue. The `smash` job triggers a `hints` job that alters the program with edge cases either pre-defined in Syzkaller or from previous iterations. Once the programs are in the queue, the fuzzer starts again, checking the queue for the next input to execute. Note that in the case of the `smash` job, requests are sent to the queue sequentially; in other words, the nth request generated by the `smash` mode only enters the queue after the $n-1$th request finishes executing.

Mutate is a popular mode as it can be initiated either organically by the fuzzer with a `mutate` request or by a `smash` job. When the priority queue used to select the next request to execute is empty, the fuzzer issues a `generate` request with probability $p = 0.95$ or a `mutate` request with probability $q = 1 - p$. A `smash` job essentially involves mutating the target program (using the same `mutate` mode mentioned previously) 100 times. This difference in the origin of the request is why we will distinguish both instances of `mutate`.

During the fuzzer's execution, we record the number of times each mode is triggered and the total time spent executing each mode. Additionally, we record how many basic blocks are discovered by each mode and whether the coverage increases for a specific execution of a mode, as well as

whether this increase originates from a minimize request. A minimize request takes a program and tries to remove any unnecessary information. This helps keep the corpus small in terms of overall size and reduces the noise in the stored programs. For each request, we track how much time it spends in Syzkaller's priority queue. For each mutator, namely `insertCall`, `removeCall`, `mutateArg`, `splice`, and `squashAny`, we record how many times each of these mutators is called.

We aim to understand which modes are the most frequently used and which ones contribute the most to overall coverage. By analyzing the execution frequency and the time spent on each mode, we can identify which modes are the most dominant and effective in discovering new coverage. Moreover, by timing how long each mode runs, we can identify which modes are the most resource-intensive. This helps us pinpoint potential bottlenecks in the fuzzing process and determine if certain modes are consuming a disproportionate amount of resources relative to their contribution to coverage. Based on the collected data, we can then later make informed decisions on how to optimise the fuzzing strategy. For instance, if certain modes or mutators are found to be less effective, we can adjust their probabilities or refine their implementation to improve overall fuzzing efficiency.

Running this simple preliminary profiling on the fuzzer's building blocks allowed us to identify an execution mode that was particularly often used and a mutator that could often fail, potentially leading to wasted cycles. We realised that the `smash` mode was used extensively during the initial stages of the fuzzing campaign. This prompted us to focus on this particular mode. The `smash` job starts by scheduling a `hints` job asynchronously. At the same time, it begins mutating the input program 100 times. Finally, it tries to incorporate some faults into the program and tests for new coverage. We will take a deeper look at the results related with the `squashAny` mutator in subsection 5.2.5.

Running the simple preliminary profiling on the `generate` mode reveals that it is heavily underused at the start of the fuzzing process. This occurs because the fuzzer queue is almost never empty in the early stages. However, we notice that the few `generate` requests made in the early stages often lead to new coverage (74.6% of the time). This observation inspires our probabilistic generate experiment in section 3.7, which aims to increase the frequency of `generate` mode usage during the initial stages of fuzzing.

3.3.1 Deep Analysis of `smash` Mode

The `smash` mode is by far the most time consuming and complex of the modes. A new `smash` job is started every time new coverage is discovered. This substantial predominance of the `smash` mode prompted us to conduct a more thorough analysis of its operations.

3.3.2 Deep Analysis of `squashAny` Mutator

Most mutators can modify a program regardless of the nature of the syscalls inside of said program. However, the mutator `squashAny` deviates from this norm. `squashAny` selects a random complex pointer from a given program and compresses its arguments into an `ANY` type. If the program lacks any complex pointers, the mutator will fail. We chose to scrutinize this mutator because it is the only one susceptible to failure depending on the input program's nature. Additionally, it has a 20% chance of being chosen with each mutation, which is significant and may result in wasted cycles if the failure rate is high.

3.4 Single Mode Study

To understand the specific contributions of each fuzzing mode, we will conduct single mode studies. In these experiments, Syzkaller is configured to operate with only one mode active at a time, allowing us to observe the unique characteristics and effectiveness of each mode without interference from the others.

We will run Syzkaller with only one of the four modes (`generate`, `mutate`, `mutate with hints`, or `smash`) enabled for each experiment. We will measure the code coverage and the number of bugs identified over a 48-hour period. The environment and resource constraints were kept the same as in the simple vanilla Syzkaller profiling experiment portrayed in section 3.3.

These insights should help us understand the strengths and weaknesses of each mode, guiding further optimisations and experiments.

3.5 Ablation Study

Ablation studies – along with single mode studies – are a critical methodology used to understand the contribution and importance of each component within a complex system. The key difference is that ablation studies help us understand the relative importance and contribution of each mode within the full system, highlighting how the absence of a mode affects overall performance. In contrast, single mode studies focus on the standalone performance of each mode without the influence of the others. Thus, ablation studies give us a more holistic view of the interplay and dependency between modes, which single mode studies do not provide.

In the context of our research, we will perform ablation studies to evaluate the individual impact of each of Syzkaller 's four modes. By selectively disabling one mode at a time and observing the results, we aim to determine the significance of each mode in the overall fuzzing process.

By disabling one mode at a time, we can measure how the absence of that mode affects the system's performance in terms of code coverage and bug discovery. This approach helps us understand which modes are crucial for achieving high coverage and which modes might be less effective or even redundant.

We will conduct a series of experiments where Syzkaller is run multiple different times, each time with one of the four modes disabled. The experiments are designed as follows:

- without `generate` mode: Syzkaller cannot generate new sequences of syscalls (programs) from scratch;
- without `mutate` mode: Syzkaller cannot modify existing programs using mutators;
- without `mutate with hints` mode: Syzkaller cannot guide the mutations using hints;
- without `smash` mode: Syzkaller cannot perform aggressive mutations on existing programs.

3.6 Shuffle Mutator

To further enhance the variability of syscall sequences composing each program, we introduced a new mutator we called `shuffle`. `shuffle` is a new mutator that randomly shuffles the order of system calls in a given sequence. Shuffling system calls can create novel sequences that explore different execution paths, potentially uncovering bugs that other mutators miss (or at least did not uncover yet). The uniqueness of `shuffle`, is that by rearranging system calls, the fuzzer may trigger subtle bugs and race conditions that depend on specific call sequences.

The main idea behind the `shuffle` mutator is based on the hypothesis that certain vulnerabilities may only be triggered when syscalls are executed in a specific order. Standard mutators in Syzkaller, such as `insertCall` or `removeCall`, focus on slightly modifying the content and overall structure of syscall sequences, but they might not adequately explore the variations in execution order. By shuffling the order of syscalls within a sequence, the `shuffle` mutator can expose new interactions between syscalls that could potentially lead to bugs that have not yet been encountered.

For example, a particular bug might only manifest when syscall `A` is followed by syscall `B`, but if the sequence is always generated or mutated in a fixed order, this specific interaction may not get tested easily.

However, randomly shuffling syscalls might generate sequences that are less likely to trigger "happy path" behaviour or reach deep into the kernel's code. This can result in a higher number of ineffective tests that do not contribute to coverage.

Moreover, `shuffle` may break dependencies, leading to invalid sequences and wasted fuzzing

efforts. For instance, Listing 1 shows an example of a simple program (excluding error handling for simplicity), and a shuffled version that breaks system call dependencies.

```
const char *data = "Hello Sailor!";

/* original version */
int fd = open("file.txt", O_WRONLY | O_CREAT, 0644);
write(fd, data, strlen(data));
close(fd);

/* shuffled version */
write(fd, data, strlen(data)); // fd uninitialized => invalid operation
close(fd); // fd uninitialized => invalid operation
int fd = open("file.txt", O_WRONLY | O_CREAT, 0644); // file is opened, but too late
```

Listing 1: Example of program reordering breaking syscall dependency

The experiment is worth conducting as it can provide valuable insights into the benefits and drawbacks of introducing more randomness into the fuzzing process. If successful, it could be a valuable addition to Syzkaller's arsenal of mutators.

3.7 Probabilistic Generate Experiment

To introduce an element of randomness and attempt to enhance Syzkaller's exploration capabilities, we will implement a probabilistic generate experiment. This experiment aims to randomly trigger a `generate` request with a certain fixed probability when selecting the next request to execute instead of picking the intended one from the queue.

Probabilistic generate is conceived to address a key challenge in fuzzing: the risk of the fuzzer getting stuck in a less productive part of the code. By introducing a probabilistic element that can easily uncover widely different parts of the kernel, we aim to increase the diversity of syscall sequences, encouraging the fuzzer to explore previously untested areas of the kernel.

The experiment can help prevent the fuzzer from getting stuck mutating programs that are already well explored, ensuring a more comprehensive exploration of the target codebase. In theory, introducing new sequences at random intervals can result in higher overall code coverage, as more unique paths through the code are exercised. Determining the optimal probability for triggering the `generate` mode can be challenging. If the probability is too low, the benefits of increased exploration may not be realized; if too high, it can lead to the fuzzer performing shallow searches (superficially exploring a myriad of widely different code areas).

As an analogy, mutation-based mutators could be compared to exploring the kernel under test in a DFS (depth-first search) manner, where the fuzzer focuses on investigating surrounding areas

well before branching out. On the other hand, `generate` mode can be seen as exploring the kernel using a BFS (breadth-first search) approach. The BFS approach is useful for quickly achieving a burst of coverage, as generating new programs from scratch is likely to uncover new, unexplored areas. This is due to the kernel's sheer size, the large amount of syscalls available, and the myriad of possible syscall arguments. However, to uncover deep-rooted bugs within the kernel, we also need to examine further the various discovered areas. Overall, it is a trade-off, and we should strive to find a balance between these approaches.

3.8 Dependency Analysis Removal

Syscalls in operating systems often depend on each other. For Syzkaller to generate meaningful and effective fuzzing inputs, it must ensure that the system calls it sequences together can logically follow one another. Dependency analysis is used to maintain the state and context required for subsequent system calls. For instance, a `read` syscall typically depends on a previous `open` syscall. Without satisfying these dependencies, many syscalls would fail due to invalid context. Dependency analysis is used three times in `syz-fuzzer`: during the execution of mutators `insertCall`, `squashAny`, and `mutateArg`.

When inserting a new call with `insertCall`, its parameters must be correctly initialized and valid within the program. For example, if the inserted call requires a file descriptor, dependency analysis ensures a valid file descriptor is available from a prior open call.

`squashAny` removes redundant or unnecessary calls from a sequence to simplify it while trying to maintain the sequence's ability to trigger a bug. Dependency analysis ensures that removing a system call does not break the dependencies required by subsequent calls in the sequence. For example, if a removed call creates a resource needed later, dependency analysis identifies this and prevents such removals. By removing only those calls that are genuinely redundant, dependency analysis helps in maintaining an effective and minimal sequence that can still trigger the bug.

`mutateArg`'s job is to alter the arguments of existing system calls in a sequence of syscalls. Analysis for `mutateArg` ensures that the mutated arguments do not lead to invalid resource usage. For example, if a system call uses a pointer to a memory region, the mutation must ensure the pointer is valid and points to a properly allocated memory region. The analysis also makes sure that the new arguments generated by the mutation are valid and satisfy the dependencies within the context of the sequence. For instance, if a system call requires a buffer of a certain size, dependency analysis ensures that the mutated argument meets this requirement. The `read` syscall is an example of such syscall.

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

When fuzzing with Syzkaller, the `mutateArg` mutator might change the `nbytes` argument of a `read` system call. The buffer `buf` should be allocated (by the user) and large enough to hold the number of bytes specified by `nbytes`. If the buffer is not large enough to hold `nbytes` bytes, it can lead to buffer overflows.

By disabling dependency analysis, we can assess the impact of this crucial component on the effectiveness of Syzkaller's fuzzing campaigns. We can identify if certain types of bugs are missed or if the absence of dependency analysis leads to an increase in the discovery of shallow, easily detectable bugs compared to deeper, more elusive ones.

3.9 Profiling `syz-executor`

This new experiment focuses on the `syz-executor` component. We aim to gain a deeper understanding into its execution behaviour and performance characteristics. Unlike the previous experiments at the strategy level, this experiment involves directly logging into the QEMU virtual machine hosting `syz-fuzzer` and `syz-executor`.

We will use the Linux `perf` command [9] to collect hardware events including CPU cycles and caches misses [4]. After having recorded a decent amount of hardware events, we will generate flame graphs [8] using the recorded data.

The goal of this experiment is to identify potential performance bottlenecks in `syz-executor`. By understanding how `syz-executor` utilizes system resources during the execution of test cases, we can uncover inefficiencies that may hinder the overall fuzzing process.

We will have a total of three samples: the first sample will be taken at the fuzzer creation (when candidates are being added to the corpus) and a second sample will be collected after 2 hours of fuzzing. Lastly, a final sample will be obtained after 24 hours of fuzzing. We will run the `perf-record` for 60 seconds at each occasion and then plot the result in flame graphs.

Chapter 4

Implementation

All the code related to profiling Syzkaller can be found on our Syzkaller fork [21]. The Jupyter Notebooks and Python scripts used to analyse and plot fuzzer results are available in the repository dedicated to this research project [20]. The Syzkaller profiler we implemented can be activated using the build tag `profiling` or simply by running the build script `make_manager.sh` provided in the repository.

A complete list of build tags can be found below:

- `profiling`: enables the profiler we created for this project;
- `log_pc`: enables fuzzer logs. By default, only manager logs are allowed because fuzzer logs often clog the log file with redundant and highly specific information. Fuzzer logs are useful when you want to analyse the basic blocks reached by each mode;
- `prob_generate`: enables the probabilistic generate experiment;
- `disable_dependency_analysis`: disables Syzkaller's automatic dependency analysis.

4.1 Basic Profiling Tools

We implemented simple profiling tools using basic Go primitives. We used integer counters to keep track of how many times a mode or mutator was executed. To measure the duration of specific code sections, we utilised the `time.Duration` type from the Go `time` package. These profiling tools are activated only when the `profiling` build tag is used at compilation time.

Syzkaller already has a built-in dashboard that tracks various metrics such as uptime, number of crashes, or number of completed triage jobs. We extended this dashboard to include our custom

metrics. Additionally, we configured the system to dump the entire dashboard into a log file every 30 seconds, allowing us to build detailed time series for analysis.

To integrate new data into the dashboard, we mainly used the fuzzer object. This made it easier to add our custom metrics to the dashboard. For metrics like mutator counts, which are deeper in the code, we used observer patterns. These observers collected information during the mutation process and reported it back to the fuzzer once the mutation was done. This method helped us gather comprehensive and accurate data for our profiling needs.

4.2 Coverage Analysis

When the fuzzer is running, it is based on an infinite loop fetching the next input at each iteration. This input can take different forms:

- It may be a combination of a job (which encodes the priority of the request in the queue) and a request placed in the queue using the `fuzzer.exec(job, request)` function;
- If the queue of requests awaiting execution is empty, Syzkaller generates either a new program and executes this request or mutates a random program from the corpus and executes this request;
- Alternatively, it could consist of a group of candidates retrieved from the manager. These candidates represent programs fetched from an online repository and include additional information, such as whether they have been minimized or affected by smash detection.

These requests come from different parts of the code. We wanted to keep track of which part of the code, like which mode, created each request. This would help us determine whether the mode is efficient by looking at the results for each request the mode ran. To do this, we added a field in the `Request` struct to keep track of which part of the code initiated the request.

```
type Request struct {
        // ...
        requesterStat string
}
```

4.3 Ablation Study

We initially tried to implement the ablation study using build tags, but we quickly realised it was cumbersome. This approach would require a lot of duplicated files to achieve the desired result.

Instead, we decided to use a configuration file, "ablation_configuration.json", already present in the repository [21]. This file is read each time a new fuzzer instance is created. By default, all modes and mutators are active. You can disable any mode or mutator by changing a boolean in this config file and compiling the code with the `profiling` tag.

We also used this system for the single mode studies we developed. If we want to perform a single mode study on a particular mode of execution, we just disable the other three modes in the json configuration file.

4.4 Deep Analysis of `smash` Mode

In our experiment, we decided to instrument the `smash` job in Syzkaller by adding timers and counters to profile this mode. This decision was made to better understand the internal workings and time distribution of the `smash` job, as well as to identify potential inefficiencies. By breaking down the execution times of different components within the `smash` job, we will be able to pinpoint which parts are most time-consuming and explore ways to optimise the overall process. This profiling should help us learn how the various stages of the `smash` job contribute to its total duration and how improvements could enhance the fuzzer's performance.

4.5 Shuffle Mutator

The implementation of the `shuffle` mutator is similar to other existing mutators. It does not make use of Syzkaller's dependency analysis, but instead tries to shuffle a given program until it finds a valid sequence up to a certain fixed limit of tries set to 16 by default. Validation includes checking the correct usage of system calls, appropriate argument values, and ensuring dependencies between system calls are maintained.

When choosing a mutator to use, Syzkaller distributes mutation energy following the information from diagram 4.1. It starts at the blue box and calls an pseudo-arbitrary amount of mutators and continues picking mutators successively until the mutation completes.

The probability of calling each mutator is shown in Table 4.1. The absolute probability corresponds to the probability of choosing the mutator when given the opportunity (i.e., in the diagram, when Syzkaller did not choose `squashAny`, there is a 1% chance to choose `splice`). The relative probability represents the overall chance of choosing a certain mutator for one mutation iteration. Thus, the relative probability of choosing the `splice` mutator is the conditional probability of choosing `splice` given that `squashAny` was not chosen.

We will settle with an absolute probability of 1% for the `shuffle` mutator, the same as `splice`,

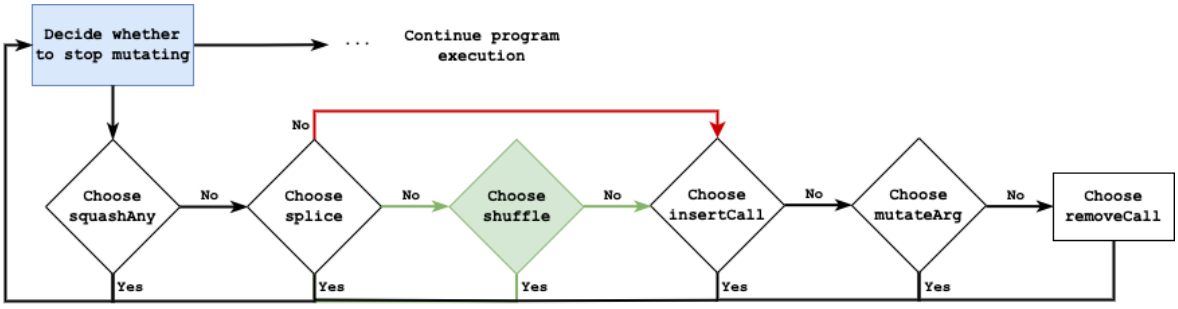


Figure 4.1: Syzkaller mutator choice process during mutations. The red path only exists when the `shuffle` mutator is disabled. The green path only exists when the `shuffle` mutator experiment is running

Mutator	Absolute Probability		Relative Probability	
	no `shuffle`	`shuffle`	no `shuffle`	`shuffle`
`insertCall`	64.52%	64.52%	51.10%	50.59%
`mutateArg`	90.91%	90.91%	25.55%	25.29%
`removeCall`	100.0%	100.0%	2.55%	2.53%
`splice`	1.00%	1.00%	0.80%	0.80%
`squashAny`	20.00%	20.00%	20.00%	20.00%
`shuffle`	–	1.00%	–	0.79%
			100%	100%

Table 4.1: Comparison of mutation energy assignation with and without the `shuffle` mutator

because it is similar to the `splice` mutator in the sense that it heavily alters the original program. We do not want to call this mutator on every mutation.

4.6 Probabilistic Generate Experiment

The concept behind this experiment is straightforward: at any given point, instead of proceeding to the next request in the priority queue, Syzkaller has a 0.1% chance to switch to `generate` mode and create a new program from scratch. This probabilistic trigger is intended to inject fresh sequences into the corpus, breaking the potential narrow discoveries of the mutations and exploring brand new execution paths earlier in the execution. We chose a probability of 0.1% to trigger the probabilistic generate experiment. This probability was selected to strike a balance between maintaining the deterministic sequence and introducing enough randomness to enhance exploration without overwhelming the system with too many new sequences.

4.7 Dependency Analysis Removal

Dependency analysis can be turned off using the `disable_dependency_analysis` build tag. To implement this experiment, we simply skipped the analysis and let Syzkaller's `insertCall`, `squashAny`, and `mutateArg` unrestricted.

4.8 Profiling `syz-executor`

Syzkaller is supposed to have an easy way to create QEMU base disk images [5]. I however could not make it work on the Debian-based virtual machine. We will explain in detail the steps necessary to use `perf` on QEMU. Build a disk image (e.g., simply `./create_image` without the `-add-perf` flag) and run the virtual machine (see [5]). `ssh` into the emulator using the public key produced by the `./create_image` script. Then, make sure to establish internet connection on the emulator by adding a new entry in `/etc/hosts`, namely "`127.0.1.1 \t syzkaller`" separated by a tabulation character. Install the package `linux-perf` on Debian via `apt`.

Exit the virtual machine and start Syzkaller in debug mode using `-debug` when starting the manager. When Syzkaller is up and running, `ssh` into the running emulator using the port provided by `syz-manager`, start the `perf` recording and generate the flame graph [8]. Finally, retrieve the graph using `scp`.

4.9 Analyzing Fuzzer Logs

After the execution of a particular run finishes, we get a log file containing logs from the manager and the fuzzer. To analyse these logs, plot them, and create tables for different results, we created different Python scripts and Jupyter notebooks available in the project's repository [20].

The notebook `0x00-simple-analyser` extracts and plots basic Syzkaller metrics, such as information on the popularity and efficiency of modes and mutators.

The `0x01-fuzzer-behaviour` notebook helps us understand the continuous evolution over time of important specific metrics we chose, as well as comparing the efficiency of different modes of execution over time.

The notebook for single modes analysis, `0x02-single-modes`, inspects runs restricted to a single mode of execution. It is mainly utilised to compare a set of runs using a single mode only with vanilla Syzkaller.

The ablation study notebook `0x03-ablation-modes` analyses runs where one of the four modes

of operation was disabled. This is primarily used to compare coverage and modes popularity for a set of runs using three out of the four modes with vanilla Syzkaller.

For the probabilistic generate experiment, we have created a dedicated Jupyter notebook called `0x04-probabilistic-generate` to analyse runs and mainly compare the relative coverage reached versus vanilla Syzkaller.

The deep `smash` and `squashAny` notebook `0x05-deep-analysis-smash-squashany` is used for a detailed analysis of the `smash` mode's time breakdown and the success rate of the `squashAny` mutator.

The `shuffle` mutator notebook (`0x06-mutator-shuffle`) was developed to analyse the performance, popularity, success rate, and efficiency of our custom `shuffle` mutator.

Lastly, the notebook `0x07-no-dependency-analysis` compares various selected metrics between the experiment and vanilla Syzkaller.

Chapter 5

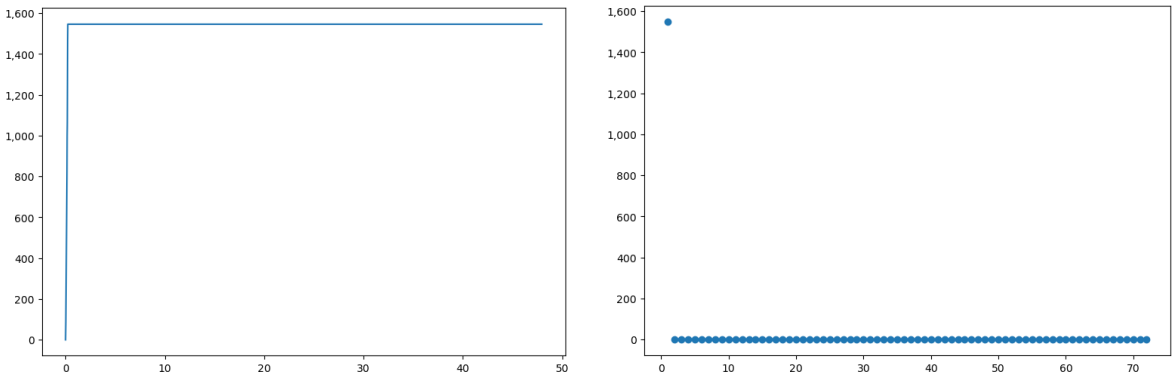
Evaluation

5.1 Experimental Setup

We will run our experiments on a machine with an two Intel(R) 14-core Xeon(R) CPUs E5-2680 v4 2.40GHz CPUs and 256GB of memory where we will use docker containers to effectively profile Syzkaller and conduct our experiments on isolated and reproducible environments. Each container is individually set up with two cores and 2 GB of memory. These limits were chosen to simulate constrained scenarios, allowing us to run multiple experiments simultaneously without exhausting system resources. Another reason for the reasoning behind this limit is that it effectively meets the needs of Syzkaller and is commonly used in evaluations of Syzkaller-based kernel fuzzers [29] [28] [11]. Using the same hardware configuration for all tests also ensures that performance differences are due to the fuzzer itself, and not underlying resource variations.

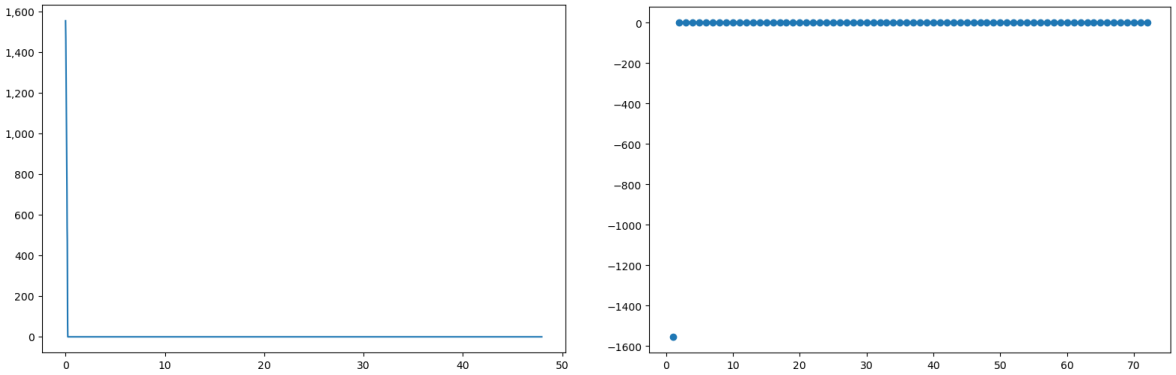
Inside each docker container, we install all necessary dependencies for Syzkaller and used the v6.7 release of the Linux kernel [14]. We run a setup script to ensure that all containers are configured consistently, minimizing discrepancies that could skew the results. With these restrictions, we aim to see how Syzkaller's various modes and mutators perform under pressure, providing valuable insights into their efficiency and effectiveness.

To maintain reproducibility, we will use version control for most scripts and keep them in a custom Syzkaller repository fork [21]. We also will keep detailed logs of each experiment. This thorough documentation will enable us to repeat experiments as needed and verify our results, enhancing the reliability of our findings.



(a) evolution over time (b) relative increase per hour

Figure 5.1: Evolution of the “candidates” metric over a 72 hours period



(a) evolution over time (b) relative increase per hour

Figure 5.2: Evolution of the “triage queue” metric over a 72 hours period

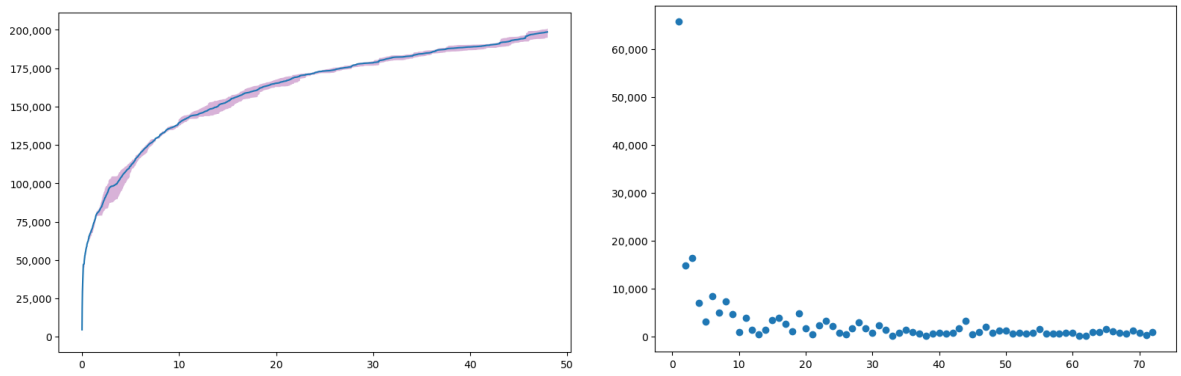
5.2 Profiling Vanilla Syzkaller

5.2.1 Evolution of Syzkaller’s Behaviour over Time

Evolution of Key Metrics

To best understand the evolution of the fuzzer’s behaviour at various time windows, we analysed the performance of the unaltered version of Syzkaller by monitoring several key metrics over time. The following figures present these metrics’ evolution. Figures on the left illustrates the cumulative number of calls for selected metrics from the fuzzer and the manager, while figures on the right displays the hourly contributions to these metrics (each sample represent an hour of fuzzing).

The metrics “candidates” and “triage queue” exhibit an inverse relationship, which is evident in



(a) evolution over time

(b) relative increase per hour

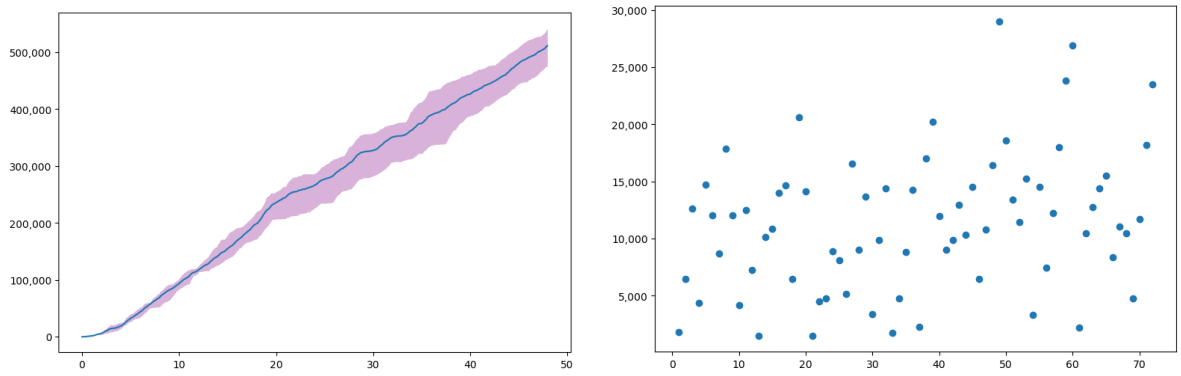
Figure 5.3: Evolution of the “coverage” metric over a 72 hours period

both figures 5.1 and 5.2. During the initial 15 minutes of the fuzzing campaign, the fuzzer actively fetches candidates from a pre-determined online repository. These candidates are specifically designed to bootstrap the fuzzing process, leading to almost guaranteed new coverage discoveries. This influx of candidates saturates the triage queue initially. After this initial period, no further candidates are fetched, leading to a less congested triage queue as not all subsequent programs trigger new coverage. This behaviour is expected because the initial burst of candidate fetching is a built-in strategy of Syzkaller to quickly explore a broad range of inputs.

The “coverage” metric, as seen from the manager’s perspective, shows a distinctive pattern. Initially, in Figure 5.3a, there is a significant spike in coverage at the start of the fuzzing campaign, which then gradually slows down. This initial burst is expected because the fuzzer quickly discovers and covers many new code paths early on, primarily due to the use of candidates. As time progresses, the rate of new discoveries decreases because the fuzzer has already explored the more easily reachable parts of the code (see Figure 5.3b). This slowing down is common when fuzzing and matches our understanding of how coverage typically grows.

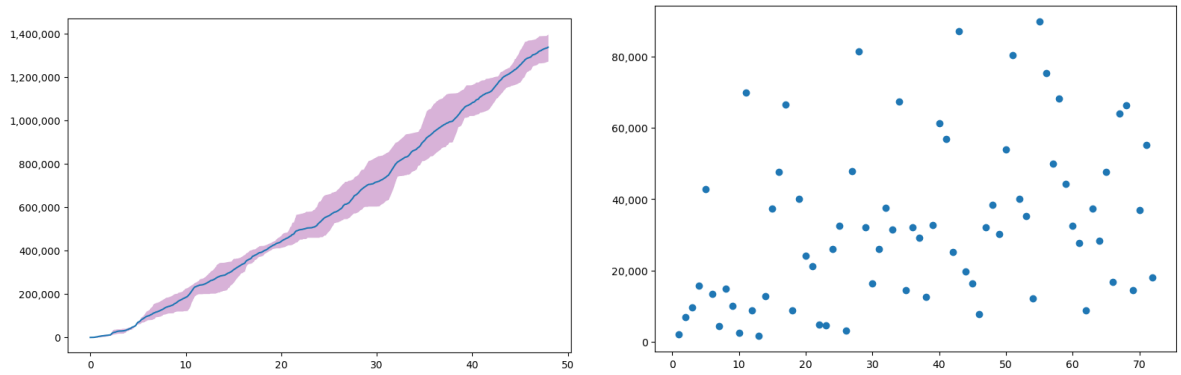
The metrics for “`smash` jobs” (exec smash in Figure 5.4) and “`hints` jobs” (exec hints in Figure 5.5) follow almost identical trajectories, which is explained by the way these jobs are implemented in Syzkaller. Each `smash` job triggers a `hints` job, and each `hints` job can only be created by a `smash` job. This cyclic dependency means that the two types of jobs are directly linked, resulting in their closely aligned progression over time. This behaviour is expected because the design of Syzkaller dictates that hints are derived from the results of smash operations to refine the test cases further.

The “exec total” metric (see Figure 5.6) represents the cumulative execution count across all four modes, along with minimize, collide, and candidates. This metric shows a steady linear increase over time, reflecting the ongoing and consistent activity of the fuzzer. This steady growth is anticipated because the fuzzer continuously processes new inputs, maintaining a regular pace of executions.



(a) evolution over time (b) relative increase per hour

Figure 5.4: Evolution of the “exec `smash`” metric over a 72 hours period

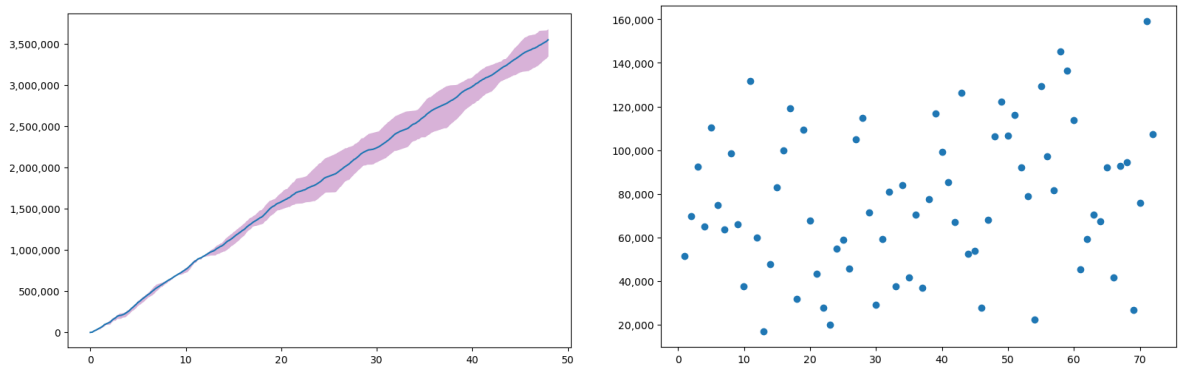


(a) evolution over time (b) relative increase per hour

Figure 5.5: Evolution of the “exec `hints`” metric over a 72 hours period

The metrics “exec fuzz” and “exec gen” illustrate the number of `mutate` and `generate` requests, respectively, that are directly ordered by the fuzzer when the request queue is empty. “Exec fuzz” accounts for 95% and “exec gen” for 5% of these requests. Both metrics display similar trends in their evolution (Figure 5.7a and Figure 5.8a) and their hourly contributions (Figure 5.7b and Figure 5.8b). This similarity arises because both types of requests are triggered by the same event (namely request queue being empty) and are chosen probabilistically. Given a sufficiently large sample size, we expect both curves to have the same shape due to their shared triggering mechanism. This observed behaviour confirms our expectation that the probabilistic nature of these requests will result in similar overall patterns.

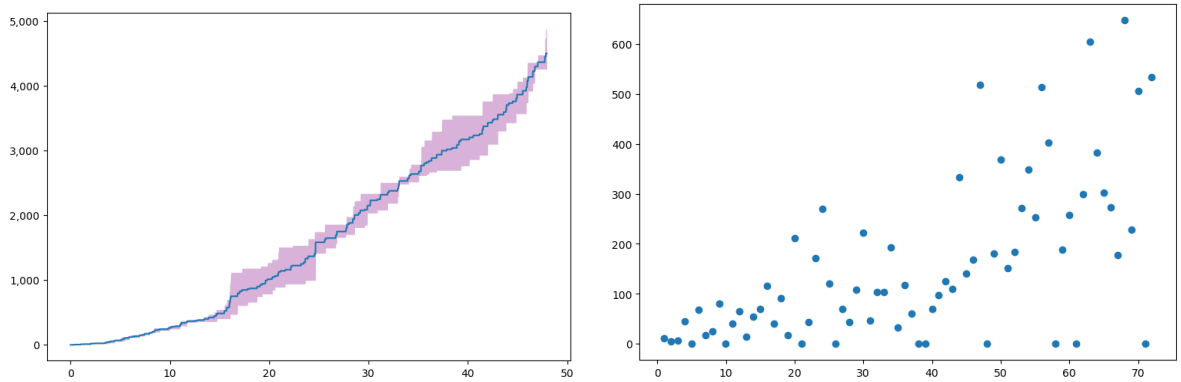
The “rpc traffic (MB)” metric measures the volume of data transferred between fuzzer instances and the manager. This metric shows a gradual increase over time, which reflects the growing volume of communication required to manage and coordinate the fuzzing activities effectively. As more test cases are generated and results are processed, the amount of data exchanged naturally increases as



(a) evolution over time

(b) relative increase per hour

Figure 5.6: Evolution of the “exec total” metric over a 72 hours period



(a) evolution over time

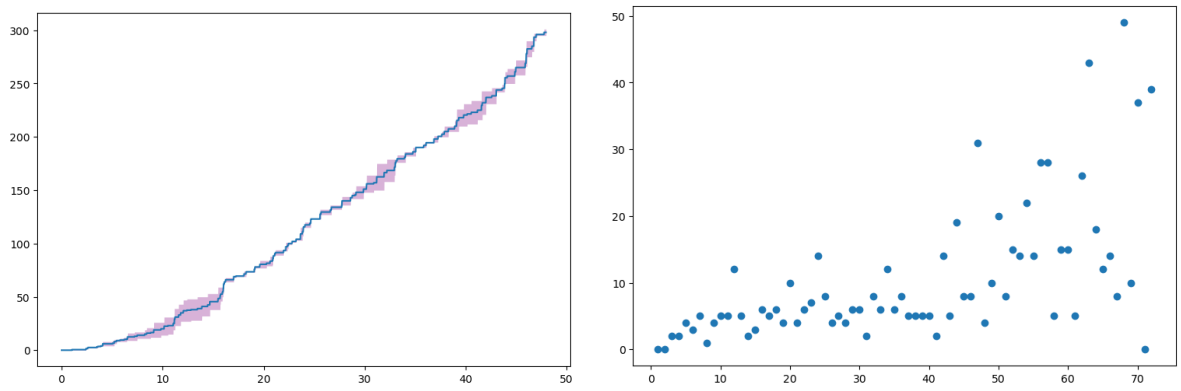
(b) relative increase per hour

Figure 5.7: Evolution of the “exec fuzz” metric over a 72 hours period

shown in Figure 5.9b.

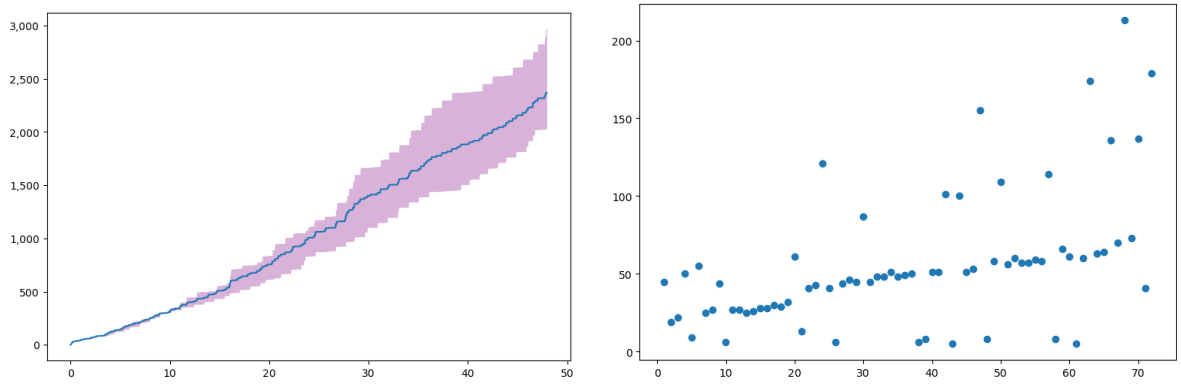
When a new test case is found to increase coverage, the triage job minimizes it before adding it to the corpus. Mutations then occur on these reduced programs. Over time, as the fuzzer discovers more complex bugs that involve interdependencies between multiple syscalls, the average number of calls in a sequence from the corpus increases. Consequently, mutations ordered by the fuzzer occur on, on average, larger programs, and `smash` jobs are triggered on these larger programs as well, explaining the increase in traffic.

Finally, the “exec triage” metric in Figure 5.10 measures the number of triage jobs executed by the fuzzer. Triage jobs are used to validate and minimize the test cases that have resulted in crashes or coverage increases. This process helps confirm that the behaviour is reproducible and reduces the test cases to the smallest form that still triggers the issue. A steady rate of triage jobs indicates that the fuzzer is continuously verifying its findings and ensuring that the discovered issues are



(a) evolution over time (b) relative increase per hour

Figure 5.8: Evolution of the “exec gen” metric over a 72 hours period



(a) evolution over time (b) relative increase per hour

Figure 5.9: Evolution of the “rpc traffic (MB)” metric over a 72 hours period

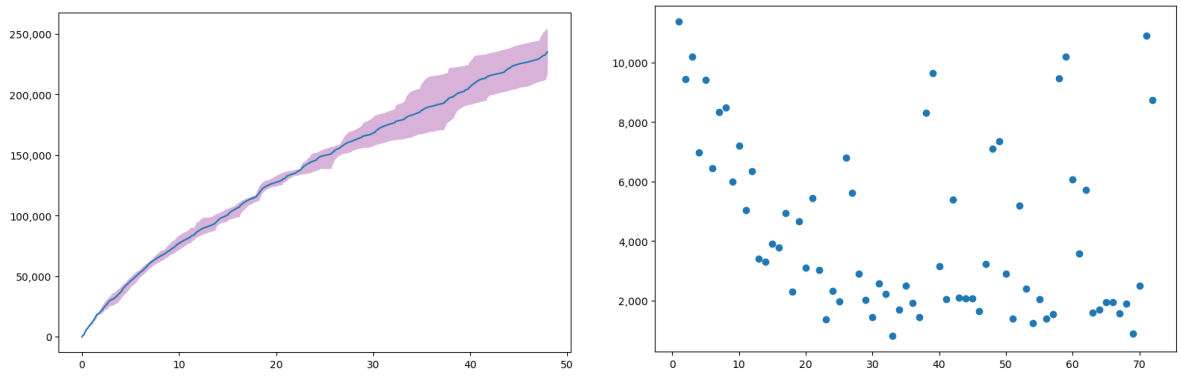
consistently reproducible and minimized for easier analysis and debugging.

Overall, these figures helped us identify two distinct thresholds in the execution of vanilla Syzkaller. The first threshold, already discussed in section 5.2.1, occurs at the 15-minute mark when Syzkaller finishes adding all the candidates from the repository. This initial 15-minute period is also when the coverage gains the most during the fuzzing campaign.

The second threshold appears at the 4-hour mark. At this point, coverage growth starts to slow down significantly, and the fuzzer begins to use `mutate` and `generate` requests more frequently. This increased frequency indicates that the request queue is empty more often after this threshold.

As a point of comparison, vanilla Syzkaller achieves 32 crashes including 8 distinct crash types over a 48-hour fuzzing campaign.

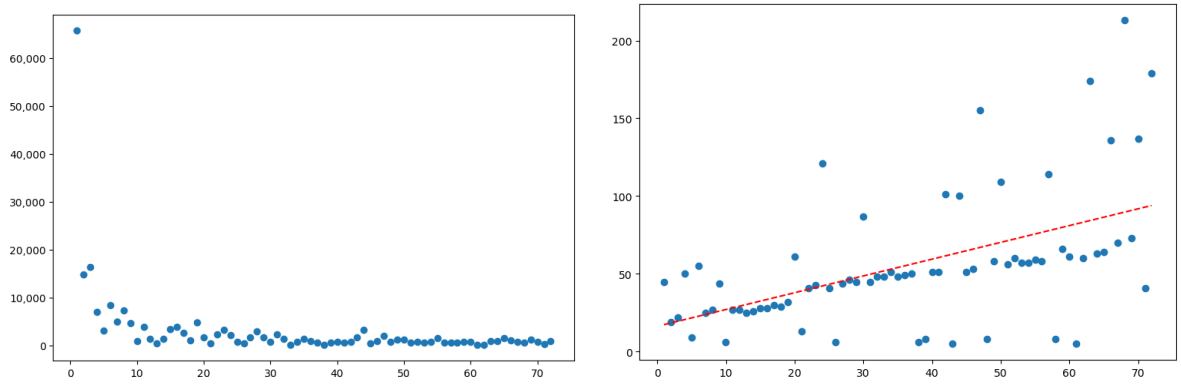
In summary, these metrics collectively illustrate the dynamic behaviour and performance of



(a) evolution over time

(b) relative increase per hour

Figure 5.10: Evolution of the “exec-triage” metric over a 72 hours period



(a) relative increase of coverage

(b) fitted relative increase of rpc traffic

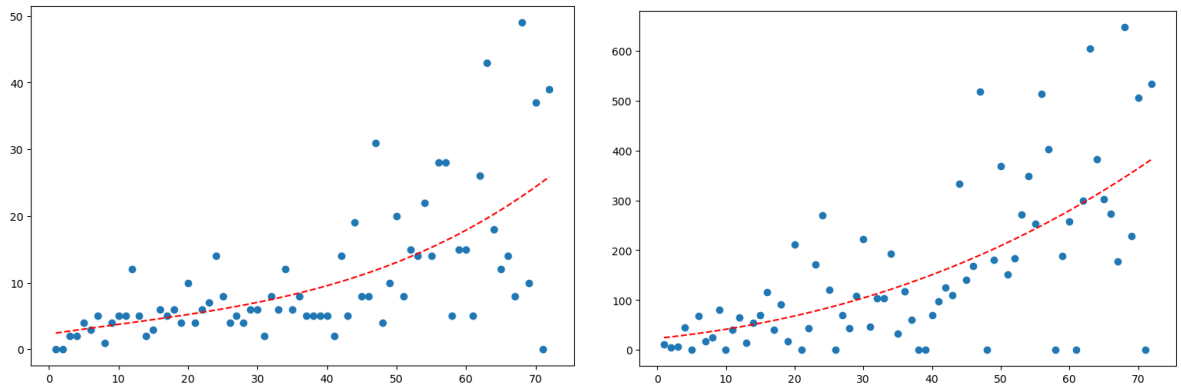
Figure 5.11: Relative evolution per hour of the coverage and fitted relative evolution of the rpc traffic over a 72 hours period

Syzkaller in its unaltered state. The initial burst in coverage, the interdependencies between different job types, and the steady increase in total executions and communication traffic are all indicative of a robust and active fuzzing process. These insights form a baseline for evaluating modifications and optimisations in subsequent experiments.

Overall Trend of Key Metrics

Figures 5.11 and 5.12 provide a closer look at the relative increase of four key metrics of the fuzzing campaign: coverage, RPC traffic, exec gen, and exec fuzz. These metrics offer insights into the behaviour and efficiency of Syzkaller over time.

The coverage metric shows significant contributions at the start of the fuzzing campaign (see Fig-



(a) fitted relative increase of `generate` requests

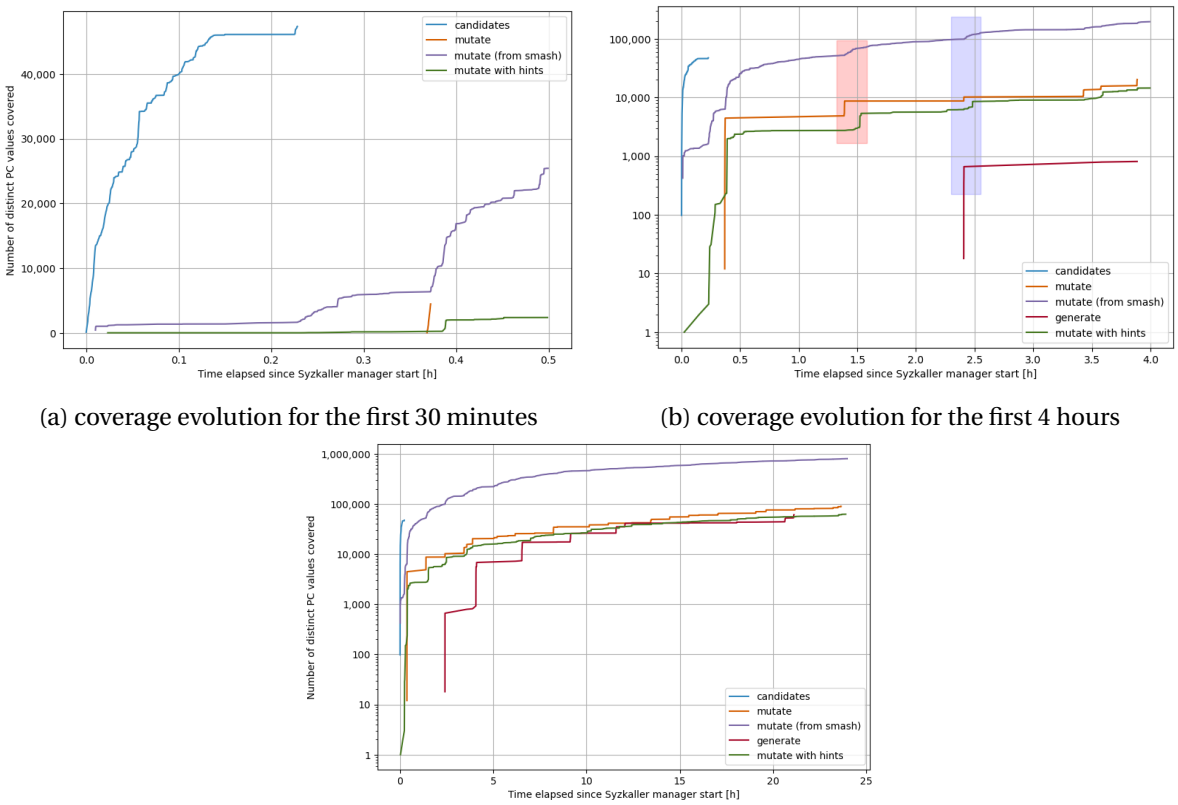
(b) fitted relative increase of `mutate` requests

Figure 5.12: Fitted relative evolution per hour of `generate` and `mutate` requests over a 72 hours period

ure 5.11a). During the first hour, there is a notable burst in coverage, primarily due to the addition of candidates. These candidates are pre-determined programs from a repository that are designed to quickly expose new areas of the kernel, leading to almost guaranteed new coverage. The following three data points remain high because the fuzzer executes numerous `smash` jobs, thus exploring and discovering various parts of the kernel under test. However, after the 4-hour mark, the rate of new coverage discoveries diminishes. This decrease occurs because the fuzzer has already identified the easily accessible parts of the code, reducing the frequency of `smash` jobs and, consequently, the rate of discovering new blocks.

The RPC traffic metric shown in Figure 5.11b displays a steady increase over time. This increase can be attributed to several factors. As the fuzzing campaign progresses, the fuzzer generates an increasing number of test cases and their results, all of which need to be communicated back to the manager. Additionally, detailed coverage data, crash information, and performance logs are continually transmitted to ensure the manager has the latest information. As the volume of these communications grows with the ongoing campaign, the amount of data transmitted via RPC naturally increases.

The gen and fuzz metrics shown in Figure 5.12 both follow a similar trend, starting slowly in the early stages of the fuzzing process and gradually increasing over time. Initially, the fuzzer focuses more on using candidates and executing `smash` requests to maximize new coverage quickly. However, as the easily discoverable code paths are exhausted, the frequency of `smash` jobs decreases. Consequently, the fuzzer relies more on `generate` and `mutate` requests since the request queue is often empty. This shift leads to an increase in generate and mutate metrics. Over time, these metrics are expected to converge towards a stable value. This stabilization will occur because, in the later stages of the fuzzing process, new discoveries become rare, resulting in the request queue being empty more often and the fuzzer more frequently executing `generate` and `mutate` requests.



(a) coverage evolution for the first 30 minutes

(b) coverage evolution for the first 4 hours

(c) coverage evolution for the first 24 hours

Figure 5.13: Coverage evolution after different thresholds: 30 minutes, 4 hours, and 24 hours

In summary, figures 5.11 and 5.12 highlight the dynamic changes in key metrics throughout the fuzzing campaign, providing a detailed view of how Syzkaller's behaviour evolves over time. The initial burst in coverage, the gradual increase in RPC traffic, and the shifting reliance on `generate` and `mutate` requests all illustrate the progression of the fuzzing process and the fuzzer's adaptation to the diminishing returns of new code path discoveries.

Syzkaller Behaviour Thresholds

Figure 5.13a highlights the number of candidates added and the frequency of each execution mode within the first 15 minutes of the fuzzing campaign. We can clearly see a substantial influx of candidates during this initial period. This surge in candidates is critical as it bootstraps the fuzzing process with pre-determined programs designed to explore the kernel efficiently. Once the candidates line stops growing, we observe a rise in `smash` jobs, indicating that the fuzzer is transitioning from using predefined programs to aggressively mutating existing ones to uncover new code paths.

In Figure 5.13b, we focus on the time frame leading up to the 4-hour mark, with particular

Time (hours)	generate	mutate	hints	smash mutations	smash
1	0	11	21	3948	2,169
4	4	68	256	31,714	7,424
12	43	366	1,146	128,224	17,319
24	115	1,468	2,260	242,407	23,213
48	301	4,254	4,762	497,335	34,166
72	714	10,726	8,100	836,907	48,069

Table 5.1: Comparison of modes popularity over time. The popularity of a mode is the number of times it was used in a specific time window

attention to the purple-highlighted section. A `generate` request, which creates a new program from scratch, often discovers (in early stages in the fuzzing campaign) new kernel regions not yet discovered. In this highlighted section, a `generate` request (red line) led to a noticeable increase in coverage, triggering a triage job to verify this increase. The triage job (blue line) then initiated a `smash` job (purple line), which further explored the newly discovered area, subsequently starting a `hints` job (green line) to optimise the test cases. This sequence demonstrates the cascading effect of a successful `generate` request, highlighting its importance in early-stage coverage growth.

Examining the period around the 1.5-hour mark highlighted in red, we observe a similar phenomenon where `mutate` requests ordered by the fuzzer lead to new code discoveries. This is evidenced by the bumps in the purple line (smash jobs) following an increase in `mutate` requests. `smash` jobs, which dig deeper into the newly discovered areas, in turn, trigger `hints` jobs (green line) to refine and optimise the mutations. This pattern defines the fuzzer's adaptive strategy, where initial mutations that reveal new paths prompt further targeted exploration through `smash` and `hints` jobs.

These detailed analyses from the subfigures of Figure 5.13 illustrate how different types of requests and execution modes interact dynamically over the course of the fuzzing campaign. The early influx of candidates sets a strong foundation for coverage, while `generate` and `mutate` requests continue to drive exploration and optimisation as the campaign progresses. Understanding these interactions provides valuable insights into the efficiency and behaviour of Syzkaller's fuzzing strategy.

5.2.2 Analysis of Syzkaller Modes of Operation

In the time window we selected, the fuzzer's workforce is dominated by the sheer number of `smash` jobs and mutations (97.72% in this case) resulting from `smash` as shown in Table 5.1. This is explained by the fact that early in the fuzzing process, the fuzzer frequently makes new discoveries. Each new discovery triggers a new `smash` job, which can also lead to further discoveries, creating a positive

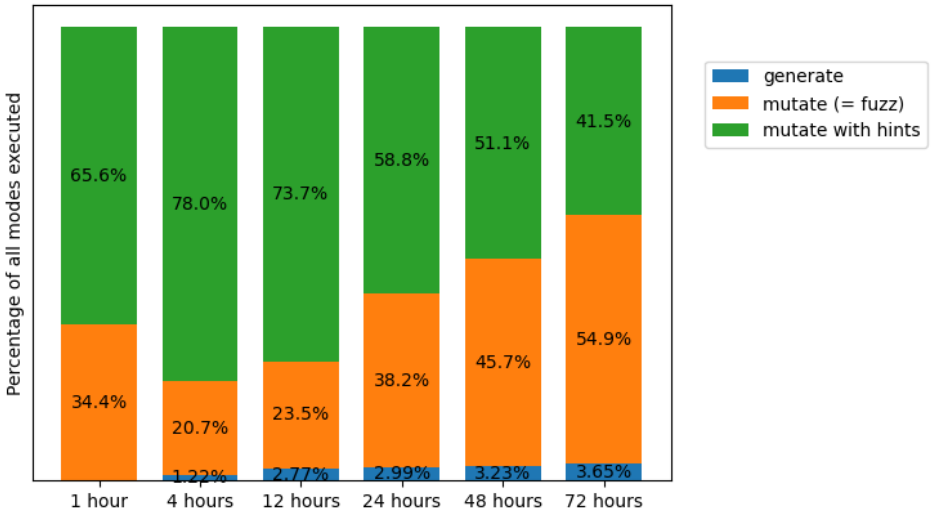


Figure 5.14: Modes popularity comparison without `smash`

feedback loop. Each program that triggers new coverage is also minimized and stored in Syzkaller's corpus. The minimization stage itself often contributes to new discoveries, as 10.24% of the new PC values discovered come from minimization requests. To better understand the distribution of the less popular modes, we plotted Figure 5.14, ignoring the `smash` mode and the `mutate` requests coming from `smash` jobs. Overall, we observe a trend where, as time progresses, the fuzzer increasingly uses more `generate` and `mutate` requests.

Similarly, looking at Figure 5.15, the `smash` mode is significantly ahead in terms of basic blocks discovered. Figure 5.16, however, presents a more evenly distributed metric. This figure shows the relative efficiency of each mode, computed by comparing the number of requests leading to new, undiscovered basic blocks versus requests that lead to basic blocks already covered in previous iterations. Every mode appears to be quite efficient: `generate` (see Figure 5.16a) discovers new basic blocks 75.19% of the time (i.e., more than three-quarters of the requests initiated by `generate` found new coverage), `mutate with hints` 73.37% of the time (see Figure 5.16d), `smash` 75.91% of the time (see Figure 5.16c), and `mutate` an impressive 100% of the times (over a very small sample size) it was called by the fuzzer as shown in Figure 5.16b. It is important to note that the graph includes the number of requests triggered by the different modes, encompassing minimize requests and triage jobs, and not solely the number of times the mode was called.

By analyzing the time series data from Table 5.1, we can observe that the popularity of modes changes over time. Initially, the fuzzer relies heavily on the `smash` mode and its derived mutations due to the frequent new discoveries early on. However, as the campaign progresses, the fuzzer begins to balance its approach by utilizing more `generate` and `mutate` requests. This shift is reflected in the changing trends of mode usage and their contributions to code coverage and efficiency. This analysis provides valuable insights into the fuzzer's behaviour and the effectiveness of each mode

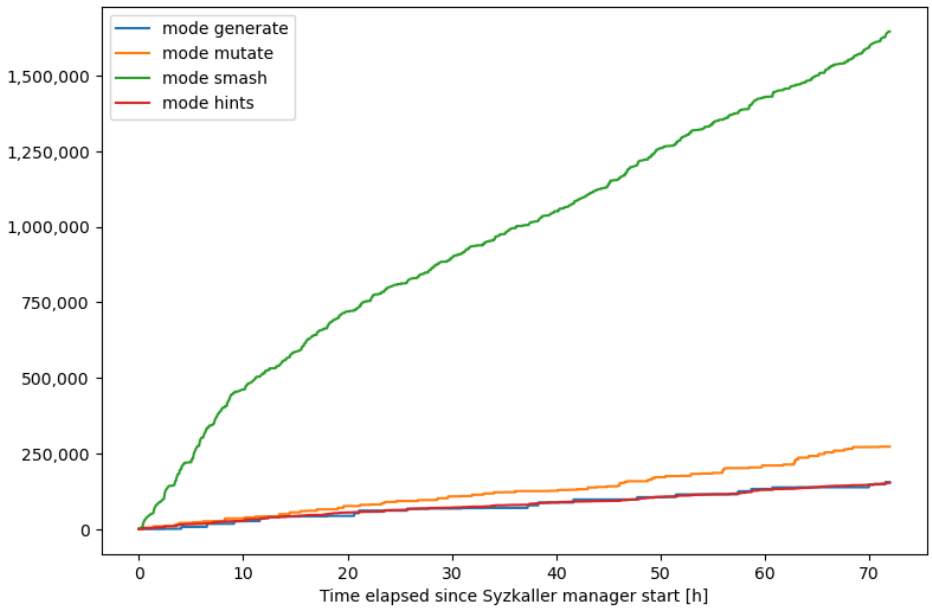


Figure 5.15: Comparison of the number of basic blocks discovered for each mode

Mutator	Number of calls	Percentage of total calls
mutator `insertCall`	1,534,757	51.09%
mutator `mutateArg`	768,424	25.58%
mutator `removeCall`	75,802	2.52%
mutator `splice`	23,861	0.79%
mutator `squashAny`	600,975	20.00%

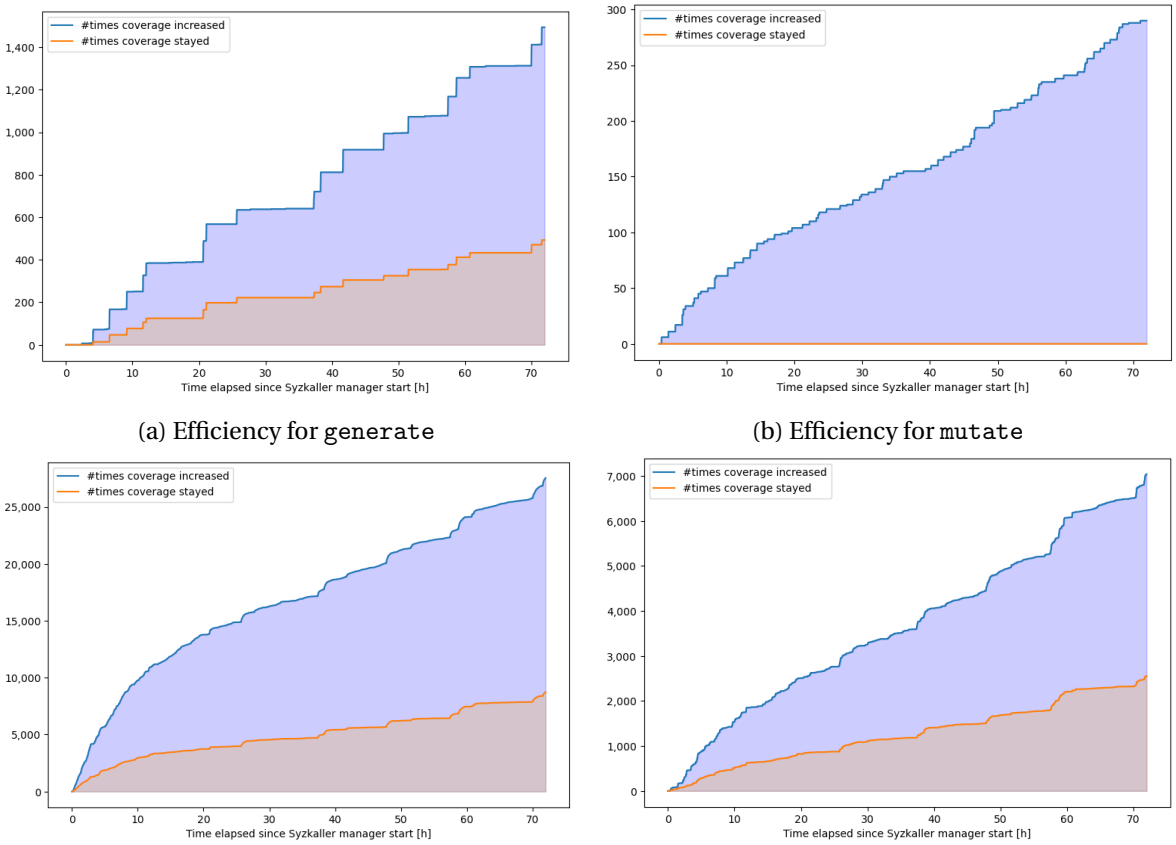
Table 5.2: Comparison of mutator popularity

over the course of the fuzzing process.

5.2.3 Analysis of Syzkaller Mutators

In Figure 5.17, we can see the evolution of the number of executions for each mutator over time. All five mutators follow a linear progression, which makes sense because the mutation energy, or how mutators are chosen for each mutation, is probabilistic. This comparison is shown in Table 5.2.

We also experimented with changing the way mutation energy was assigned. Using a uniform distribution, where each mutator has an equal 20% chance of being chosen, did not yield significant results. The results, in terms of code coverage, are very similar to the ones from vanilla Syzkaller. This likely indicates that the specific probabilities assigned to each mutator in the default setup do not significantly impact the overall effectiveness of the fuzzer. It suggests that the different mutators



(a) Efficiency for `generate`

(b) Efficiency for `mutate`

(c) Efficiency for `smash`

(d) Efficiency for `hints`

Figure 5.16: Comparison of the number of times coverage increased and number of times the coverage stagnated for requests triggered by each mode

are capable of producing similar sets of test cases, which ultimately lead to the same coverage. This redundancy could imply that despite their operational differences, the various mutators contribute equally to discovering new code paths.

5.2.4 Deep Analysis of `smash` Mode

Listing 2 depicts a simplified version of a program `p` being smashed in Syzkaller. The `smash` job begins by asynchronously starting a `hints` job (line (2)), then performs 100 mutations, where the fuzzer needs to wait before starting the next one (i.e., `execute` at (line (6)) is a blocking call, meaning that mutation n may only start when mutation $n-1$ finished executing). Note that each new mutation is performed on a fresh copy of `p`. The mutant is then utilised in a `collide` operation, which is responsible for detecting race conditions within the mutant. Finally, a fault injection is performed on the original program `p`. Fault injection in Syzkaller involves intentionally introducing

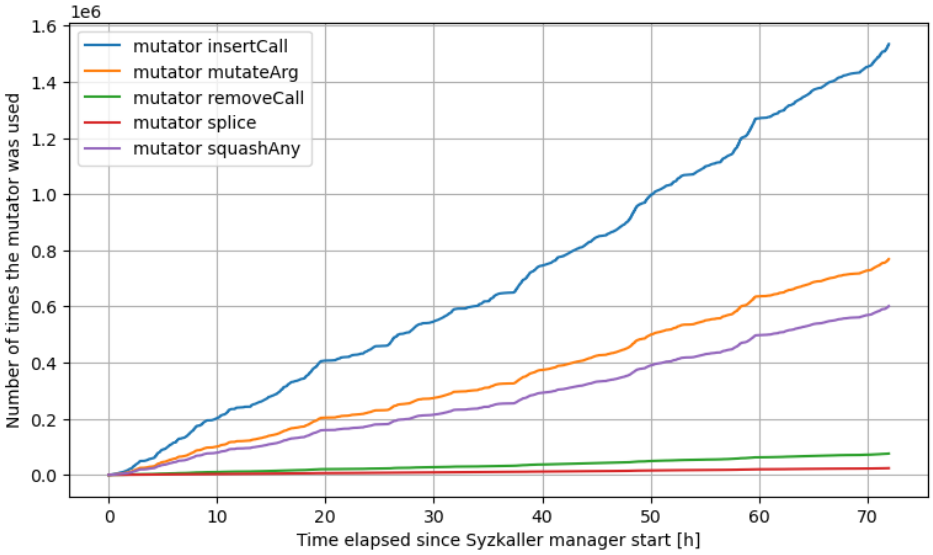


Figure 5.17: Comparison of the popularity of each mutator over time. The popularity of a mutator is the number of times it was used in a specific time window

```
1   func smash(p *Prog) {
2       go startHintsJob(p.Copy())
3
4       for i := 0; i < 100; i += 1 {
5           mutated := mutate(p)
6           execute(mutated)
7           collide(mutated)
8       }
9
10      faultInjection(p)
11  }
```

Listing 2: Simplified version of a `smash` job

errors into the program to test its resilience and discover potential vulnerabilities.

Our profiling revealed that an average `hints` job takes about 20.29 minutes. Interestingly, the entire `smash` job also takes on average 20.53 minutes. The fuzzer spends on average 5.298 seconds to mutate the program 100 times and around 0.198 seconds on fault injection. We observed that the average time spent in the queue for each of the 100 `mutate` requests is 11.64 seconds as shown in Figure 5.18, and this time remains constant throughout the process. This significant queuing time is a key factor contributing to the overall duration of the `smash` job.

When we exclude the time spent in the queue, then the combined execution time for performing every mutations, performing fault injection, and starting the `hints` job is only on average 67.8

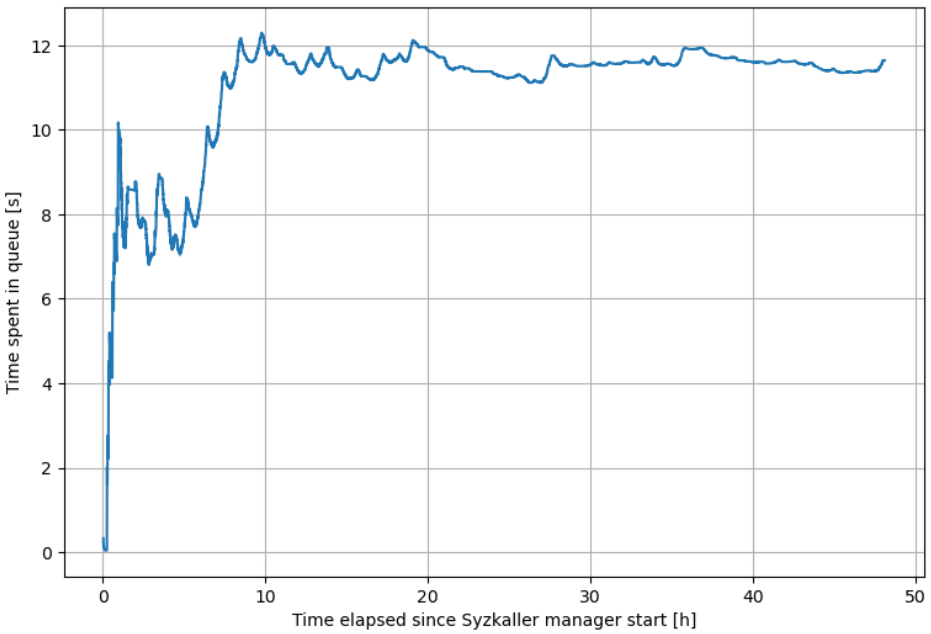


Figure 5.18: Evolution of the average time spent for a request in Syzkaller's priority queue

seconds. This indicates that the queuing process is a major bottleneck, representing 94.53% of the total time spent in the `smash` mode. Addressing this inefficiency could substantially reduce the total time required for a `smash` job, thereby enhancing the efficiency and effectiveness of the fuzzer.

5.2.5 Deep Analysis of `squashAny` Mutator

As a reminder, the `squashAny` mutator can fail, leading to potential wasted cycles. Considering that the mutator is chosen with a probability of 20% as shown in Table 5.2, this could potentially lead to serious performance issues since, as a reminder, three out of the four Syzkaller modes are using mutators.

We first started to analyse the mutator by taking a look at its success rate. We computed an average success rate of 28.5% on a 48-hour experiment. The detail for the success rate over time is shown in Figure 5.19, approaching the 28.5% average as time goes on. A second important component to understand whether `squashAny` is a potential issue is looking at the total wasted time the failures generated. We computed that, in our 48-hour run, 13.88 seconds were wasted on failures while 54.03 seconds were spent on the mutator success. This averages to an average of 0.047 milliseconds per failure, which is small. Thus, even though the mutator fails almost two-thirds of the time, the time wasted is negligible.

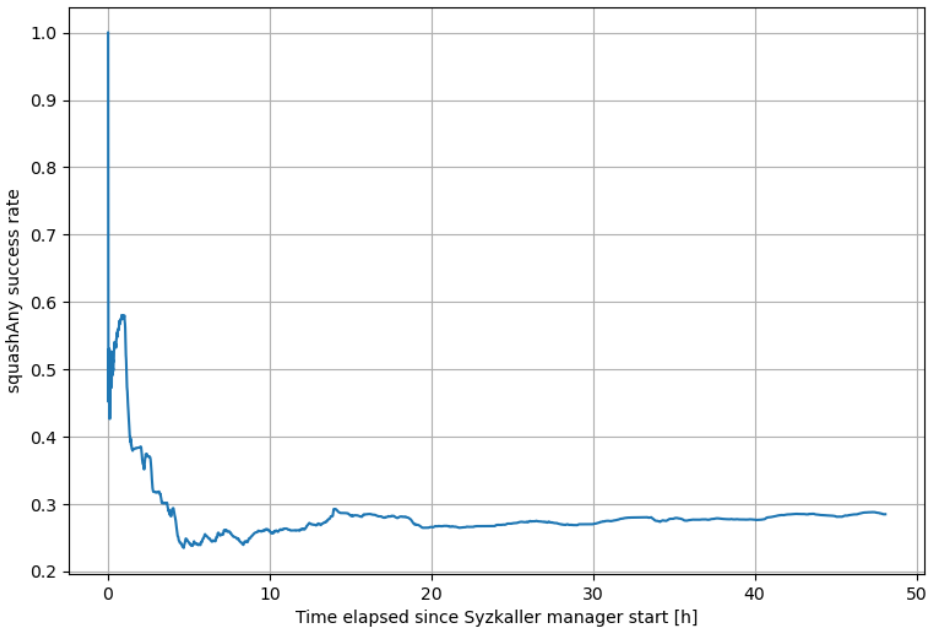


Figure 5.19: Evolution of `squashAny`'s success rate

5.3 Single Mode Study

In the first Figure 5.20, we see the coverage reached by Syzkaller running with a single mode enabled. The number of executions for each mode is depicted in Figure 5.21. For instance, the purple line indicates the number of executions for `smash` when the `smash`-only experiment is running. It is important to note that the experiment running with only the `smash` mode still has access to mutations when performing the 100 mutations inside of the `smash` jobs. Unsurprisingly, none of the single-mode experiments outperform the original Syzkaller.

The `smash` mode experiment comes close, which makes sense since Syzkaller predominantly uses the `smash` mode. Therefore, its performance does not differ much from normal Syzkaller execution. In the second figure, we observe that the rate at which the fuzzer triggers new `smash` jobs decreases over time. This is because `smash` jobs are triggered when new coverage is discovered. After the initial coverage burst from the candidates in the first 15 minutes of the campaign, the `smash` mode alone struggles to create programs that can explore entirely new parts of the kernel under test, unlike the `generate` mode, which leads to fewer smash jobs being triggered.

The `generate`-only experiment is straightforward to explain. Since the request queue is always empty, a new `generate` request is executed at each fuzzer iteration. The `generate` mode does not depend on any other mode and typically does not trigger any additional jobs or requests in vanilla Syzkaller. This results in an almost perfectly linear progression in the second graph. However, the coverage reached slows down faster than standard Syzkaller because the `generate` mode creates

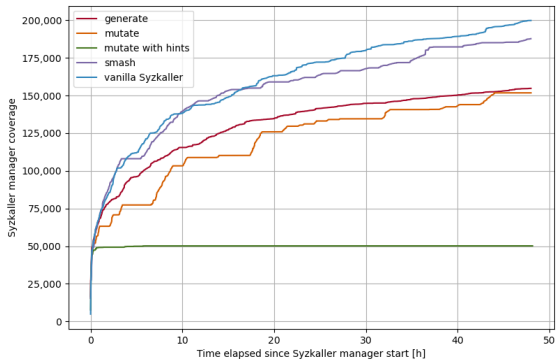


Figure 5.20: Single mode comparison

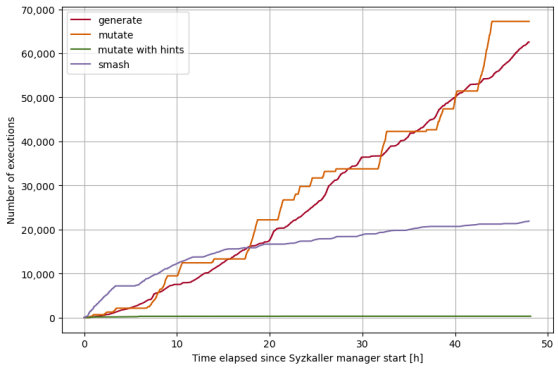


Figure 5.21: Single mode executions

many unique programs that cover various parts of the system under test but does not dig further to alter the generated programs like a mutation-based mode would.

When we enable only the `mutate with hints` mode, we notice that the coverage stops growing after just a few minutes as shown in Figure 5.20. The reason for this might be that `mutate with hints` relies heavily on existing knowledge and patterns discovered during the fuzzing process to create new test cases. While this mode can quickly explore areas of the code that are close to already discovered paths, it lacks the diversity and randomness provided by other mutation strategies. As a result, it quickly exhausts its potential to discover new code paths and becomes less effective in increasing coverage over time.

In conclusion, while single-mode experiments provide valuable insights into the individual contributions of each mode, the results highlight the importance of using multiple modes together for optimal fuzzing performance. Combining the strengths of different modes allows Syzkaller to achieve greater coverage and discover more vulnerabilities.

5.4 Ablation Study

In Figure 5.22, we can see the coverage achieved by each ablation study (one for each of the four modes of operation) compared to the coverage by vanilla Syzkaller (blue line).

Mutations are the primary way Syzkaller alters programs to create new ones. Thus, removing the `mutate` mode (orange line) leads to the worst results, causing a steep decrease in Syzkaller's efficiency. When the `mutate` mode is removed, the number of `generate` requests executed increases compared to vanilla Syzkaller, but this does not compensate for the loss in overall coverage.

With the removal of the `generate` mode, Syzkaller is restricted to "local searches", focusing only on variations of existing test cases and thus restricting its ability to explore entirely new code paths.

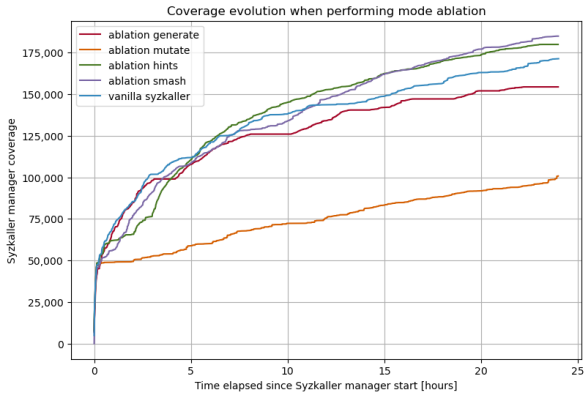


Figure 5.22: Coverage evolution when performing modes ablation

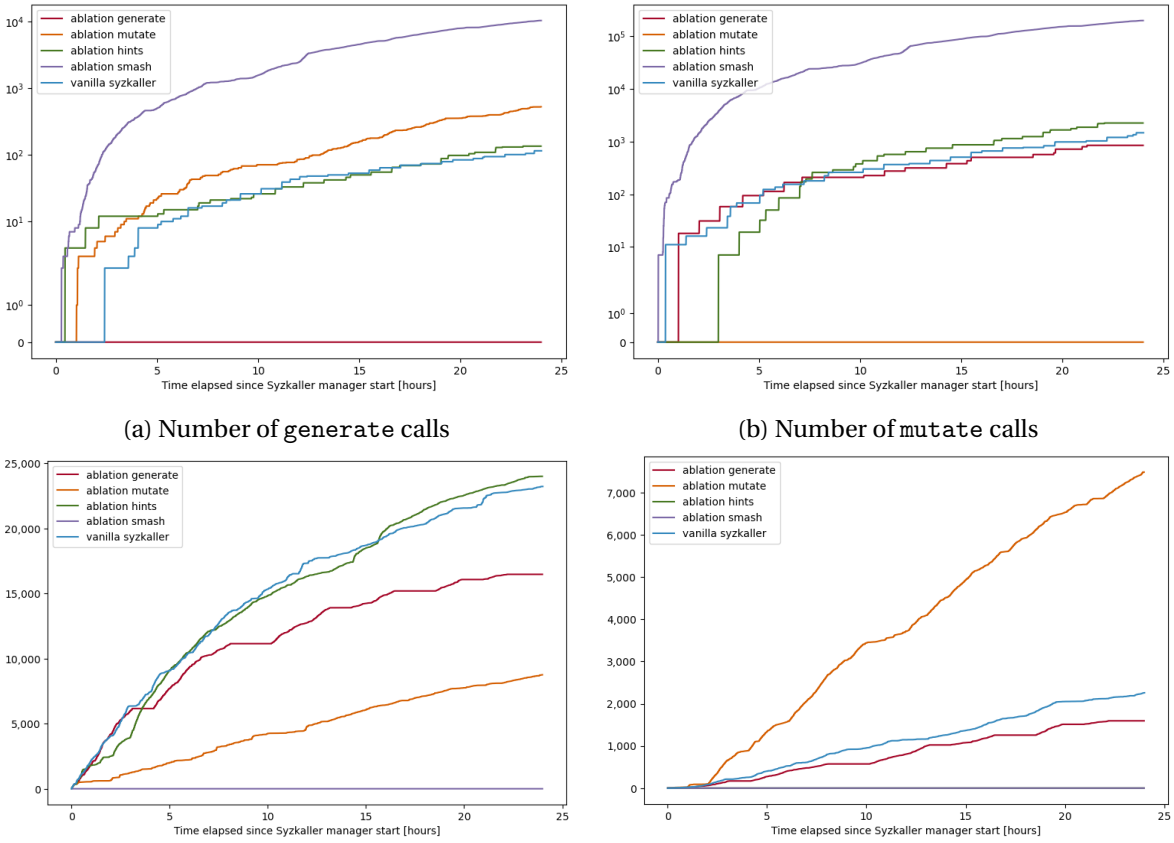
Since mutations do not create entirely new programs like the `generate` mode, the fuzzing campaign ends up with more similar programs. This does not mean that Syzkaller will stop discovering basic blocks, but rather that it might discover them more slowly. For the last four graphs in Figure 5.23, the results are overall close to vanilla Syzkaller, which is expected since Syzkaller does not use the `generate` mode much in the first 24 hours. It is also interesting to note that the rate at which `smash` jobs are triggered slows down over time. This is because `smash` jobs are initiated by triage jobs when new coverage is found, which is lessened when the `generate` mode is disabled.

Removing the `smash` mode results in the fuzzer only using `generate` and `mutate` modes (purple line in each graph). The coverage starts to grow faster than vanilla Syzkaller because more `generate` requests are executed, discovering various parts of the kernel quicker and providing a burst of coverage. In the long run, however, vanilla Syzkaller outperforms the ablation of the `smash` mode experiment.

Removing the `hints` mode results in coverage very similar to vanilla Syzkaller. This is because mutations with `hints` are softer mutations. Over time, this experiment is also outperformed by vanilla Syzkaller. This is because, without this mode, Syzkaller is not as thorough when searching an area of the kernel under test and easily jumps to a totally new part of the kernel due to a `generate` request. It is better to thoroughly search an area before jumping to a totally new one to ensure all potential vulnerabilities in that area are discovered.

5.5 Shuffle Mutator

We observe that on average, each time the `shuffle` mutator is called, it fails the validation process on average 4.00 times before being accepted as a valid program (i.e., no syscall dependencies broken). As detailed in section 4.5, `shuffle` attempts to shuffle the program a maximum of 16 times before giving up. The failure rate of approximately 4 attempts on average highlights the complexity and



(a) Number of `generate` calls

(b) Number of `mutate` calls

(c) Number of `smash` calls

(d) Number of `hints` calls

Figure 5.23: Comparison of the number of calls made to each of the four Syzkaller modes for each ablation study

delicacy of rearranging system call sequences without disrupting program behaviour.

Furthermore, our analysis revealed that for each mutation, Syzkaller employs an average of 3.58 mutators. This finding implies that there is roughly a 2.80% chance that any given mutation will contain at least one instance of the `shuffle` mutator.

The `shuffle` experiment yields a total of 39 crashes and 10 different crash types over a 48-hour period which is better than vanilla Syzkaller.

Overall, implementing a `shuffle` mutator is a promising experiment that could enhance the effectiveness of Syzkaller by introducing additional diversity into the test cases. However, its success heavily depends on robust dependency analysis to ensure that the shuffled sequences remain valid. If the mutator frequently breaks dependencies, leading to invalid sequences, it indicates a need for more sophisticated dependency handling.

Time (hours)	Vanilla Syzkaller	Probabilistic generate	Increase (delta)	
1	70,863	70,724	▽	-0.20%
4	108,900	107,359	▽	-1.42%
12	143,567	148,157	△	+3.20%
24	171,270	172,795	△	+0.89%
48	199,612	200,984	△	+0.69%

Table 5.3: Coverage comparison at different points in time

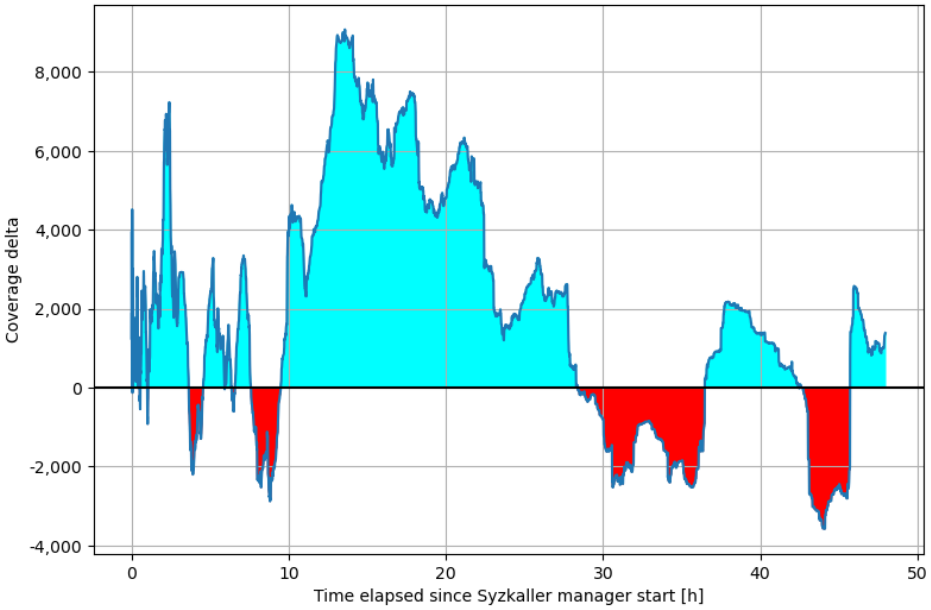


Figure 5.24: Delta coverage comparing the difference between standard Syzkaller and the probabilistic generate experiment

5.6 Probabilistic Generate Experiment

Our experiment of occasionally adding generate requests in the fuzzer did not yield conclusive evidence that it improved the fuzzer's performance. As shown in Table 5.3, the results between the probabilistic generate experiment and the normal Syzkaller are very similar. At key thresholds such as 12 hours, 24 hours, and 48 hours, the coverage was only marginally higher for our experiment compared to the standard Syzkaller.

In the following graph (see Figure 5.24), blue areas represent windows in time where the experiment yielded better results than the standard Syzkaller. Conversely, red sections depict moments where the vanilla Syzkaller performed better. The probabilistic generate experiment seemed beneficial between the 10-hour and 28-hour marks. This makes sense because, at the beginning, Syzkaller

does not frequently use the `generate` mode, which is known to discover entirely new parts of the codebase and thus increase coverage easily. As time goes on, it starts using the `generate` mode more often, reducing the impact of our experiment. With this modification, we make the fuzzer operate more like BFS, so at the early stages, the fuzzer with more `generate` mode covers more code than vanilla Syzkaller, but in the end, they both achieve similar coverage, as the total amount of code the fuzzer can cover is the same; we are merely changing the order of the search.

However, after the 28-hour mark, the experiment does not provide any significant advantage, with performance oscillating between being slightly better and slightly worse than the standard Syzkaller. It also achieved a total of 23 crashes including 6 distinct types over the 48-hour period. This suggests that probabilistic generate is not particularly impactful in the later stages of the fuzzing campaign. By this point, Syzkaller has already covered most of the code through its regular `generate` and `mutate` requests. Adding a few more `generate` requests (one in a thousand) does not significantly affect the overall results because the code that can be covered has largely already been explored. Extending the time further will not yield better results, as the additional `generate` requests do not contribute to uncovering new code paths.

5.7 Dependency Analysis Removal

In our experiment, we observe several notable changes compared to a vanilla Syzkaller run. Over a 48-hour period, the number of crashes increased dramatically from 32 to 135 as shown in Figure 5.25a. However, the number of distinct crash types only went up slightly from 8 to 9. This supports the theory that without dependency analysis, Syzkaller more frequently encounters the same bugs, such as syscall dependency breakage via mutations or mutations producing invalid arguments (e.g., out-of-bounds values). The substantial increase in crashes negatively impacts other metrics; each crash forces Syzkaller to start a new virtual machine with a fresh instance of the fuzzer, which consumes valuable fuzzing time unnecessarily. Consequently, the "coverage" metric is lower due to these interruptions as shown in Figure 5.25b.

Interestingly, the number of executions for most modes is greater than in the vanilla Syzkaller run. This is because operations that previously required dependency analysis now execute faster without it. Specifically, the metrics for `generate`, `mutate`, and `smash` modes shown in figures 5.25c, 5.25d, and 5.25e all show an increase over the 48-hour period. The only mode with lower usage is the `mutate with hints` mode as shown in Figure 5.25f. Each `hints` job is triggered by a `smash` job, so we would expect an increase in `hints` jobs corresponding to the greater number of `smash` jobs. However, this is not the case due to the increased crash frequency. Although `smash` jobs and `hints` jobs take approximately the same amount of time to complete (see subsection 5.2.4), the higher likelihood of crashes during the 100 `smash` mutations for the various `smash` jobs running concurrently prevents the `hints` job from finishing execution, as the fuzzer crashes beforehand.

5.8 Profiling `syz-executor`

Flame graphs are read from bottom to top, offering a snapshot of a recorded call tree. Functions, represented by blocks, are called from the base and extend upwards, with each function being invoked by the one directly below it. The width of each block corresponds to the execution time of the function it represents. The color of each block indicates the frequency of its calls. Red highlighting hotspots of frequent usage. As the color shifts towards yellow, it signifies a decrease in the number of calls.

`syz-executor` runs sequences of system calls via the function `execute_one()` and individual syscalls via `execute_call()` both situated in the "`executor.cc`" file [6].

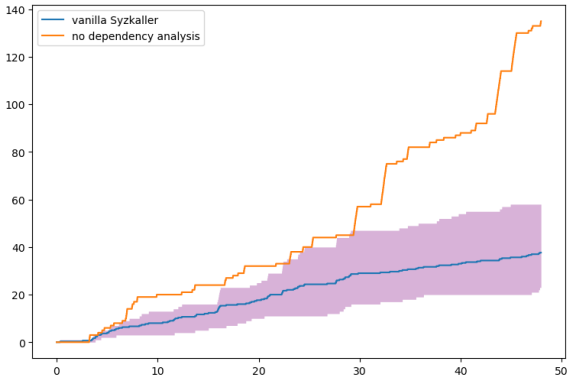
After running the fuzzer for 2 minutes, `syz-executor` did not have time to execute any individual syscall, but started to run at least one program. At this moment in time, `syz-executor` was the author of 20.75% of the instructions run by the CPU whereas `syz-fuzzer` only accounted for 0.56%.

Inspecting the 2-hours mark recording, `syz-executor` did execute individual syscalls and programs. Figure 5.26 shows the breakdown; `execute_call()` is circled in green and `execute_one()` in purple. The CPU spent 0.05% of its time executing individual syscall and 0.6% running programs via `execute_one()`. Zooming in `execute_call()`'s tree (see Figure 5.27), we see a different picture. Syscall executions are, in fact, a "hotspot" as shown by the vivid red color for `execute_call()`'s block. However executing these system calls is finished so quickly compared to the other CPU activities that we barely do not see the metric on the zoomed out graph.

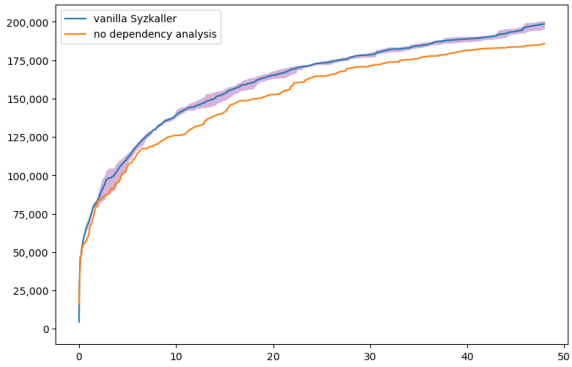
Figure 5.28 depicts the opposite for `execute_one()`. Execution of the programs does take a small, visible, amount of time on the zoomed out graph (0.6% of the processor time spent during the sampling). The function is however not a hotspot. This is likely due to the fact that `execute_one()` does not always execute the syscall itself. This time however, `syz-executor` alone was responsible for 65.85% of the CPU time spent executing instructions, and 4.89% for `syz-fuzzer`.

At the 24-hour mark, results are almost identical with the processor spending 0.05% of its time on `execute_call()` and 0.7% on `execute_call()`.

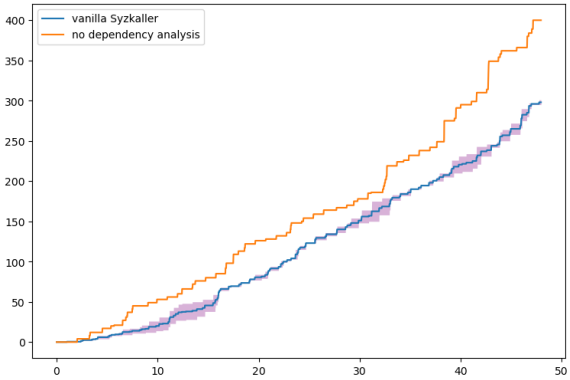
The results indicate that `execute_call()` is frequently executed by the CPU, making it a hotspot. However, despite its high call frequency, it does not pose a performance problem due to its negligible execution time. Conversely, `execute_one()` also requires minimal execution time and is not a hotspot, therefore not presenting a significant issue either.



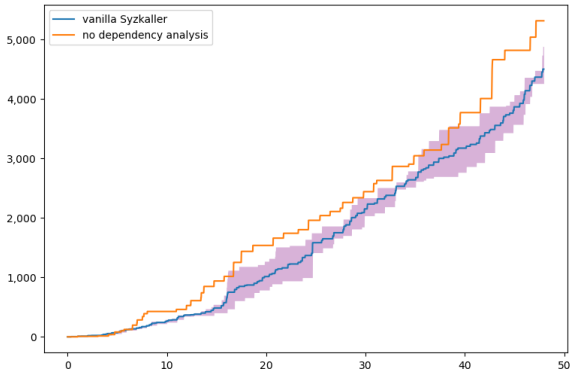
(a) Crashes comparison



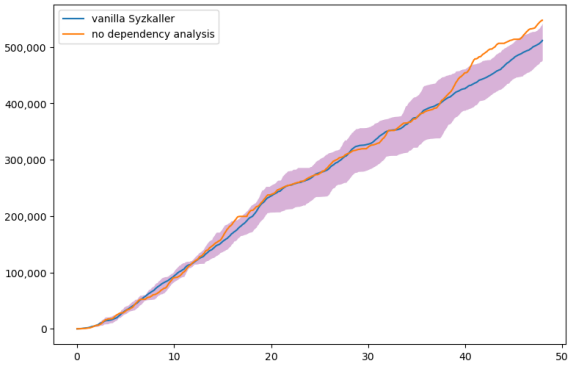
(b) Coverage comparison



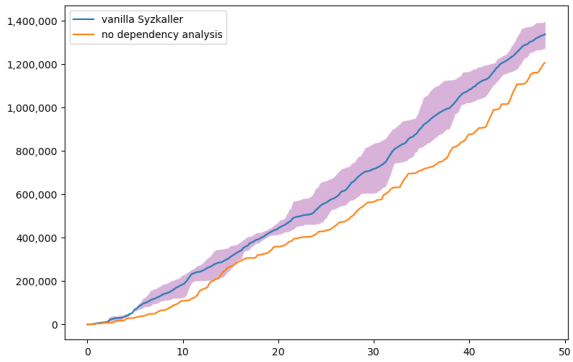
(c) `generate` requests comparison



(d) `mutate` requests comparison



(e) `smash` jobs comparison



(f) `hints` jobs comparison

Figure 5.25: Comparison of the number of crashes, the coverage achieved, the quantity of `generate` and `mutate` requests, and the quantity `smash` and `hints` jobs executed with and without the dependency analysis removal experiment

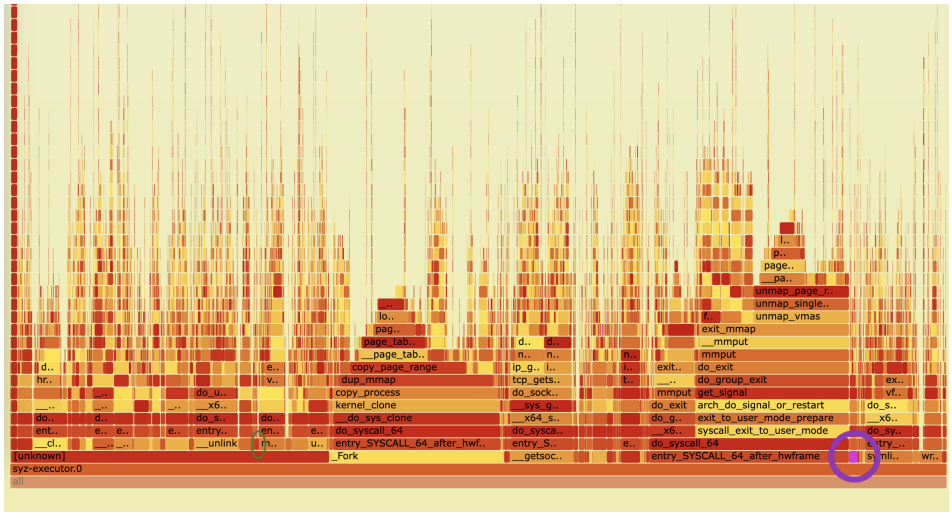


Figure 5.26: Zoomed out view of `execute_call()` circled in green and `execute_one()` circled in purple after 2 hours of fuzzing

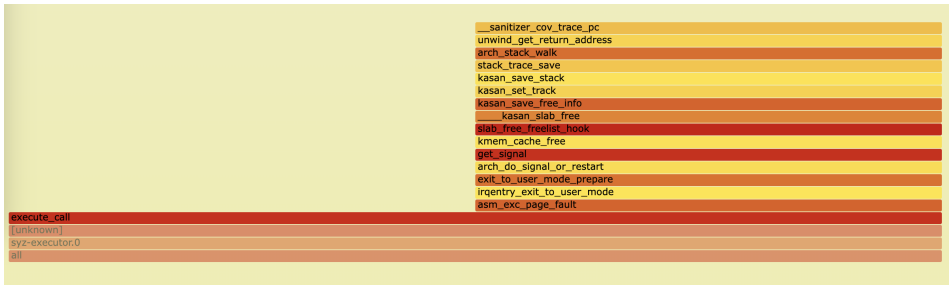


Figure 5.27: View of `execute_call()` after 2 hours of fuzzing

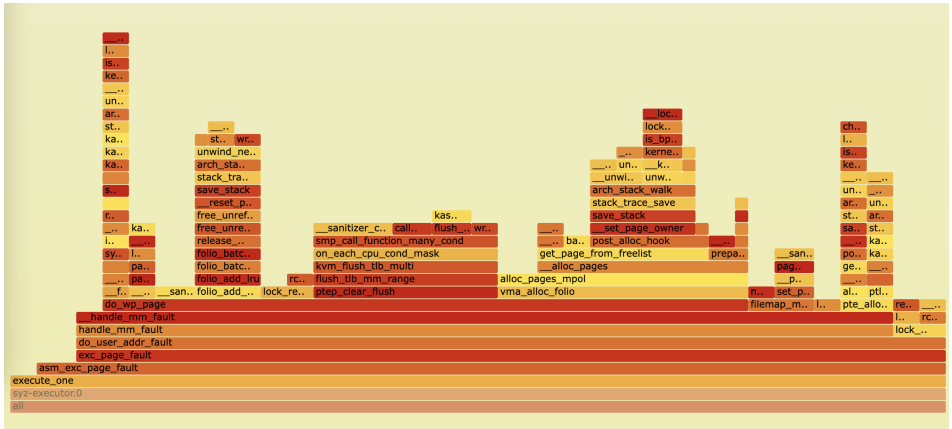


Figure 5.28: View of `execute_one()` after 2 hours of fuzzing

Chapter 6

Related Work

A number of papers improving Syzkaller were published recently with goal to optimise kernel fuzzing. For instance, FastSyzkaller [12] applies the N-Gram model to extract vulnerable sequences through combinations of calls that already crashed in the past. The best results are achieved when setting $N = 5$.

Thunderkaller [11] introduces Kernel image duplication (KID) and three separate Syzkaller optimisations to achieve a 2.8× speedup and 25.8% more basic blocks coverage. Thunderkaller introduces Kernel Image Duplication (KID) and three separate Syzkaller optimisations. KID creates three copies of the kernel image: the original copy with both KASAN and KCOV removed, a full copy with all instrumentation, and a triage copy without KASAN's checks. Thunderkaller then dispatches targets that are not in minimization or triage stage on the full copy. The triaging is done on the copy without KASAN.

Kernel fuzzing often depends on precise and hard-coded critical components to be effective. Syzkaller relies on templates, now called "syscall descriptions" [25], to operate. These template are man-made, and thus not scalable and inflexible. SyzVegas [27] improves coverage by eliminating human-curated components through reinforcement-learning.

HEALER [23] [22] is an OS fuzzer that enhances fuzzing effectiveness by learning the dependencies between syscalls through dynamic analysis of minimized test cases. This approach improves input generation and mutation, leading to higher quality test cases and more effective fuzzing. HEALER outperforms Syzkaller, improving branch coverage by 28%.

The Linux source code [14] evolves rapidly; the enriched corpus [17] [18] leverages previously discovered crashes, creating an enhanced base corpus. Combined with HEALER, this duo outperforms vanilla Syzkaller's coverage by on average 51%.

Horus [15] speeds up data transfer between the fuzzing campaign by eliminating slow network-

based transfers. The idea is to make the kernel fuzzing-aware (i.e., modify the kernel and the QEMU instance without inadvertently introducing new external bugs) via stub structures that facilitate data transfers.

Some improvements focus solely on Syzkaller's syscalls dependency analysis [10]. MoonShine [19] is a strategy implemented in the hope of improving OS fuzzing by "distilling" seeds coming from syscall sequences from real-world program traces, thus preserving dependencies across calls. MoonShine was implemented into a Syzkaller extension, enhancing the code coverage achieved on the Linux kernel by an average of 13%.

Chapter 7

Conclusion

In this paper, we have profiled Syzkaller at two levels: the strategy level with `syz-fuzzer` profiling and the implementation level with the profiling of `syz-executor`. We have dissected the popularity and cost of each mode of operation and each mutator. Additionally, we have taken a deeper look at a mode and a mutator that are particularly popular. Based on these results, we identified Syzkaller's queue of requests as the largest bottleneck. We then designed two experiments aimed at increasing Syzkaller's effectiveness. The first experiment, probabilistic generate, demonstrated positive results with an average increase of 2.5% in coverage during the early stages. The second experiment, which involved developing a new `shuffle` mutator, showed promising potential, contingent on combining adequate system calls dependency analysis during its mutation process. Our research advances the status quo by providing an updated profiling of the latest Syzkaller, highlighting inefficiencies, and proposing enhancements.

Bibliography

[1] AlDanial. *cloc: Count lines of code*. June 6, 2024. URL: https://github.com/AlDanial/cloc.

[2] Andrey Konovalov. *Syzkaller main three components*. https://xairy.io/articles/syzkaller-external-network. Accessed: 2024-06-06.

[3] Bart Miller. *Fuzzer course assignment*. https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf. Accessed: 2024-06-06.

[4] Linux Developers. *Linux wiki: kernel profiling with perf*. https://perf.wiki.kernel.org/index.php/Tutorial. Section "Sampling with perf record", Accessed: 2024-06-07.

[5] Syzkaller's developers. *Setup: Ubuntu host, QEMU vm, x86-64 kernel*. https://github.com/google/syzkaller/blob/master/docs/linux/setup_ubuntu-host_qemu-vm_x86-64-kernel.md. Accessed: 2024-06-07.

[6] Google. *Syzkaller: an unsupervised coverage-guided kernel fuzzer*. June 6, 2024. URL: https://github.com/google/syzkaller/tree/master.

[7] Google. *Syzkaller: an unsupervised coverage-guided kernel fuzzer*. Updated version with jobs and requests. June 6, 2024. URL: https://github.com/google/syzkaller/tree/c35c26ec6312219507c518bae2e56c1ea46a5f36.

[8] Brendan Gregg. *Flame Graphs: visualize profiled code*. June 7, 2024. URL: https://github.com/brendangregg/FlameGraph.

[9] Brendan Gregg. *perf examples*. https://brendangregg.com/perf.html. Accessed: 2024-06-06.

[10] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. "Demystifying the Dependency Challenge in Kernel Fuzzing". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2022, pp. 659–671. DOI: 10.1145/3510003.3510126.

[11] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 2023, pp. 1567–1578. DOI: 10.1109/ASE56229.2023.00124. URL: https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00124.

[12] Dan Li and Hua Chen. “FastSyzkaller: Improving Fuzz Efficiency for Linux Kernel Fuzzing”. In: *Journal of Physics: Conference Series* 1176.2 (Mar. 2019), p. 022013. DOI: 10.1088/1742-6596/1176/2/022013. URL: https://dx.doi.org/10.1088/1742-6596/1176/2/022013.

[13] Linux Developers. *KCOV: code coverage for fuzzing*. June 3, 2024. URL: https://www.kernel.org/doc/html/latest/dev-tools/kcov.html.

[14] Linux Developers. *Linux kernel*. Version 6.7. June 3, 2024. URL: https://github.com/torvalds/linux/releases/tag/v6.7.

[15] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. “Horus: Accelerating Kernel Fuzzing through Efficient Host-VM Memory Access Procedures”. In: *ACM Trans. Softw. Eng. Methodol.* 33.1 (Nov. 2023). ISSN: 1049-331X. DOI: 10.1145/3611665. URL: https://doi.org/10.1145/3611665.

[16] Barton P. Miller, Lars Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[17] Palash B. Oswal. “Improving Linux Kernel Fuzzing”. Available at https://doi.org/10.1184/R1/23593134.v1, Accessed: 2024-06-06. PhD thesis. July 2023.

[18] Palash B. Oswal. *Linux kernel enriched corpus*. June 6, 2024. URL: https://github.com/cmu-pasta/linux-kernel-enriched-corpus.

[19] Shankara Pailoor, Andrew Aday, and Suman Jana. “Moonshine: optimizing OS fuzzer seed selection with trace distillation”. In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC’18. Baltimore, MD, USA: USENIX Association, 2018, pp. 729–743. ISBN: 9781931971461.

[20] Nicolas Raulin. *Research project*. Accessed: 2024-06-02. 2024. URL: https://github.com/RaulinN/research_project.

[21] Nicolas Raulin. *Syzkaller fork*. Based on Google’s Syzkaller repository, Accessed: 2024-06-02. 2024. URL: https://github.com/RaulinN/syzkaller.

[22] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. *Healer: kernel fuzzer inspired by Syzkaller*. June 6, 2024. URL: https://github.com/SunHao-0/healer.

[23] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. "HEALER: Relation Learning Guided Kernel Fuzzing". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 344–358. ISBN: 9781450387095. DOI: 10.1145/3477132.3483547. URL: https://doi.org/10.1145/3477132.3483547.

[24] Syzkaller's developers. *Documentation: fuzzer coverage computation*. https://github.com/google/syzkaller/blob/master/docs/coverage.md. Accessed: 2024-06-03.

[25] Syzkaller's developers. *Documentation: syscall descriptions*. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions.md. Accessed: 2024-06-06.

[26] Syzkaller's developers. *Syzbot dashboard*. https://syzkaller.appspot.com/upstream. Accessed: 2024-06-06.

[27] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. "SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2741–2758. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng.

[28] Wen Xu et al. *Fuzzing File Systems via Two-Dimensional Input Space Exploration*. https://taesoo.kim/pubs/2019/xu:janus.pdf. Accessed: 2024-06-03.

[29] Zhenpeng Lin et al. *GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs*. https://zplin.me/papers/GREBE.pdf. Accessed: 2024-06-03.