SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning

Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh,
University of California, Riverside

https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

SYZVEGAS: Beating Kernel Fuzzing Odds with Reinforcement Learning

Daimeng Wang, Zheng Zhang, Hang Zhang
Zhiyun Qian, Srikanth V. Krishnamurthy, Nael Abu-Ghazaleh
University of California, Riverside
{dwang030, zzhan173, hang, zhiyunq, krish, nael}@cs.ucr.edu

Abstract

Fuzzing embeds a large number of decisions requiring fine-tuned and hard-coded parameters to maximize its efficiency. This is especially true for kernel fuzzing due to (1) OS kernels' sheer size and complexity, (2) a unique syscall interface that requires special handling (e.g., encoding explicit dependencies among syscalls), and (3) behaviors of inputs (i.e., test cases) are often not reproducible due to the stateful nature of OS kernels. Hence, Syzkaller [14], the state-of-art gray-box kernel fuzzer, incorporates numerous procedures, decision points, and hard-coded parameters master-crafted by domain experts. Unfortunately, hard-coded strategies cannot adjust to factors such as different fuzzing environments/targets and the dynamically changing potency of tasks and/or seeds, limiting the overall effectiveness of the fuzzer. In this paper, we propose SYZVEGAS, a fuzzer that dynamically and automatically adapts two of the most critical decision points in Syzkaller, *task selection* and *seed selection*, to remarkably improve coverage reached per unit-time. SYZVEGAS's adaptation leverages multi-armed-bandit (MAB) algorithms along with a novel reward assessment model. Our extensive evaluations of SYZVEGAS on the latest Linux Kernel and its subsystems demonstrate that it (i) finds up to 38.7% more coverage than the default Syzkaller, (ii) better discovers bugs/crashes (8 more unique crashes) and (iii) has very low 2.1% performance overhead. We reported our findings to Google's Syzkaller team and are actively working on pushing our changes upstream.

1 Introduction

Gray-box fuzzing or *coverage-guided fuzzing* has recently gained traction. Fuzzing is often perceived as an art, as fuzzers embed various heuristics, often with many decision points and parameters (e.g., which seed to mutate) that collectively determine their overall effectiveness. Fuzzer design choices often involve not only strong intuitions and domain expertise, but also much empirical testing and tuning. Often, the choices can cause an over-specialization for a particular set of target codebases used during the tuning process.

Although there are attempts to auto-tune various fuzzing decisions, including seed selection [26, 33, 35] and mutation operators [8, 9, 17, 22], prior efforts are mostly point solutions and none are specifically tailored for Operating System (OS) kernel fuzzing. Kernel fuzzing is uniquely challenging for the following reasons: (1) modern OS kernel often has a huge code base and many dependencies across components; (2) the input to an OS kernel is via the system call interface that needs special handling; and (3) an OS kernel maintains a massive state space that a single input (i.e., test case) may not be able reproducible. To illustrate, note that the state-of-the-art kernel fuzzer, Syzkaller [14] has over 62,000 lines of code and numerous parameters that are tunable to improve its efficiency. Given this large, complex space, and the ad-hoc strategies used to tune parameters, we believe that there are marked opportunities to improve kernel fuzzing.

To address the above challenges, Syzkaller uses a combination of generation [13] and mutation [5] based input crafting strategies. Specifically, to generate inputs (sequence of syscalls) from scratch, Syzkaller needs hand-crafted input models called "templates". It also takes known good inputs (aka. *corpus seeds*) that previously unearthed new code coverage, and *mutates* (i.e., modify) them to generate new ones. Finally, Syzkaller triages an input to ensure that a minimal input can reproduce the achieved coverage, before turning it into a seed. Syzkaller uses a fixed strategy to schedule these different types of fuzzing tasks, and a hard-coded strategy to select which seeds to mutate.

In this paper, we propose SYZVEGAS, a Syzkaller-based fuzzer, capable of dynamically and automatically adapting its strategies to improve coverage. Specifically, we focus on addressing the two aforementioned first-order decision-making processes: 1) selecting (scheduling) the most rewarding fuzzing tasks (e.g., generation, mutation, and triage) and 2) selecting the most potent seeds for mutation. Both of these are done dynamically in SYZVEGAS via a unified reward assessment model to significantly improve the odds of ex-

cavating new code coverage and finding new vulnerabilities. Our main contributions are:[1]

- **Identifying optimization opportunities.** We perform a systematic analysis of Syzkaller's default (fixed) task and seed selection policies. We identify several opportunities for improving Syzkaller's fuzzing efficiency.
- **Realizing dynamic fuzzing.** SYZVEGAS employs a lightweight Adversarial MAB algorithm to adjust the task and seed selection policies dynamically. It consists of a novel approach for fuzzing tasks reward modeling by consolidating the discovery of new coverage and the time cost incurred. The approach also accounts for the associations between different types of tasks, can quickly adapt during the different stages of fuzzing, and has very low overhead. To the best of our knowledge, SYZVEGAS is the first to 1) use the Adversarial MAB formulation and design reward functions that are applicable for task selection, and 2) incorporate the notion of time associated in unearthing new coverage in the reward function.
- **Improved coverage growth.** We perform extensive evaluations of SYZVEGAS on the latest Linux kernel and show that it consistently attains 38.7% more coverage than the default Syzkaller and finds more unique crashes. In total, we found 13 more crashes (8 unique) than Syzkaller in the same period. For OS kernels such as Linux, such an improvement makes a big difference as every kernel version is being constantly fuzzed and tested. (e.g., by Google [15]).
- **Applicability in user space.** We also demonstrate that the seed-selection module of SYZVEGAS can be applied to user-space as well and compares favorably to a state-of-art reinforcement-learning-based fuzzer, viz., EcoFuzz [34]

2 Background and Motivation

2.1 Syzkaller

Syzkaller explores the OS kernel by executing a series of test **programs**, i.e. a sequence of system calls. To craft such programs, Syzkaller has two options: generate a new program from scratch or mutate an existing program. It invokes three types of tasks during the fuzzing process: **Generation**, **Mutation** and **Triage** (more details in Section 8.1).

- **Generation**. Syzkaller creates a brand new test program using *templates*, which are manually curated by domain experts (e.g., kernel developers), and contain information on the argument type of each system call, and the dependencies between system calls (e.g., the return value of `open` is usable later in `read`). This allows Syzkaller to generate meaningful syscall sequences and arguments, improving the likelihood of exploring deeper kernel code.
- **Mutation**. Syzkaller randomly picks a program (also called a *seed*) from a *corpus* (i.e., programs that previously found new coverage), and performs a series of random mutations (e.g., inserting/removing a new syscall, or changing the argument of an existing syscall, using built-in templates) and executes the mutated program.
- **Triage**. Syzkaller fetches a newly Generated or Mutated program that has produced new coverage. It first performs "**Verification**" to ensure that the new coverage can be reliably reproduced, i.e., is unaffected by 1) the stateful nature of OS kernels (e.g., control flow affected by a global variable), 2) non-determinism in execution (e.g., mutex slow path, kmalloc cache replenish path) and 3) concurrency and interaction between several processes. If successful, Syzkaller then performs a "**Minimization**" of the program (remove of some system calls and/or shorten the arguments, while retaining the stable coverage) and adds the program to the seed corpus (where future mutations can be performed). During minimization, Syzkaller may discover that a partially minimized program can achieve new coverage; these programs are marked for later triage.

By default, Syzkaller selects the aforementioned three types of fuzzing tasks as per the following hard-coded priorities:

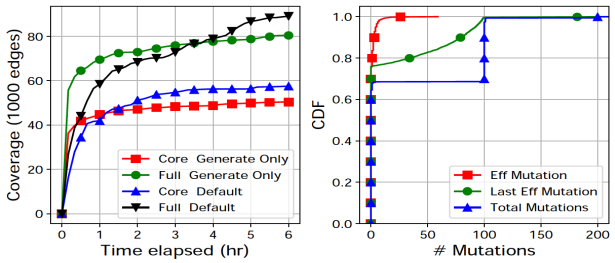
1. Triage is prioritized over generation and mutation.
2. When no triage task is available, the highest priority is to mutate programs that were just added to the seed corpus. Syzkaller mutates each new seed for a fixed number of (100) times. These mutations receive some special treatment and are called **Smash** in Syzkaller.
3. If no triage or smash tasks are available, Syzkaller executes generation and regular mutation tasks with a fixed 1:99 ratio (one generation task for every 99 mutation tasks).

In practice, upon starting from scratch, Syzkaller performs a generation task and some part of the kernel codebase is covered as a result. This very first program will then go through triage, producing the initial seed and potentially creating more programs for triage during minimization. Syzkaller will then focus on triaging these additional programs (if any from minimization) and smashing the new seeds, which in turn creates more seeds for smashing and programs for triaging. Proceeding in this manner typically leads to a huge chain reaction. As a result, the actual number of generations Syzkaller performs is much lower than the policy description may suggest.

When it comes to mutation, Syzkaller chooses which seed to mutate as per the following principles. First, as discussed, a newly created seed enjoys a high-priority invocation of 100 mutations, i.e., smash. Second, each seed is assigned a weight equal to the number of new and stable edge coverage it brings. This number is static and remains unchanged over time. When Syzkaller needs to pick a seed from the corpus, it does so on the basis of this weight, from among all the seeds.

Scheduling between different tasks is unique to Syzkaller as user-space fuzzers often 1) do not have well-defined templates

[1] Our system is completely open sourced at [20] to facilitate the reproduction of the results and future research.



(a) Coverage growth comparison (b) Mutation effectiveness

Figure 1: Evaluating default Syzkaller strategies.

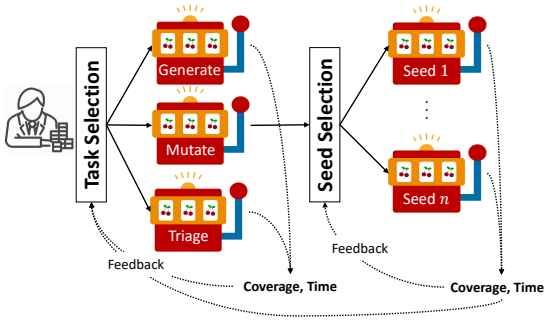


Figure 2: High-level idea/design of SYZVEGAS.

to perform Generation and 2) do not need to Triage as they are not affected by statefulness, non-determinism, or concurrency in OS kernels. Therefore, optimizations for user-space fuzzers often cannot be applied directly to Syzkaller.

2.2 Observations and Intuition

In this section, we motivate the use of a learning-based approach to improving Syzkaller's coverage. We run the default Syzkaller alongside a modified Syzkaller which performs only generations, on a 2-core single process fuzzer VM, for 6 hours. We collect metrics such as coverage growth and task effectiveness to gain insights into the former's operations. Our experiments are on a Intel(R) Xeon(R) E5-2680 v4 2.40 GHz CPU. Our experiment yields the following observations:

The best strategy evolves over time. Based on the task selection policy discussed earlier, Syzkaller gives a low priority to generation. However, with a well-written template, generation can be powerful, especially in the earlier stages of fuzzing where most of the kernel code is yet unexplored/uncovered.

Figure 1(a) compares the coverage growth with the two Syzkallers, when fuzzing 1) the full Linux kernel and 2) the core kernel excluding sub-systems such as filesystem and drivers. With the full kernel, we see that in the first 1 hour of fuzzing, the generation-only Syzkaller markedly outperforms the default Syzkaller. After 4 hours, however, the generation-only Syzkaller falls behind the default Syzkaller. When fuzzing the core kernel, it takes only 1 hour for the default Syzkaller to overtake the generation-only Syzkaller. These results suggest that the optimal strategy is dynamic, and must adapt over time, motivating a learning-based approach.

Ad-hoc decisions can be harmful. Syzkaller prioritizes the mutation of newly-discovered seeds, invoking a mandatory 100 mutations, to extract large coverage quickly. Further, Syzkaller's "task scheduling" logic uses LIFO stacks to ensure that the newest seeds are explored first. Undoubtedly, the domain experts behind Syzkaller chose this strategy carefully with extensive testing. However, this static (ad-hoc) decision has limitations. Figure 1(b) shows the mutation effectiveness of Syzkaller, fuzzing the whole Linux kernel, for 6 hours. We see that there are three opportunities for improvement: 1) Many seeds are not being mutated because Syzkaller is too busy performing the mandatory number of mutations and triaging. 2) We observe chain reactions where the 100 new "smash" mutations of a program discover new coverage and in turn schedule additional 100 new "smash" mutations immediately for each of these, causing focused exploration on seeds from the same roots; and, 3) Of the mandatorily mutated seeds, many do not deserve to be mutated 100 times. These behaviors seem to be unintended consequences of the ad-hoc (but perhaps empirically acceptable) decision to apply 100 mutations on each new seed. We tried providing some simple static adjustments (e.g. reduce the number of smash mutations, force some generations at the beginning), none of which produces desirable coverage improvement nor adaptability across different scenarios. (e.g. kernel version, initial seed corpus) Therefore, a learning-based technique is the best approach to overcome these, and thus improve the fuzzing effectiveness.

Intuition. Our observations indicate that there are many opportunities to tune various hard-coded parameters (e.g. mutation count, generation to mutation ratio) and priorities. They also suggest that the right strategy and the right seed dynamically change over time. What is needed is an automated way to identify the "most promising" task at the given time, and if appropriate the "best seed" to be invoked in association with that task. To identify these, a *reinforcement-learning* scheme is a natural fit, wherein a model to maximize the coverage rewards relative to the time cost of execution is applicable.

2.3 Multi-armed Bandit Problem

The *Multi-Armed Bandit* (MAB) reinforcement-learning problem is well suited to model the various decisions of Syzkaller. In this problem, a gambler must play a number of competing slot machine arms (choices) to maximize the expected gain. Each arm's properties are only partially known to begin with and may become better understood as the arm is played more. The MAB problem is a classic example of the tradeoff between *exploration and exploitation*.

One of the strongest generalizations of the MAB problem is

Table 1: Symbols we use to describe SYZVEGAS

Symbol	Description
c or c_i	Number of edge coverage attained by executing a single task (of type i).
t or t_i	Execution time of a single task (of type i).
C	Total edge coverage attained throughout fuzzing.
T	Total elapsed time of fuzzing.
t_{exp}	Estimated expected execution time of task/tasks.
g or g_i	Un-normalized reward attributed to task/tasks (of type i).
$c^p_{mut}(m)$	Total edge coverage of mutating a seed p for m times.
$t^p_{mut}(m)$	Total execution time of mutating a seed p for m times.
x	Normalized reward attributed to task/tasks.
$\hat{G}_i$	Accumulated reward estimation of MAB arm i.

Adversarial MAB, introduced in 1995 [6]. In this variant, the reward from each arm can be arbitrarily altered during each play. This requires its solution to react quickly to the changing rewards of each arm, which maps well to the fuzzing process where each decision can receive different rewards over time.

Auer et.al. proposed the *Exponential-weight algorithm for Exploration and Exploitation* (*Exp3*) [7] for the adversarial bandit problem. The idea is to introduce an exponential growth in a good arm's weight (i.e. probability of playing), thereby ensuring that good arms are quickly identified and exploited. Notable variants of *Exp3* algorithm includes *Exp3.1* [7] (resets periodically, performs better over time), *Exp4* [7] (allows for an additional advice input vector), *Exp3-M.B* [36] (playing multiple arms at the same time with a limited budget) and *Exp3-IX* [23] (uses implicit exploration).

Other reinforcement learning models exist [31], with different emphasis and strengths. We chose MAB because we believe it is a natural fit for our problem since its decisions are discrete, and the overhead is low. Since the reward of fuzzing choices can change over time (e.g., generating programs from scratch is only helpful early on, seed programs become decreasingly efficient as they are mutated), we argue that the Adversarial MAB is well suited to the fuzzing problem (as explored in previous researches such as EcoFuzz [34]). Our contribution is proposing the use of reinforcement learning in kernel fuzzing, rather than the specific learning algorithm. We leave exploring other learning strategies to future work.

3 Design and Implementation

We propose SYZVEGAS, a dynamic fuzzing approach to select between the three types of tasks in Syzkaller. The main design goals of SYZVEGAS are as follows:

- *Optimal coverage.* SYZVEGAS must schedule tasks or pick mutation seed programs to maximize the coverage achieved by Syzkaller, while minimizing the incurred time cost.[2]
- *Adaptive adjustment.* SYZVEGAS should determine which type of task is the best, at each stage of fuzzing, and adapt its strategy accordingly. When performing mutations, SYZVEGAS must assess the quality (change) of the mutated seed and adjust its priority in the seed corpus accordingly.

To achieve these goals, SYZVEGAS abstracts the task/seed selection problem as an Adversarial MAB problem as shown in Figure 2. The generation, mutation, and triage tasks are the three arms. When invoking mutation, SYZVEGAS treats seed selection as another layer of the MAB problem, i.e., each seed is treated as a separate arm. After each play, SYZVEGAS gathers the coverage and time cost and computes the feedback to the MAB decision process, using an algorithm similar to solutions such as *Exp3-IX* [23] (see Section 8.2 for more details) and *Exp3.1* [7]. As this process repeats, SYZVEGAS updates which arms should be played to maximize the coverage achieved per unit-time.

For SYZVEGAS to perform effective task and seed selection dynamically, the key challenges are: 1) assessing the value of the selected task or the mutated seed, 2) picking the task or seed with the maximum potential. We discuss how SYZVEGAS overcomes these challenges next. Table 1 lists the symbols used in subsequent sections for reference.

3.1 Reward Assessment

Whenever a generation/mutation/triage task has completed execution, we need to assign a reward to the task to be used with SYZVEGAS's Adversarial MAB model. The key requirements/challenges in computing this reward are as follows:

- *Gain and cost considerations.* The goal of SYZVEGAS is to maximize the gain (i.e. number of edges covered) while minimizing the cost (i.e., the time taken for execution). Our model must unify these metrics with different units into a single measure of the effectiveness (utility) of each task.
- *Dependencies between tasks.* Unlike what is assumed in a classic MAB problem the arms *are not* independent in the context of Syzkaller. As shown in Figure 10, there is a strong relationship between Triage and Mutation. SYZVEGAS needs to properly address this relationship when assigning rewards to each arm.
- *Normalization.* The utilities observed on different systems can be different. For example, the time it takes to execute a program on Android will be much longer than executing the same program on a powerful server. In addition, the algorithms that are used to solve an Adversarial Bandit problem often require the reward to be normalized.

To address these challenges, we build our reward assessment model as follows, considering each task of interest.

Generation. Generation is not directly intertwined with either mutation or triage. Thus, its reward is assessed independently. Let c be the new coverage (measured by the number of edges) obtained by generating a program. Let t be the cost in time of executing this program. Let C and T be the total achieved coverage (regardless of attribution to generation), and the total

[2] Syzkaller collects information relating to two types of coverage, viz., unstable and stable coverage. We design SYZVEGAS to optimize for maximum unstable coverage as both types of coverage can lead to crashes but unstable coverage is a superset over the stable coverage. See Section 8.1 for more details.

elapsed time from when the fuzzer began, respectively. Given these, the expected time for finding the new coverage c (given our average performance up to T), can be "estimated" by:

$$t_{exp} = c \cdot \frac{T}{C} \tag{1}$$

The reward for the generation task can be modeled as the expected time cost minus the actual time cost t:

$$g = t_{exp} - t = c \cdot \frac{T}{C} - t \tag{2}$$

Note that g essentially compares the coverage discovery rate of the current generation task (c/t) and the historical coverage discovery rate (C/T). If the task has a better-than-historic coverage discovery rate, it will always have a positive reward, while a worse-than-historic coverage discovery rate will cause a negative reward. This representation also ensures that if two tasks A and B both produce the same coverage c, but consume different times, say $t_A > t_B$, we always have $g_A < g_B$; intuitively a task that discovers coverage faster should be rewarded more. Note also that we use time instead of rate as the reward unit to ensure that if tasks A and B both produce no new coverage (happens often in later stages of fuzzing), we always have $g_A < g_B < 0$. In other words, a task that wastes more time is punished harder than one that wastes less time.

Mutation and Triage. Mutation tasks heavily depend on triage because: 1) the seed driving a mutation is only obtained via triage and 2) triage tries to minimize the seed, thus reducing costs for future mutations. Thus, the reward of mutation and triage must be modeled in conjunction. Consider a seed program p, where the time cost of the triage task that verified and minimized p is t^p_{tri}. As discussed in Section 2, triage consists of two phases viz., verification and minimization, costing t^p_{ver} and t^p_{min} respectively, with $t^p_{ver} + t^p_{min} = t^p_{tri}$. In minimization, triage first receives a generated/mutated program p' (costing $t^{p'}$ to execute) and "minimizes" it to p (costing t^p) by removing system calls and/or shortening arguments. Thus, the time saved from minimization is $\Delta^p_t = t^{p'} - t^p$. Finally, minimization may also discover new coverage c^p_{min}.

The verification phase may also produce new coverage from simply re-executing the original program. However, since this new coverage was not observed in the prior execution of the same program, the input program in this form is unstable (the coverage is not reproducible) by definition. As a result, Syzkaller does not attempt to process such new coverage possibilities. We follow Syzkaller's design on this matter i.e., ignore new coverage possibilities from verification.

The seed program p, is then mutated m times. The observed edge coverage with each mutation are $c^p_1, c^p_2, ...c^p_m$, while the time costs associated with each of these mutations are $t^p_1, t^p_2, ..., t^p_m$, respectively. For simplicity, we denote:

$$c^p_{mut}(m) = \sum_{j=1}^{m} c^p_j \;\;,\;\; t^p_{mut}(m) = \sum_{j=1}^{m} t^p_j. \tag{3}$$

Note here that without minimization, Syzkaller can only mutate from p' instead of p. In this case, on average, each mutation should take Δ^p_t longer; thus, minimization results in a total of $m \cdot \Delta^p_t$ time savings, over m mutation tasks. If we treat the one triage and m mutations as a single task, the expected time to discover the new coverage of $c_{mut}(m)$ without minimization can be computed by:

$$t^p_{exp} = \left(c^p_{min} + c^p_{mut}(m)\right) \cdot \frac{T}{C} + m \cdot \Delta^p_t \tag{4}$$

The first part of the right hand side of the equation, estimates the total expected time to discover the new coverage $c^p_{min} + c^p_{mut}(m)$ by mutating p; the second part is the estimated time savings from minimization. Now, the total reward from triaging and mutating seed p is the difference between the "expected and actual time" utilities and is given by:

$$g^p_{tri+mut} = \left(c^p_{min} + c^p_{mut}(m)\right) \cdot \frac{T}{C} + m \cdot \Delta^p_t - \left(t^p_{tri} + t^p_{mut}(m)\right) \tag{5}$$

We reiterate here that since the main contribution of minimization is to save time in future mutations, the time savings part of Equation 5 must be fully credited to minimization. In addition, minimization is also finding new coverage c_{min} from testing minimized programs. Combining them both, we can thus estimate the reward attributed to minimization as:

$$g^p_{min} = c^p_{min} \cdot \frac{T}{C} + m \cdot \Delta^p_t - t^p_{min} \tag{6}$$

Verification is needed for creating the seed p (without it mutation will have no seeds to mutate). Thus, verification and mutation should share the reward of finding new coverage, proportional to their costs. Hence, the reward attributed to verification and mutation are:

$$g^p_{ver} = c^p_{mut}(m) \cdot \frac{t^p_{ver}}{t^p_{ver} + t^p_{mut}(m)} \cdot \frac{T}{C} - t^p_{ver} \tag{7}$$

$$g^p_{mut} = c^p_{mut}(m) \cdot \frac{t^p_{mut}(m)}{t^p_{ver} + t^p_{mut}(m)} \cdot \frac{T}{C} - t^p_{mut}(m) \tag{8}$$

Adding Equations 6 and 7, we obtain the total reward attributed to triage as:

$$g^p_{tri} = \left(\frac{c^p_{mut}(m) \cdot t^p_{ver}}{t^p_{ver} + t^p_{mut}(m)} + c^p_{min} \right) \cdot \frac{T}{C} + m \cdot \Delta^p_t - t^p_{tri} \tag{9}$$

Note that Equations 8 and 9 are only approximate estimates of the rewards with mutation and triage, respectively. In practice, it is difficult if not impossible to predict how many times a seed program p will be mutated. In addition, it is impractical to compute the reward after all mutations are complete. Every time a seed program p is mutated, we need to update the weight of the triage and mutation arms. To achieve this goal, we first compute the reward for triage and mutation when seed p is added to the corpus via triage as: $g^p_{tri}(0) = c^p_{min} \cdot \frac{T}{C} - t^p_{tri}$ (as the reward of performing the triage task alone) and $g^p_{mut}(0) = 0$. As p is mutated, we keep track of the observed new coverage and time costs.

Updating rewards. For the k^{th} mutation, we estimate the total reward $g^p_{tri}(k)$ and $g^p_{mut}(k)$ using Equations 9 and 8. We then compute the difference with respect to the estimated total reward after the $(k-1)^{th}$ mutation step, as $\Delta(g^p_{tri}, k) = g^p_{tri}(k) - g^p_{tri}(k-1)$ and $\Delta(g^p_{mut}, k) = g^p_{mut}(k) - g^p_{mut}(k-1)$.

Algorithm 1 Task selection Algorithm

for all $r = 1, 2, \ldots$ **do**
  $\hat{G}_{gen}(0), \hat{G}_{mut}(0), \hat{G}_{tri}(0) \leftarrow 0$
  $t \leftarrow 0$
  $\gamma \leftarrow 2^{-r}$
  $\eta \leftarrow 2 \times \gamma$
  $\hat{G}_{threshold} \leftarrow \frac{3ln3}{e-1} \cdot 4^r - \frac{1}{3\gamma}$
  while $max_i(|\hat{G}_i|) < \hat{G}_{threshold}$ **do**
    $w_i(t) \leftarrow e^{\eta \hat{G}_i(t)}$
    $pr_i(t) \leftarrow \frac{w_i(t)}{\sum_j w_j(t)}$
    Draw i_t according to $pr_{gen}(t), pr_{mut}(t), pr_{tri}(t)$
    if $i_t = gen$ **then**
      Receive reward $x_{gen}(t)$
      $\hat{G}_{gen}(t+1) \leftarrow \hat{G}_{gen}(t) + x_{gen}(t)/(pr_{gen} + \gamma)$
    else if $i_t = tri$ **then**
      Receive initial reward for triage $x_{tri}(s)$
      $\hat{G}_{tri}(t+1) \leftarrow \hat{G}_{tri}(t) + x_{tri}(t)/(pr_{tri} + \gamma)$
    else if $i_t = mut$, Seed s is selected **then**
      Receive reward detlas $x_{mut}(t), x_{tri}(t)$
      $\hat{G}_{tri}(t+1) \leftarrow \hat{G}_{tri}(t) + x_{tri}(t)/(pr_{mut} + \gamma)$
      $\hat{G}_{mut}(t+1) \leftarrow \hat{G}_{mut}(t) + x_{mut}(t)/(pr_{mut} + \gamma)$
    end if
    $t \leftarrow t + 1$
  end while
end for

We then use $\Delta(g^p_{tri}, k)$ and $\Delta(g^p_{mut}, k)$ as the reward for the triage and mutation tasks at the kth mutation, respectively; this is used later in our task selection algorithm (Section 3.2).

Normalization. The rewards g for generation, mutation and triage tasks can take values from $(-\infty, \infty)$. However, single-factor algorithms such as Exp3, Exp3.1 and Exp3-IX require the reward be normalized to $[0, 1]$. For budget-constrained algorithms such as Exp3-M.B. [36], both the gain and cost are normalized to $[0, 1]$, and the resulting (gain - cost) $\in [-1, 1]$. The *Logistic function* $1/(1+e^{-y})$ is commonly used for a normalization from $(-\infty, \infty)$ to $(0, 1)$ [32]. We rescale the logistic function from $(0, 1)$ to $(-1, 1)$ as $(1-e^{-y})/(1+e^{-y})$, ensuring that a zero reward is always normalized to 0. In order to account for the variations, we use $z' = g/\sigma_g$, a shifted version of standard Z-score to replace the y in the logistic function. We shift $z = (g - \overline{g})/\sigma_g$, the standard Z-score with a mean of $\overline{g}$ to a mean of 0, in order to make sure that a positive reward g is always normalized to a positive normalized reward x. The final normalization equation is:

$$x = \frac{1 - e^{-g/\sigma_g}}{1 + e^{-g/\sigma_g}} \tag{10}$$

3.2 Task Selection

Now that we have our reward functions, we leverage *Exp3.1* [7] and *Exp3-IX* [23] to determine which task of Syzkaller to invoke at each stage. We incorporate the exponential weight growth and the implicit exploration of *Exp3-IX* to ensure sufficient exploitation of the good arms and rapid adaption to changing rewards with regards to the different arms. We combine this with *Exp3.1* to periodically reset the weight of each arm and adjust the exploration and growth factors, ensuring the stability of the algorithm over an extended (infinite) period. Finally, we combine these with our novel reward assessment model (from Section 3.1) to address the association of mutation and triage tasks. SYZVEGAS's task selection algorithm is shown as Algorithm 1.

Similar to *Exp3.1*, the algorithm divides the fuzzing timeline into epochs (automatically determined by Algorithm 1), indexed by r. Epochs dictate when to reset the weights of the arms (required in Exp3.1). Our algorithm estimates a target reward $\hat{G}_{threshold}$ for each epoch, and tunes the exploration/growth factors γ and η as in *Exp3.1*. Within each epoch, our algorithm performs arm selection and reward updates similar to *Exp3-IX*. Upon each update, it detects if the estimated gain $\hat{G}_i$ exceeds the threshold. If so, a transition is made to the next epoch resetting the observed gains $\hat{G}_i$s to zero and increasing $\hat{G}_{threshold}$ by 4 $\times$ (for the next epoch).

A major difference between a traditional MAB solution and SYZVEGAS is the division of the reward between the triage and mutation functions. The *Exp3* algorithms assume that the arms are independent and thus, when an arm is pulled only its own weight is affected. However with SYZVEGAS (see Section 3.1), when the mutation arm is pulled, the weight of both the mutation and triage arms are updated. A second difference with the *Exp3*-like algorithms is that the normalized reward $x_i \in (-1, 1)$ in SYZVEGAS (in *Exp3*-like algorithms, the rewards are often normalized to $[0, 1]$). As discussed in Section 3.1, our design choice is driven by two intuitive reasons: 1) we do not want the arms (tasks) that produce no coverage to receive any gains in weight, and 2) when comparing tasks that produce no coverage, we want to punish those tasks that cost more, harder, which a $[0, 1]$ normalization cannot achieve.

Algorithm 2 Seed Selection Algorithm

Require: $\theta \in (0, 1)$
for all $t = 1, 2, \ldots$ **do**
  $K \leftarrow$ number of seeds.
  $\eta = 2\gamma = \theta \sqrt{\frac{2 ln K}{K}}$
  $w_i \leftarrow e^{\eta \hat{G}_i}$
  $pr_i \leftarrow \frac{w_i}{\sum_j w_j}$
  Draw seed i_t randomly according to pr_i
  Receive reward $x_i(t)$
  $\hat{G}_i(t+1) \leftarrow \hat{G}_i(t) + \frac{x_i(t)}{pr_i + \gamma}$
end for

3.3 Seed Selection

In addition to choosing the right task, the proper seed has to be associated with each mutation task. Towards this, we again use an *Exp3-IX*-like algorithm, shown in Algorithm 2.

While the seed selection algorithm is similar to the one for task selection in that it includes a reward assessment model, a normalization for the reward and a weight update process, there are some key differences.

The reward assessment model only considers mutation tasks. When a mutation task is finished, we reuse the gain/loss model from Section 3.1 to compute the reward of mutating the current seed. However, since our focus is now only on mutation tasks, we no longer split the reward with triage (as with task selection). Instead, we compute the reward in the same way as Equations 1 and 2, and no longer consider the rewards from generation and triage, in normalization.

Let C_{mut} and T_{mut} be the total achieved coverage and the elapsed time, for all mutation tasks. Let c_i and t_i be the achieved coverage and elapsed time for mutating a seed i. The observed gain of mutating this seed can thus be computed as:

$$g_i^{(ss)} = c_i \cdot \frac{T_{mut}}{C_{mut}} - t_i \tag{11}$$

Let $\sigma_{mut}^{(ss)}$ be the standard deviation of the observed gain across all mutation tasks; the final reward of mutating seed i is then:

$$x_i^{(ss)} = \frac{1 - e^{-g_i^{(ss)}/\sigma_{mut}^{(ss)}}}{1 + e^{-g_i^{(ss)}/\sigma_{mut}^{(ss)}}} \tag{12}$$

Ever-increasing number of arms. Syzkaller starts with no seed in the corpus, which is only populated as Syzkaller creates and executes more and more programs. Thus, if we treat the seed selection problem as a MAB problem, we may have an ever-increasing number of arms. This is not typical in classic MAB problems, but we can make adjustments to fit our problem. Specifically, when a new seed i is added to the seed pool, it starts with a neutral accumulated estimated reward $G_i^{(ss)} = 0$. As a result, its initial weight $w_i^{(ss)}$ will be 1 (in accordance with Algorithm 2). The probability of selecting this seed will initially depend on the accumulated rewards (e.g., $G_j^{(ss)}$) of other seeds already in the corpus. Once seed i is later mutated, the probability of picking seed i will be determined by whether the benefits of attained coverage out-weigh the time cost. In addition, as more seeds are being added, we reduce the exploration and growth factors of our algorithm to ensure these do not thus dominate the probability of picking a single seed (which is decreasing with more seeds).

Reset is not necessary. Since the mutation process is random, the more a seed program is mutated the less likely that future mutations of that program will lead to the discovery of new coverage. Hence, each arm in the seed selection MAB has a diminishing reward. Consequently, there is no point in adopting the *Exp3.1*-style reset mechanism for the seed selection algorithm (since seeds die out). Our seed selection algorithm simply follows the *Exp3-IX* algorithm, with the only exception being that new arms are created once a new seed has been added to the corpus.

3.4 Implementation

Our implementation of SYZVEGAS incorporates our reward assessment models and the previously discussed extensions of the *Exp3.1* algorithm on top of Syzkaller. (based on commit `1128418` on 05/24/2020 [14]). Our implementation consists of roughly 1,800 lines of code. Below, we describe some of the subtleties we handled in our implementation.

Standard deviation computation. During normalization, we need to compute the standard deviation of all previously observed rewards as shown in Equation 10 and 12. Keeping track of all the reward values is impractical (as Syzkaller can execute millions of programs). In addition, these numbers need to be synced with the host machine and restored if the fuzzer VM/device crashes or disconnects. Fortunately, we only need to keep track of 1) the total number of observations n, 2) $\sum g$ and 3) $\sum g^2$. We can then compute the standard deviation as:

$$\sigma(g) = \sqrt{E(g^2) - E^2(g)} = \sqrt{\sum g^2/n - (\sum g/n)^2}. \tag{13}$$

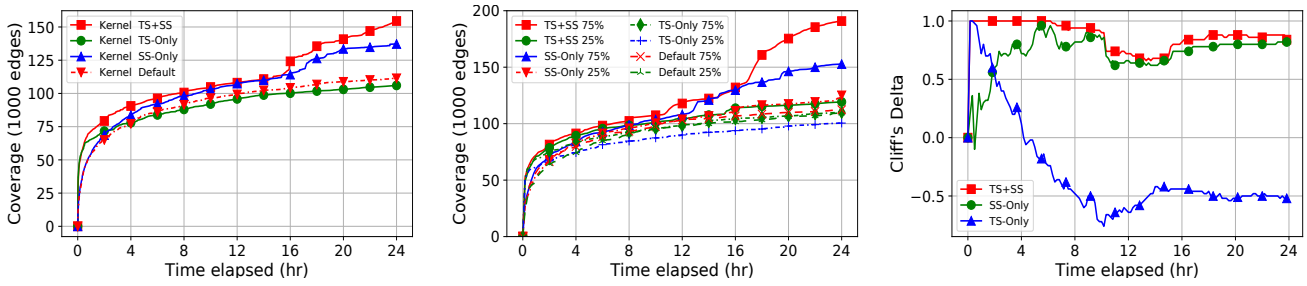
Outlier Handling. Programs on the fuzzer VM/device can consume different execution times. In some cases, a program can take several seconds for execution. Although this happens rarely, a mere (insignificant in number) few extreme cases can severely throw off the time estimation and standard deviation, hurting our task selection and seed selection algorithms. Thus, it is crucial that we detect these outliers and prevent them from damaging the integrity and effectiveness of our algorithms.

Our measurements show that triage is the most costly task and its execution time can vary greatly. If we use the "3rd quartile + interquartile range" method to detect outliers, we would set the threshold at 0.32 seconds. To allow some slack without compromising experimental integrity, we set one second as the outlier detection threshold. For any task that costs more than one second, we treat it as being executed in one second and proceed as normal. In practice, less than 1% of all tasks need to have their cost adjusted.

4 Evaluation

We conduct extensive evaluations of SYZVEGAS with different configurations, and by default comparing it with Syzkaller as a baseline. Unless otherwise stated, each configuration is run 10 times with one fuzzer VM that uses 2 cores and 2 GB memory, on a server with Intel Xeon Gold 6248 2.50GHz CPUs. For seed selection, we choose $\theta = 0.1$ for all experiments. The key experiments we perform are:

- A 24-hour experiment on a Linux kernel from scratch, with a comprehensive analysis of the results.
- A 24-hour experiment on a Linux kernel using an initial seed corpus to study the impact on coverage growth.
- A 24-hour experiment on multiple Linux kernel versions, to study how SYZVEGAS auto-adapts to different kernels.
- A 7-day experiment to study long-term effects and crashes.



(a) Median (b) 25 and 75 percentile (c) Cliff's delta vs Default

Figure 3: Median, 25/75 percentile and Cliff's delta of coverage reached for fuzzing Linux kernel for 24hrs. Comparison of SYZVEGAS with/without task selectiion (TS) and seed selection (SS).

- 24-hour experiments comparing SYZVEGAS with state-of-art fuzzers HFL [18] and EcoFuzz [34].

4.1 Fuzzing the Linux Kernel for 24 hours

First, we conduct a 24-hour fuzzing experiment on the full Linux kernel version 5.6.13 to perform a systematic in-depth evaluation and analysis of SYZVEGAS.

Coverage growth. Figure 3(a) shows the median coverage growth reached after fuzzing the Linux kernel for 24 hours. Figure 3(b) shows the 25 and 75 percentiles instead. From these two figures, we make several interesting observations:

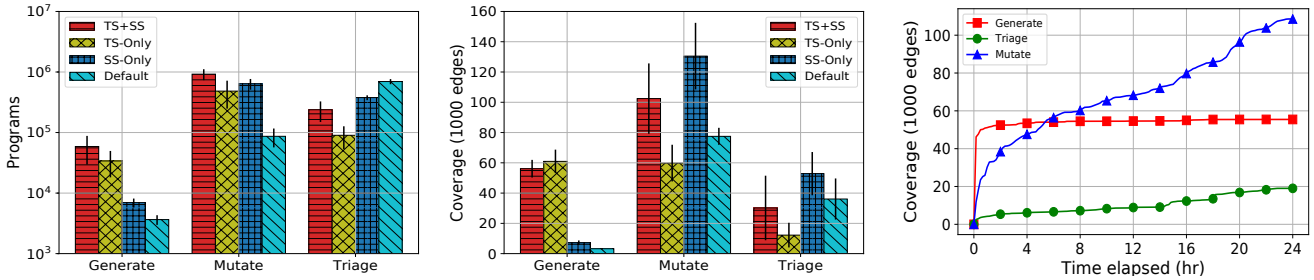
- MAB task selection works best in the early stages of fuzzing. However, the initial advantage is lost as fuzzing reaches its later stages.
- MAB seed selection has little effect in the first few hours. However, as we run for longer, seed selection begins to increase coverage growth, providing an improvement of 25,830 edges (23.2%) at 24 hours, in terms of the median.
- Combining MAB task and seed selection produce considerable improvements in the coverage growth, improving the median coverage by $\approx$ 43,130 edges (38.7%) at 24 hours. Interestingly, while MAB task selection does not provide an advantage by itself at 24 hours, combining it with MAB seed selection yields additional coverage improvement.
- With seed selection, the variation in coverage is much higher. This is because SYZVEGAS's seed selection truly picks seed with weighted randomness, while in vanilla Syzkaller, deterministic smash mutations dominate seed selection (as discussed in Section 2.2).

Since luck plays a prominent role in the coverage growth of fuzzing, researchers propose that statistical methods be used to determine the likelihood of the observed differences in coverage [19]. To evaluate whether the coverage advantage of SYZVEGAS is consistent across all runs, we compute Cliff's delta [10] between runs with MAB task and/or seed selection against the default Syzkaller. Cliff's delta lies in the range $[-1, 1]$ and represents the pair-wise comparison result between runs (in our case between our setup and the default Syzkaller). A higher Cliff's delta means that our setup is more likely to outperform the default Syzkaller. Figure 3(c) demonstrates that SYZVEGAS with SS-Only and TS+SS can reliably beat the default Syzkaller, verifying our observations in Figure 3(a) and 3(b) with high confidence.

Another interesting observation is that the power of seed selection really starts to kick in at around 12-14 hours of fuzzing. Earlier, SYZVEGAS only has a small lead in coverage, over the default Syzkaller. A closer look shows that at 12-14 hours, most existing seeds are already heavily exploited and seed selection assigns negative rewards to them (i.e., gives them very low priorities). Seed selection immediately favors a new seed(s) and extends its priority if it produces good coverage. Syzkaller, in contrast, is negatively impacted by the 100-new-seed-mutation policy. As discussed in relation to Figure 1(b), it causes a huge workload backup on mutating new seeds, which have more potential than older-spent ones. Even after new seeds get their 100 mutations, they will compete with the old seeds based solely on the coverage achieved **initially**, i.e., the coverage achieved by mutating them is disregarded. Thus, SYZVEGAS better utilizes new seeds and increases the chance of "unlocking" new kernel code blocks.

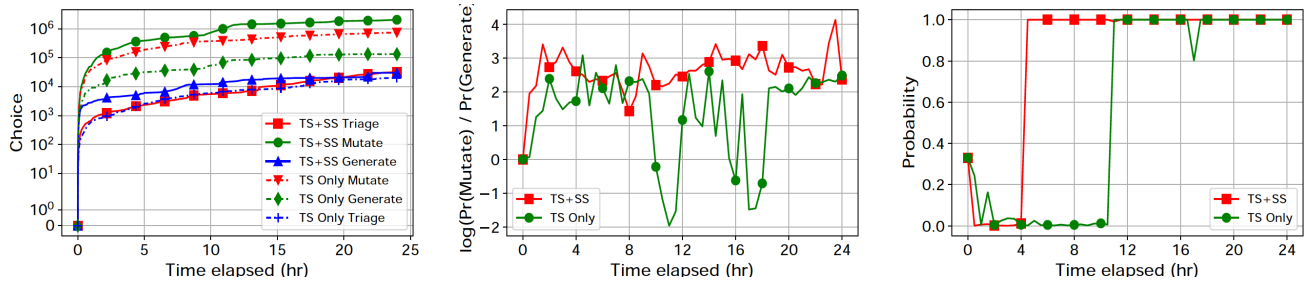
Figure 4(a) shows the number of programs executed by different types of tasks. Understandably, all of our optimizations generate more programs by giving a higher priority to generation and/or removing the mandatory smash mutation. An interesting observation is that with MAB-based seed selection, SYZVEGAS executes more programs in total than the default Syzkaller. This is primarily due to favoring the mutation of seeds with low execution times (i.e., allowing SYZVEGAS to perform more mutations). This reflects SYZVEGAS's design goal of optimizing coverage-time efficiency of tasks.

Figure 4(b) breaks down the coverage by the task types. Based on our observations, MAB task selection significantly shifts the workload from mutation to generation, giving generation a 20 times boost in terms of the coverage found. This comes with a sacrifice though, in the form of a 50% reduction in coverage discovered by mutations. Fortunately, seed



(a) Programs executed (b) Coverage reached (c) Coverage growth by task for SYZVEGAS

Figure 4: Statistics of program execution.



(a) Task choices made (b) Generation vs mutation probabilities (c) Triage probabity

Figure 5: Statistics of MAB task selection.

selection compensates for this loss, bringing the power of mutations back to its original level.

Interestingly, we find that when MAB task selection is present, generation produces a huge amount of coverage. However, when we take a look at the number of programs executed (Figure 4(a)), SYZVEGAS still favors mutation. If we break down the coverage achieved by the different tasks over time, as shown in Figure 4(c), we observe that the coverage reached by generation is almost exclusively achieved in the first 2 hours when the kernel code space is not explored much, and finding new coverage is simple. This, as we will show later, is due to the fact that task selection performs plenty of generation at the early stages of fuzzing.

MAB Task selection. Next, we take a look at the inner workings of MAB task selection. In particular, we want to understand "how much is the probability assigned by the task selection algorithm to each type of task."

Figure 5(a) captures the choices made by the task selection algorithm. We observe that at the beginning of fuzzing, task selection quickly "pulled" the generation "arm" (more than 1000 times), giving generation much higher priority than the default Syzkaller. Triage, on the other hand, is less-favored compared with generation at the beginning, but it starts to slowly catch up as fuzzing goes on.

Figure 5(b) illustrates how MAB task selection balances generation and mutation over time, with or without MAB seed selection. To begin, generation and mutation are initialized to have the same probability. With the help of the associated seed selection, the task selection algorithm quickly determines that mutation is the better option, giving it around a 500 times higher likelihood. Without seed selection, however, the task selection algorithm favors generation much more, even giving it a higher probability of being invoked than mutation, occasionally. This is expected due to the issues from the default seed selection algorithm, as discussed in Section 2.2. Without the improved seed-selection algorithm, mutations are less effective in finding new coverage and thus fall out of favor.

Figure 5(c) shows the probability change over time for triage. Triage is not always available (when no more interesting programs are in the work queue), and thus, Figure 5(c) only considers its probabilities when it is available. In the beginning, task selection gives triage a few chances before assigning it a very low priority, favoring generations and mutations, much more. At this stage, generation and the initial seeds (accumulated from the few triage tasks) are still very powerful, causing the task selection algorithm to give generation and mutation higher probabilities. However, as these initial seeds lose power and generation becomes less effective, both generation and mutation accrue negative rewards (no new coverage yet but time costs are incurred). Triage will then be favored naturally. Its ability to generate new seeds and maintain a diverse seed pool becomes essential to discovering new coverage. This effect is especially prominent when there is MAB seed selection to make mutation more effective

(while the initial seeds exhaust power quickly), causing triage to be invoked earlier on. Thanks to its exponential weight growth feature, SYZVEGAS quickly adjusts its policy giving triage the absolute priority (when appropriate) just like the default Syzkaller. Note that a near 100% triage probability does not mean SYZVEGAS will only perform triages. Triage tasks are created by generation and mutation and are not always available. When SYZVEGAS has no more triage tasks to schedule, it will perform generations or mutations.

Surprisingly, according to the task selection algorithm, the power of generation and initial seeds can last as long as 4 to 10 hours, and the default Syzkaller does not exploit this as much. As Syzkaller evolves with improved templates and mutation strategies, the power of generation and mutation may change as well, making auto-tuning task selection the best longer-term option (instead of hand-picking a threshold).

Overall, we find that the main effects of MAB task selection are **performing more generations** and **deferring triages** at the very early stages of fuzzing. After a few hours, however, MAB task selection eventually converges to the same policy of the default Syzkaller. Triage takes absolute priority, while mutation tasks are heavily favored over generation tasks. This behavior is the most prominent when combined with seed selection, where mutations are more rewarding.

We now examine why combining MAB task selection and seed selection significantly outperforms MAB seed selection only, even when the latter is losing its effectiveness and converging toward a policy similar to that of the default Syzkaller. As discussed before, the main effect of task selection is performing more generation tasks and fewer triage tasks at the early stages of fuzzing, which heavily impacts the initial seeds added into the corpus. Researchers have demonstrated the benefits of choosing good initial seeds on kernel fuzzing [24]. For the same reasons, the early-stage behavior of SYZVEGAS which populates the corpus with good seeds, yields long-term benefits, which we will explore further in Section 4.2.

Seed power. Mutation, the main workhorse of finding new coverage, requires seed programs to function. Therefore, the "power" of seeds, i.e., how much coverage a seed can produce through mutation, has a huge influence on fuzzing efficiency.

Figure 6(a) shows the number of seeds generated by the fuzzer through the 24 hours. We find that with MAB task selection, Syzkaller produces much fewer seeds. Figure 6(b) depicts the distribution of new coverage attained by mutating these seeds, a.k.a. seed power. As expected, the MAB seed selection improves seed power by preferring good seeds for mutation. Interestingly, we see that adding MAB task selection *improves the seed power*, despite not directly affecting seed selection. Thus, the coverage benefits induced by MAB task selection must come from its contribution to seed quality; this is where the initial generations invoked by MAB task selection help (by creating some very powerful seeds).

We break down the seed power distribution (how much new coverage a seed yields) based on the origin of the seeds in Figures 6(c). We see that task selection improves the power of the seeds originating from generation. This verifies our hypothesis and validates our motivation: having more early generations is beneficial to the Syzkaller fuzzing process.

Performance overhead. Finally, we evaluate the overheads of the MAB task selection and seed selection algorithms. The overheads are from two sources: 1) computing and updating weights and probabilities for tasks and seeds and 2) synchronizing the MAB status between the fuzzer VM and the manager host. The latter is closely related to how often the fuzzer crashes, as when does, it needs to fetch all information about the seed corpus the from manager host, again. During the 24 hour experiment, updating costs around 9 minutes while synchronizing costs 22 minutes. Overall, the overhead of SYZVEGAS is less than 2.1%.

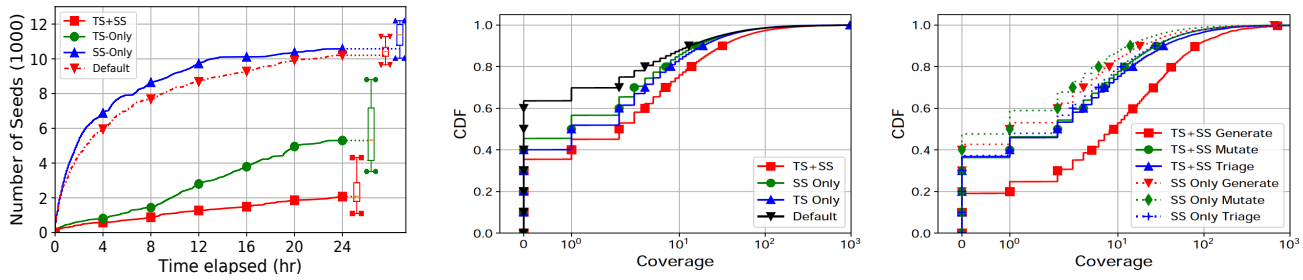
When it comes to memory, SYZVEGAS needs to store some additional information such as the weights of the arms, the total reward thus far, etc.. Copies of these records must be maintained by each fuzzer VM and the manager host, in case the fuzzer crashes. For task selection, we use a constant 250 bytes to store the necessary information. For seed selection, we use 104 bytes of additional memory for each seed. With ∼5,000 seeds created by SYZVEGAS in 24 hours, we incur ∼520 KB of memory overhead per VM/manager.

4.2 Fuzzing with Various Setups

Fuzzing With Initial Seed Corpus. Kernel fuzzing is often performed with an initial seed corpus. This lowers the number of programs Syzkaller needs to generate at the beginning and improves its coverage growth rate. To evaluate SYZVEGAS in such cases, we create two seed corpora. The first is created by running the default Syzkaller from scratch for 24 hours and contains 7478 seeds with 17149 syscalls. The second is from Moonshine [24], which analyzes the syscall traces from Linux Testing Project (LTP) [12], kselftest [1] and Open Posix Tests [2]. Specifically, we obtained traces from the authors directly to generate the Moonshine corpus consisting of 561 seed programs with a total of 8127 system calls. We run SYZVEGAS and Syzkaller (10 instances each) with and without each corpus for 24 hours, result in Figure 7(a).

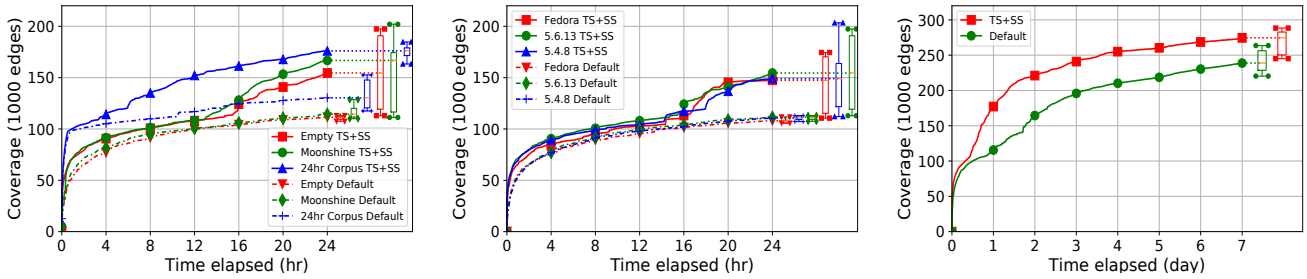
We find that the initial seed corpus yields limited benefits for the default Syzkaller. With the 24 hr seed corpus, the coverage spikes at first when Syzkaller imports and triages seeds, but flattens out later due to inefficient usage of these seeds. Interestingly, Moonshine offers almost no gains.

We find that importing the 24-hour seed corpus directly results in over 109,000 branch coverage, while the Moonshine corpus is only directly responsible for 33,700. As discussed in Sections 2.1 and 4.1, Syzkaller leaves a large number of seeds un-mutated due to its depth-first exploration of a small number of seeds as roots. As a result, the majority of imported seeds (the front ones) will never get a chance to be explored. Moreover, the current Syzkaller seed selection strategy only



(a) Seed number growth over time. (b) Seed power comparison. (c) Breakdown: TS + SS vs SS only

Figure 6: Coverage gained (seed power) by mutating seed programs.



(a) Effect of initial seed corpus. (b) Different kernel versions (c) 7-day experiment

Figure 7: Median coverage reached by fuzzing (a) Linux kernel 5.6.13 with/without initial seed corpus, (b) Linux kernel 5.6.13, 5.4.8 and Fedora kernel 5.8.0-rc1 for 7 days.

prioritizes seeds with higher initial coverage (without mutation); many imported seeds will not get mutation chances even when the workqueues are cleared, due to some other seeds with disproportionately high initial coverage. Consequently, seeds in the initial seed pools are severely under-utilized. In our experiments with both corpora, initial seeds only get < 2 mutations on average. The only benefit of having an initial seed pool for Syzkaller is the coverage achieved from executing it, not mutating it, which is why the 24-hour corpus (with more raw coverage) outperforms the Moonshine corpus. We note that when Moonshine was developed and tested, Syzkaller did not differentiate seeds, i.e., they are equally likely to be picked for mutation. This is a major difference that allows Moonshine seeds to bring in more benefits.

In contrast, SYZVEGAS makes better use of both initial corpora. Compared to the default Syzkaller with the same corpus, it achieves a median of 52416 (45.8%) more coverage with Moonshine and 45752 (35.1%) with the 24 hr corpus. Compared with the vanilla SYZVEGAS, Moonshine and the 24 hr corpus yield 12230 (7.9%) and 21564 (14.0%) more coverage, respectively. Although SYZVEGAS still suffers from the slow-import problem of Syzkaller, its seed selection strategy is smarter and makes better use of the initial corpus (Moonshine: ~120 mutations per seed; 24-hour: ~40 mutations per seed). The better utilization of Moonshine seeds also makes it more cost-effective compared to the 24hr corpus – much smaller but still yields significant coverage gains. Interestingly, the variation seen with SYZVEGAS with a 24-hour corpus is much lower. We believe that this is because this corpus is more saturated and the choices with regards to good seeds are much more limited.

Fuzzing Different Kernel Versions. We test the generalizability of SYZVEGAS by fuzzing various kernel variants. In addition to the upstream kernel in Section 4.1, we run similar experiments on 1) Linux kernel version 5.4.8, 2) Fedora kernel version 5.8.0-rc1, All fuzzing experiments are run on the same server mentioned in Section 4.1. Figure 7(b) demonstrates the median coverage growth comparison between SYZVEGAS and the default Syzkaller on these kernels. We see the effectiveness of SYZVEGAS consistently across all tested kernels. This is expected since SYZVEGAS's Adversarial MAB model requires no offline training and adjusts entirely online based on observed coverage yields and time costs.

Fuzzing Linux Kernel for 7 days. To study the long-term performance of SYZVEGAS, we run a 7-day fuzzing experiment on the full Linux 5.6.13 kernel. Figure 7(c) shows the median coverage growth, with 10 runs for each setup (same as before). Compared to Syzkaller, SYZVEGAS produces 35,736 (15.0%) more branch coverage, in the median case. We observe that SYZVEGAS is most effective in the first 24-48 hours of the fuzzing. Long-term, SYZVEGAS is still

Table 2: Crashes discovered fuzzing Linux kernel for 7 days.

Crash Reason	Function	# runs discovered	
		TS+SS	Default
Protection fault	kmem_cache_alloc*†	1	0
Protection fault	wait_consider_task*	0	2
RCU stall	ext4_file_write_iter*	0	1
RCU stall	io_uring_release*	10	10
RCU stall	io_uring_setup^R	4	6
RCU stall	tty_write^R	1	4
Slab out-of-bounds	do_update_region×	1	0
Slab out-of-bounds	vcs_scr_readw*†	1	0
Slab out-of-bounds	vgacon_scrolldelta	1	2
Slab out-of-bounds	vgacon_scroll×	9	10
Use-after-free	ata_scsi_mode_select_xlat×	2	0
Use-after-free	clear_buffer_attributes	1	0
Use-after-free	complement_pos	1	0
Use-after-free	con_scroll	2	0
Use-after-free	do_update_region×	7	7
Use-after-free	screen_glyph×	1	0
Use-after-free	screen_glyph_unicode×	1	0
Use-after-free	vc_do_resize*	6	1
Use-after-free	vcs_scr_readw†	1	0
Use-after-free	vc_uniscr_check	0	1
Use-after-free	vgacon_invert_region†	5	2
Use-after-free	vgacon_scroll×	1	0
Use-after-free	do_con_write×	3	4
Warning	dev_watchdog*	1	1
Warning	generic_make_request_checks*	4	3
Warning	xfrm_policy_insert_list*†	6	3
Total	TS+SS: 24, Syzkaller: 16	70	57

At the time of running this experiment (June 2020): *: Reported by `syzbot` [15]. ×: Reported by other sources. †: Closed. ^{R}: Reproducible new crashes.

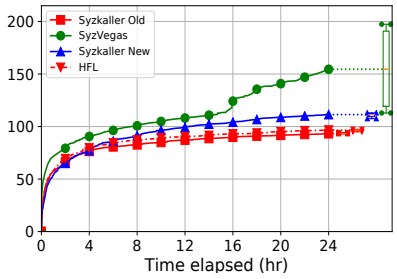


Figure 8: Comparing SYZVEGAS with HFL

effective in improving coverage growth.

Table 2 lists the unique crashes we find. SYZVEGAS discovers 57 (24 unique) crashes, while the default Syzkaller finds 70 (16 unique) crashes. 7crashes correspond to previously unknown bugs; SYZVEGAS detects 6 of these 7 crashes while the default Syzkaller detects 4. Unfortunately, automated reproduction can only reproduce 2 of these bugs; this is a known issue with real-kernel fuzzing due to statefulness, non-determinism, and concurrency.[3] For example, `clear_buffer_attributes` under `drivers/tty/vt` accesses a global array variable `vc_cons`. This array can be modified by many other functions and thus, it is very difficult to reproduce the exact state causing the user-after-free access.

[3]This experiment was performed in June 2020. At the time of May 2021, the two reproducible RCU-stall bugs are fixed and no longer present on the latest Linux kernel (5.13-rc2). The other five non-reproducible bugs cannot be produced on the latest Linux kernel.

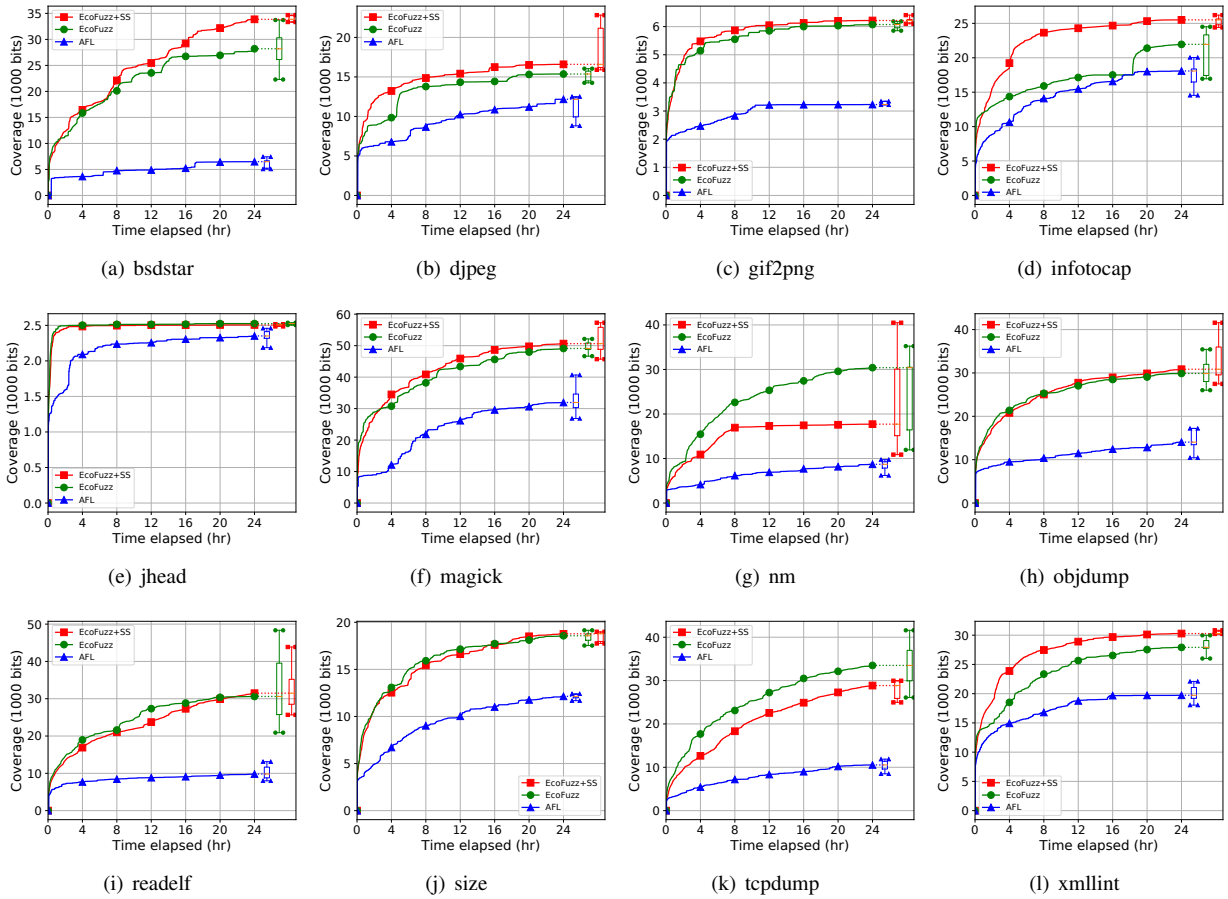
Comparing with HFL. Next, we evaluate how SYZVEGAS compares against a state-of-art Syzkaller-based optimization, viz., HFL [18]. We choose HFL because similar to SYZVEGAS , it is tailored towards kernel fuzzing in general, and is not specific for fuzzing specific drivers (e.g., [29, 30]) or finding specific kind of bugs (e.g., [16]). We run HFL from their repo [3] with the same setup as our other experiments and show the result in Figure 4.2. We notice that HFL is built upon an older Syzkaller from mid-2018, while SyzVegas and the Syzkaller we use in our experiments are from May 2020. Syzkaller has evolved significantly between the two versions, including new supported syscalls (a larger coverage space) and a better seed selection algorithm (better coverage growth rate). Thus, HFL only outperforms the older Syzkaller but not the current Syzkaller. Re-basing HFL to the new Syzkaller requires a tremendous engineering effort as it makes non-trivial modifications to Syzkaller (>8000 lines of code changes). Importantly, SYZVEGAS improves coverage growth by a larger margin over the current Syzkaller, than what HFL brought to the Syzkaller version it was based on.

4.3 Applicability of SYZVEGAS's seed-selection in user-space

User-space fuzzers such as AFL also incorporate seed-selection algorithms. Recent works such as EcoFuzz [34] model AFL's seed selection as an "Adversarial MAB" problem, but do not account for the time taken by a seed in finding the associated new coverage, like with SYZVEGAS. We replace EcoFuzz's seed selection algorithm with that of SYZVEGAS and run the same set of benchmarks for 24 hours, 10 times each. Figure 9 depicts the coverage achieved (measured in the number of bits set). The experiment shows that SYZVEGAS compares favorably with vanilla Ecofuzz. Thanks to accounting for the execution time in SYZVEGAS's reward model, SYZVEGAS outperforms Ecofuzz in 4 out of 12 benchmarks and has the similar efficiency in 6 other benchmarks. We observe, however, that the execution times with AFL-generated inputs are often similar to each other, unlike in the kernel setting. Thus the benefit of accounting for time in the reward model only yields modest benefits in terms of fuzzing coverage growth; in only two cases out of twelve applications considered, SYZVEGAS underperforms EcoFuzz in terms of the coverage.

5 Discussion & Future Work

Choice of Adversarial MAB algorithms. We consider the Adversarial MAB problem to be particularly suitable for SYZVEGAS, and demonstrated that such a stateless simple algorithm can yield considerable benefits. Other advanced reinforcement-learning/machine-learning techniques (e.g., Q-learning [9], PPO [27]) are in principally applicable to task and seed selection. However, we argue that the Adversarial



(a) bsdstar (b) djpeg (c) gif2png (d) infotocap

(e) jhead (f) magick (g) nm (h) objdump

(i) readelf (j) size (k) tcpdump (l) xmllint

Figure 9: Comparing SYZVEGAS with EcoFuzz.

MAB model suits our needs the best for the following reasons:

- Adversarial MAB algorithms we chose are "non-associative" or "non-contextual" [31], i.e., they do not need a definition of state, which is hard to formulate in the context of kernel fuzzing. Thus, they work out-of-the-box on different fuzzing setups (e.g., kernel variants/versions, initial seed corpus, etc). In contrast, many alternate RL algorithms such as Q-learning, require a definition of state.
- The adversarial MAB problem accounts for the possibility of ever-changing rewards for each choice, unlike in standard MAB problems and their solutions (e.g., UCB). Its solution, *Exp3* algorithm, which we based SYZVEGAS on, is shown to quickly detect and adapt to these changes. This makes it apt for kernel fuzzing wherein there exists diminishing potency of seeds and dynamic effectiveness of tasks.
- Adversarial MAB algorithms are computationally efficient to implement, which is critical in maintaining the throughput in fuzzing. The concern of performance is a major reason why it was also chosen in related works such as [34].

A drawback of Adversarial MAB solutions is that they only make locally optimal choices and thus may not yield long-term global optimal strategies (unlike other more complex reinforcement learning algorithms that are associative [31]). For example, SYZVEGAS may perform too many generations in the beginning and could produce some bad seeds and hurt long-term fuzzing. Fortunately, the seed selection component of SYZVEGAS ensures that good seeds are utilized heavily, and bad seeds are de-prioritized eventually.

Another concern of the MAB algorithm is that it only considers the aggregated coverage and disregards relationships between different basic blocks/edges. Thus, arguably it cannot accurately tell which basic block/edge is the most potent in discovering new coverage and reward the corresponding seed. We argue (and find) however, that this effect is diluted as the fuzzer runs for a long time. Thanks to the stochastic nature of MAB algorithm, capable seeds are recognized and exploited eventually. That said, our hope is that a combination of our MAB approach with whitebox methods (considering programs' internal structures) can work in conjunction (left to future work) and further improve fuzzing efficiency.

Delaying Triage. Syzkaller prioritizes triage to add programs into the corpus ASAP; this helps maintain seed programs in the corpus even when the fuzzer VM/device crashes. With SYZVEGAS however, triage tasks are often delayed in favor of generation/mutation, risking heavier loss when the

fuzzer VM/device crashes. However, this is acceptable as the triage work queue is only heavily populated at the beginning of fuzzing when it is much easier for programs to find new coverage. These early programs, however, often only yield "shallow" coverage and are not difficult to reproduce. SYZVEGAS eventually restores triages' absolute priority, thereby eliminating this problem in later stages of fuzzing.

Optimizing for Execution Time. SYZVEGAS optimizes for coverage per unit time and naturally favors seeds with less expensive syscalls initially. In practice, some kernel code might only be reached via time-consuming syscalls (not favored by SYZVEGAS in the beginning). However, as fuzzing goes on, seeds with less expensive syscalls will struggle to find new coverage and SYZVEGAS will naturally switch to seeds with more expensive syscalls that have not been explored much.

Optimizing for Coverage. The ultimate goal of fuzzing is to find vulnerabilities, i.e., find inputs that can crash the target program. Still, SYZVEGAS, alongside most other works (e.g., [8, 26, 34, 35]), adopts a coverage-based reward model instead of a crash-based reward model (e.g. , [33]). Our reasoning is as follows: 1) new coverage is much easier to find than a new crash, and a coverage-based reward model will provide feedback to the fuzzer more frequently; 2) mutating seeds that produce crashes tend to produce the same crashes again, which means that they not necessarily good reward signals; and 3) crashes happen when a certain code path is executed, which is closely related to code coverage (i.e., branches taken).

Other future Work. Real-world kernel-fuzzing frameworks such as syzbot [15] are often executed on top of previous fuzzing runs. In Section 4.2 we demonstrated how SYZVEGAS outperforms the vanilla Syzkaller with an existing seed corpus. However, the status of MAB task/seed selection (i.e., accumulated reward of tasks/seeds) could also be stored for future fuzzing use. We speculate that due to the fast-adapting nature of the adversarial MAB algorithm, the benefit of continuing from an existing MAB state will be limited. We leave the evaluation of this avenue to future work.

In theory, SYZVEGAS can work alongside other fuzzing optimizations using program analysis and/or ML. Since SYZVEGAS only performs task and seed selection, fuzzer optimizations targeting mutation operators (e.g., [9]) should work out-of-the-box with SYZVEGAS. Such optimizations could affect the mutation effectiveness, thus influencing SYZVEGAS's decisions. As for optimizations directly targeting seed selection (e.g., [35]), we could combine SYZVEGAS with MAB algorithms such as Exp4 [7], which can take additional advice vector inputs for guidance.

Reinforcement learning could apply towards tuning other constants or static strategies that are abundant in the Syzkaller implementation, e.g., program size, generation strategy, mutation operator choices, etc. However, the reward assessment models needed can be very different from SYZVEGAS. We believe exploring a unified model to jointly consider all tunable "knobs" in kernel fuzzing is a promising future direction. Another interesting future direction to explore is the applicability of other more advanced RL algorithms to kernel fuzzing.

6 Related Work

MAB techniques in fuzzing. There are attempts to apply MAB techniques to enhance fuzzing performance for seed selection. Woo et al. [33] use the number of crashes as the reward function to select the most "effective" seeds. Patil et al. [25] use the number of interesting test cases as the reward function in a "Contextual Bandit" problem. Yue et.al., propose EcoFuzz [34], which uses an "Adversarial MAB" algorithm to perform seed selection. Our experiments show that SYZVEGAS 's seed selection can be ported to user-space and performs favorably to EcoFuzz. In addition, SYZVEGAS also considers the unique knob of task scheduling between generation, mutation, and triage, which unique to kernel fuzzing (and absent in EcoFuzz). We show that only when the knobs are jointly considered, the MAB model can perform the best.

Other optimization-based fuzzing. In addition to MAB, there are other models proposed to optimize various aspects of fuzzing, including seed selection [8, 26, 35] and mutation strategies [9, 17, 22]. The proposed models and techniques include markov-chain [8], Q-learning [9], Monte Carlo sampling [35], Thompson Sampling [17] and Particle Swarm Optimization [22]. We choose MAB since its simplicity allows us to unify task selection and seed selection in kernel fuzzing. Conceivably, SYZVEGAS can work alongside any algorithm aiming to optimize the mutation operator distribution. We consider the mutation strategy tuning to be another optional knob that can be included in the future.

Kernel fuzzing. Much work has been done to optimize kernel fuzzing. Moonshine [24] tries to improve the quality of the initial seeds in Syzkaller by "distilling" seeds from system call traces of real-world programs. We have shown that SYZVEGAS can work well with Moonshine in Section 4.2. HFL [18] combines fuzzing with symbolic execution. We have shown that SYZVEGAS improves coverage growth by a larger margin than HFL in Section 4.2. kAFL [28] is based on AFL and doesn't have syscall templates like Syzkaller. According to their paper, "the coverage comparison (with Syzkaller) would be highly misleading". FastSyzkaller [21] combines Syzkaller with an N-Gram model, to optimize the test case generation process. Difuze [11] uses static analysis to compose correctly structured inputs in the userspace, to explore kernel drivers. These two works focus on the program generation process while our work focus on scheduling generated/mutated programs. RAZZER [16] focuses on detecting race bugs in the kernel. Agamotto [30] improves virtual machine checkpointing speed which indirectly helps fuzzing speed. Periscope [29] focuses on fuzzing the hardware interface. These goals are orthogonal to those of SYZVEGAS as

we seek to improve coverage growth rate by tuning existing fuzzing knobs more intelligently. Modifying SYZVEGAS's reward functions for other utilities considered by these works is beyond the scope and will be considered in future work.

7 Conclusions

In this paper, motivated by the observations that kernel fuzzing strategies have numerous fixed decisions and hard-coded parameters, we propose SYZVEGAS to dynamically choose the right fuzzing task in conjunction with the right seed, in Syzkaller. Towards this, we choose the specific fuzzing tasks as in a multi-armed-bandit problem, which allows the system to learn the effective strategies and adapt over time, using a novel, yet intuitive reward assessment model to capture benefits and costs. We evaluate SYZVEGAS on Linux kernel version 5.6.13 and several other variants. We show that SYZVEGAS has a low 2.1% performance overhead and makes considerably improves the coverage rate achieved and crashes found, relative to the default Syzkaller. We reported our findings to Google Syzkaller team and have been actively working on upstreaming our changes [4]. At the time of writing, we are testing the implementation of SYZVEGAS with multiple VMs and fuzzer processes and are looking forward to having SYZVEGAS integrated with syzbot soon.

References

[1] Linux kernel selftests. https://www.kernel.org/doc/html/v4.15/dev-tools/kselftest.html.

[2] Open posix tests. http://posixtest.sourceforge.net.

[3] Hfl bitbucket repo, 2020. https://bitbucket.org/anonyk/hfl-release.

[4] pkg/learning, syz-fuzzer: add mab-based seed scheduling, 2020. https://github.com/google/syzkaller/pull/1895.

[5] American fuzzy loop, Online. http://lcamtuf.coredump.cx/afl/.

[6] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 322–331. IEEE, 1995.

[7] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[9] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 116–122. IEEE, 2018.

[10] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

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

[12] LTP developers. Linux testing projects, 2012. https://linux-test-project.github.io.

[13] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[14] Google. Syzkaller. https://github.com/google/syzkaller.

[15] Google. Syzbot, Online. https://syzkaller.appspot.com/upstream.

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

[17] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 37–47. ACM, 2018.

[18] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2020.

[19] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[20] UCR Security Lab. Syzvegas git repo, 2021. https://github.com/seclab-ucr/SyzVegas.

[21] Dan Li and Hua Chen. FastSyzkaller: Improving fuzz efficiency for linux kernel fuzzing. *Journal of Physics: Conference Series*, 1176:022013, mar 2019.

[22] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[23] Gergely Neu. Explore no more: Improved high-probability regret bounds for non-stochastic bandits. In *Advances in Neural Information Processing Systems*, pages 3168–3176, 2015.

[24] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[25] Ketan Patil and Aditya Kanade. Greybox fuzzing as a contextual bandits problem. *CoRR*, abs/1806.03806, 2018.

[26] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, 2014.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[28] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[29] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An effective probing and fuzzing framework for the hardware-OS boundary. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[30] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2541–2557. USENIX Association, August 2020.

[31] Richard S Sutton and Andrew G Barto. *Introduction to Reinforcement Learning*. The MIT Press, 2018.

[32] Pierre Francois Verhulst. Logistic function, 1838. https://en.wikipedia.org/wiki/Logistic_function.

[33] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522. ACM, 2013.

[34] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[35] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In *NDSS*, 2019.

[36] Datong P. Zhou and Claire J. Tomlin. Budget-constrained multi-armed bandits with multiple plays. *CoRR*, abs/1711.05928, 2017.

8 Appendix

8.1 Workflow of Syzkaller

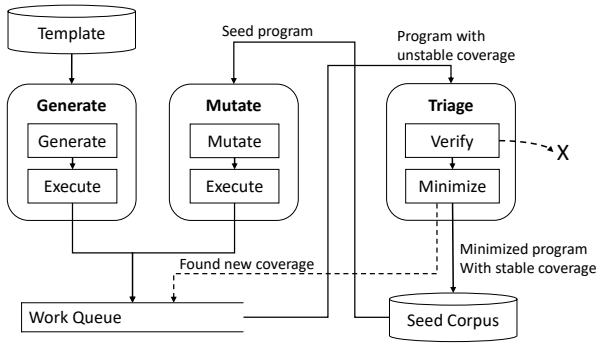


Figure 10: Workflow overview of Syzkaller.

Figure 10 depicts the detailed workflow of Syzkaller. The scheduling of tasks **Generation**, **Mutation** and **Triage** are centered around a LIFO "workqueue". Detailed interaction between tasks and the workqueue is as follows:

- **Generation**. Generation does not rely on other types of tasks as the program is created entirely from templates. If a generated program (i.e., the corresponding syscall sequence) yields new coverage, it is put into the triage *work queue*.
- **Mutation**. Mutated programs producing new coverage are added into the triage work queue. Mutation relies on seed programs created by Triage task.
- **Triage**. Syzkaller fetches a program from the triage work queue. The program was inserted in the queue since it yielded new coverage, but is uncertain if this coverage is reliably reproducible. Thus, Syzkaller re-executes the program thrice and computing coverage that is stable throughout the re-executions, aborting triage if there isn't any. Next, Syzkaller performs a "**Minimization**" of the program, attempting to remove of some system calls and/or the shorten the arguments, while retaining the stable coverage. Finally, Syzkaller puts the minimized program into the seed corpus (where future mutations can be performed). If external seeds are provided by the user (e.g. from a previous run or using Moonshine [24]), they will also need to go through Triage as the kernel and/or Syzkaller version used to generate them might be different.

8.2 *Exp3-IX* Algorithm

Algorithm 3 shows the *Exp3-IX* algorithm. *Exp3-IX* maintains the weight of each of the K arms, each of which is used to proportionally determine their playing probabilities. When an arm is played the algorithm computes the estimated reward based on the probability of this arm and an implicit exploration factor γ. The weight of each arm is exponentially adjusted based on the estimated reward, controlled by the constant growth factor η. Given the number of arms K, the total number of plays T, and an exploration/growth factor $\eta = 2\gamma = \sqrt{\frac{2 ln K}{KT}}$, *Exp3-IX* guarantees a regret bound of:

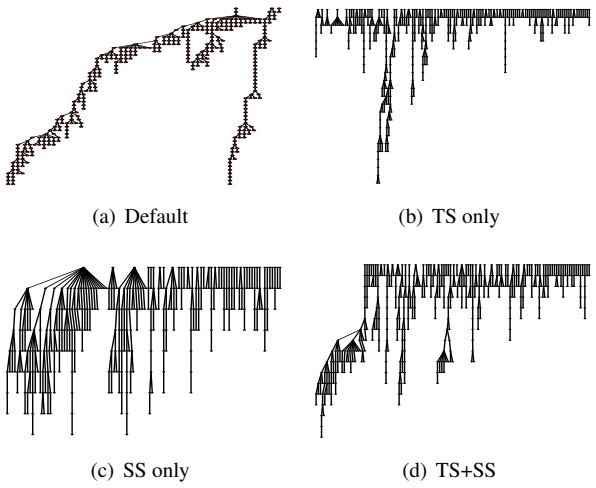
$$G_{max} - E(G_{Exp3-IX}) = \sqrt{2KT\,ln K} + \left(\sqrt{\frac{2KT}{ln K}} + 1\right) ln \frac{2}{\delta} \quad (14)$$

with probability of at least $1-\delta$ for any $0 < \delta < 1$.

Algorithm 3 *Exp3-IX* Algorithm

$w_i \leftarrow 0$. for $i = 1, ..., K$
for all $t = 1, 1, 2$ **do**
  $pr_i(t) \leftarrow \frac{w_i(t)}{\sum_j w_j(t)}$, for $i = 1, ..., K$
  Draw i_t randomly according to $pr_i(t)$
  Receive reward $x_{i_t}(t) \in [0,1]$
  for all $i = 1, ..., K$ **do**
    $\hat{x}_i(t) = \begin{cases} x_{i_t}(t)/(pr(i)+\gamma), & i = i_t \\ 0, & \text{otherwise} \end{cases}$
    $w_i(t+1) = w_i(t) \cdot e^{\eta \hat{x}_i(t)}$
  end for
end for

8.3 Program Evolution During Kernel Fuzzing.



(a) Default (b) TS only

(c) SS only (d) TS+SS

Figure 11: Evolution forests down-sampled to around 500 nodes.

To further understand the impact of MAB task selection and seed selection, we inspect the program evolution in Syzkaller. Everything starts from generated programs, which then go through a series of minimizations and mutations, creating a tree-like structure. We pick one run for each setup and construct the program evolution forest, down-sample it to around 500 nodes and show it in Figure 11.

We observe that different strategies have very different approaches to program evolution. Vanilla Syzkaller (Figure 11(a)), as discussed earlier in Figure 2.2, favors a depth-first approach thanks to the LIFO workqueue and triage-smash-first policy. It performs very few generations (13 trees before sampling) and is quite biased when it comes to exploration (a.k.a. mutation). With only task selection (Figure 11(b)), SYZVEGAS performs the most generation tasks and creates the largest number of trees (789 before sampling), but spend less time exploring them while suffering from the same biased exploration of the default Syzkaller. With only seed selection (Figure 11(c)), a reasonable number of trees (202 before sampling) are created and trees are more balanced. However, it is clear that the tree created by the very first generation is explored much more in-depth than the latter trees. Finally, with both task and seed selection, SYZVEGAS combines both the large tree numbers because of scheduling, and the exploration balance from seed selection. Specifically, with more generations at the beginning, SYZVEGAS is able to turn more of these generations (347 before sampling) into trees.