Make out like a (Multi-Armed) Bandit: Improving the Odds of Fuzzer Seed Scheduling with T-Scheduler

Simon Luo
The University of New South Wales
Australia
simon.luo@unsw.edu.au

Adrian Herrera
Australian National University
Australia

Paul Quirk
Michael Chase
Defence Science & Technology Group
Australia

Damith C. Ranasinghe
University of Adelaide
Australia

Salil S. Kanhere
The University of New South Wales
Australia

ABSTRACT

Fuzzing is an industry-standard software testing technique that uncovers bugs in a target program by executing it with mutated inputs. Over the lifecycle of a fuzzing campaign, the fuzzer accumulates inputs inducing new and interesting target behaviors, drawing from these inputs for further mutation and generation of new inputs. This rapidly results in a large pool of inputs to select from, making it challenging to quickly determine the "most promising" input for mutation. Reinforcement learning (RL) provides a natural solution to this *seed scheduling* problem—*a fuzzer can dynamically adapt its selection strategy by learning from past results*. However, existing RL approaches are (a) computationally expensive (reducing fuzzer throughput), and/or (b) require hyperparameter tuning (reducing generality across targets and input types). To this end, we propose T-Scheduler, a seed scheduler built upon multi-armed bandit theory to automatically adapt to the target. Notably, our formulation does not require the user to select or tune hyperparameters and is therefore easily generalizable across different targets. We evaluate T-Scheduler over 35 CPU-yr fuzzing effort, comparing it to 11 state-of-the-art schedulers. Our results show that T-Scheduler improves on these 11 schedulers on both bug-finding and coverage-expansion abilities.

CCS CONCEPTS

• **Security and privacy → Software and application security**;
• **Computing methodologies → Machine learning**.

KEYWORDS

Fuzzing, Software Testing, Thompson Sampling, Reinforcement Learning, Multi-Armed Bandits

ACM Reference Format:
Simon Luo, Adrian Herrera, Paul Quirk, Michael Chase, Damith C. Ranasinghe, and Salil S. Kanhere. 2024. Make out like a (Multi-Armed) Bandit: Improving the Odds of Fuzzer Seed Scheduling with T-Scheduler. In *Proceedings of The 19th ACM ASIA Conference on Computer and Communications Security (AsiaCCS '24).* ACM, New York, NY, USA, 17 pages. https://doi.org/XXXXXXX.XXXXXXX

AsiaCCS '24, July 01–05, 2024, Singapore
© 2024 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

> "*Make out like a bandit*". Idiom. To make a large profit.
>
> Merriam-Webster Dictionary

Fuzzing is a software testing technique for automatically finding bugs and vulnerabilities in a target program. Fuzzers find bugs by mutating inputs to induce new behavior in the target. Intuitively, mutated inputs are more likely to exercise corner cases in the target's behaviors, leading to bugs. While most of these mutations do not lead to anything interesting, there remains a chance that the mutated input induces new and interesting target behaviors.

Intelligently selecting which inputs to mutate is critical for maximizing fuzzer effectiveness; inputs more likely to uncover new behaviors should be prioritized for mutation. This prioritization of inputs is known as *seed scheduling*[1] [27, 40]. Seed schedulers typically use heuristics to determine an input's position in the fuzzer's queue. In a coverage-guided greybox fuzzer—the most common type of fuzzer—seed scheduling can be driven by a combination of: (i) code coverage (inputs leading to new code uncover new behaviors); (ii) input size (smaller inputs are faster to mutate); (iii) execution time (inputs with shorter execution time mean more fuzzer iterations); (iv) the number of times the input has been previously selected (avoiding local optima); and (v) similarity with other inputs (improving diversity). However, seed scheduling is challenging because of a combination of the (a) large number of inputs generated via mutation (and thus requiring prioritization), (b) large search space of the target, and (c) computational requirements (e.g., CPU time, memory, and storage).

Machine learning (ML)—in particular, reinforcement learning (RL)—is commonly applied to solve challenges in fuzzing [9, 11, 12, 16, 20, 25, 31, 32, 39, 40, 42, 50]. Notably, RL has been used to adaptively learn seed scheduling strategies more likely to lead to increased code coverage. In turn, this increases the likelihood of uncovering new bugs (after all, one cannot find bugs in code that is never executed). However, integrating RL into fuzzing introduces two challenges: *performance tradeoffs* and *hyperparameter tuning*.

[1] We use the term "seed scheduling", rather than "seed selection", to disambiguate it from the (offline) process of selecting an initial set of inputs to bootstrap the fuzzer.

Performance tradeoffs. A fuzzer's iteration rate is the number of inputs the fuzzer executes per unit of time; the faster the fuzzer's iteration rate, the quicker the fuzzer can discover new and interesting behaviors. However, balancing performance and "cleverness" in selecting the best input to mutate is difficult. Moreover, RL algorithms require computational resources to train and evaluate, impacting a fuzzer's iteration rate. Naïvely introducing RL into a fuzzer (notably, for seed scheduling) can increase run-time overhead without any performance improvement. For example, we found AFL-Hier [40] (a fuzzer using RL for seed scheduling) introduced >2× overhead over AFL++'s [15] heuristic-based scheduler without any improvement in fuzzing outcomes.

Hyperparameter tuning. RL algorithms use hyperparameters to configure their learning process. The number of hyperparameters depends on the RL algorithm used. For example, AFLFast [7], EcoFuzz [46], AFL-Hier [40], and MobFuzz [48] (fuzzers using RL in their seed schedulers) each have two hyperparameters. Hyperparameters must be set before learning (and hence fuzzing) begins. However, empirically selecting optimal hyperparameter values is time-consuming and difficult to generalize; optimal values are likely to vary across targets and input formats. Suboptimal hyperparameter values reduce fuzzing performance.

We propose an RL approach that addresses these challenges and improves fuzzing outcomes. Our approach models seed scheduling as a *multi-armed bandit* (MAB) problem, which we solve using *Thompson sampling*. Thompson sampling allows us to adaptively and efficiently model the probability of the fuzzer uncovering new and interesting behaviors. In doing so, the fuzzer can make more intelligent seed scheduling decisions with (a) no hyperparameters to tune, (b) theoretical optimality guarantees [2], and (c) constant-time overheads. Our approach also uses a self-balancing mechanism to prioritize inputs covering rare paths and newly-discovered code.

We implement our RL-based seed scheduler, T-Scheduler, in AFL++ (the current state-of-the-art coverage-guided greybox fuzzer) and evaluate it on 35 programs across two widely-used fuzzer benchmarks (Magma [17] and FuzzBench [28]). Our evaluation shows that T-Scheduler consistently improves on 11 state-of-the-art seed schedulers on 26 programs. We contribute:

- **A theoretical formulation of the seed scheduling problem.** We formulate seed scheduling as a MAB problem. Our formulation allows the fuzzer to prioritize inputs corresponding to newly-discovered target behaviors, based on learning the historical success of past seed inputs (Sections 3.1 to 3.5).
- **An RL-based seed scheduler with no hyperparameters.** We design and implement T-Scheduler, based on solving our MAB problem using Thompson sampling. Using Thompson sampling means that we benefit from the inherent theoretical guarantee that model errors grow sublinearly—important for facilitating long fuzzing campaigns. Moreover, our implementation has no hyperparameters, making it more generalizable to different targets and input formats. We integrate our implementation into AFL++ (Section 3.5).
- **An effective and generalizable seed scheduler.** We evaluate T-Scheduler by fuzzing real-world programs (>35 CPU-yr). Our approach outperforms current state-of-the-art schedulers across both bug-finding and coverage-expansion metrics (Section 4).
- **Analysis of seed scheduler costs.** Seed schedulers must carefully balance overhead costs and precision. To this end, we empirically analyze the cost of existing schedulers to understand their impact on fuzzing outcomes (Section 4.4).

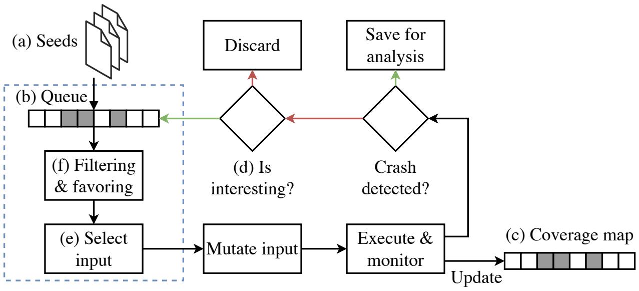


Figure 1: Overview of Greybox Fuzzing.

We release our implementation and results at https://github.com/asiaccs2024-t-scheduler.

2 BACKGROUND

2.1 Fuzzing

Fuzz testing ("fuzzing") is a dynamic analysis for uncovering bugs in software. In a security context, fuzzers have been wildly successful at discovering tens of thousands of security-critical vulnerabilities in widely-used code [10]. Bugs are found by (rapidly) subjecting a target program to automatically-generated inputs. The fuzzer generates inputs to expose and explore corner cases in the target not considered by the developer. Intuitively, it is in these corner cases where bugs are most likely to lie.

Figure 1 illustrates the fuzzing process. A fuzzing campaign begins with curating a corpus of "well-formed" inputs. These inputs are commonly exemplar input data accepted by the target (Fig. 1(a)) [18, 30]. At run time, the fuzzer maintains a queue—initially populated from this curated corpus—from which an input is selected to fuzz (Fig. 1(b)). The selected input is mutated and fed into the target to expose corner cases in program behavior. How does the fuzzer select these inputs?

Greybox fuzzers use lightweight instrumentation to track code executed (or "covered") by the target (in contrast, *blackbox* fuzzers have no internal view of the target). The fuzzer records code coverage in a *coverage map*, which tracks the number of times a particular coverage element—typically, an edge in the target's control-flow graph (CFG)—is executed (Fig. 1(c)). By tracking code coverage, the fuzzer can determine if the mutated input triggers new target behaviors. Inputs triggering new behaviors are saved back into the queue; otherwise, the input is discarded (Fig. 1(d)). This guides the fuzzer to prioritize inputs leading to new program behaviors (as captured by the instrumentation). Selecting and prioritizing inputs for mutation is handled by the *seed scheduler* (Fig. 1(e)).

2.2 Seed Scheduling

An efficient seed scheduler (the blue box in Fig. 1) must satisfy two competing constraints. First, the seed scheduler must *select the most-promising input for mutation*; i.e., the input most likely to cover new program behaviors. Second, it must make this selection from a (potentially) large queue with *minimal run-time overhead*. AFL-based [47] fuzzers solve this problem by bounding the input queue to a fixed size via an *input filtering and favoring* phase [43].

Input filtering and favoring. The fuzzer only retains inputs that induce new and interesting behaviors in the target; e.g., an input that covers a new element in the coverage map. These inputs are made available for future mutations, potentially uncovering more new behaviors. AFL "favors" an input if it is the fastest and smallest input for any of the coverage map elements [7] (Fig. 1(f)). Thus, tracking favored inputs gives the fuzzer a minimal set of inputs (that are both small and fast) covering all of the elements seen in the coverage map so far (approximating a *weighted minimum set cover*, with size and speed as weights [15]). Maintaining a set of favored inputs implicitly reduces the seed scheduling problem from an unbounded number of inputs (the union of the initial seed corpus and the inputs generated so far) to a bounded number of inputs: the number of favored inputs; i.e., the coverage map's size.

After filtering and favoring, the scheduler has a bounded number of inputs from which to select an input to fuzz. From this, the scheduler selects the "best" input to mutate. For example, AFL selects an input based on a score calculated using a set of heuristics. These heuristics calculate a performance score based on an input's: (i) coverage; (ii) execution time (faster inputs are preferred); (iii) "depth" (i.e., the number of inputs mutated to reach the given input); and (iv) the fuzzer's run time (newer inputs are prioritized). A *power schedule* [7, 15] then distributes fuzzing time across inputs by scaling the performance score based on the number of times the input has been selected, biasing fuzzing time to less-fuzzed inputs. Higher energy means the fuzzer spends more time mutating the corresponding (favored) input.

Notably, AFL's heuristics are based on intuition and experimentation. Other fuzzers (e.g., EcoFuzz [46], AFL-Hier [40], MobFuzz [48], and `K-Scheduler` [35]) also rely on hyperparameters that must be tuned per target to achieve optimal results. In contrast, our approach (described in Section 3) replaces these heuristics with an RL algorithm that uses run-time statistics to dynamically learn and adapt a seed schedule. Moreover, our approach has no hyperparameters to tune, leading to more efficient and informed input selection.

2.3 Reinforcement Learning

Reinforcement learning (RL) is an ML paradigm that trains an agent by observing changes in state and rewarding the selected actions [36]. The agent aims to select the best action to maximize the cumulative reward. However, the expected reward for each action is often unknown and must be learned dynamically via experimentation. This experimentation leads to a trade-off between *exploiting* what is already known and *exploring* territory.

An RL algorithm is typically defined in terms of *states*, *actions*, and *rewards*. The state is a set of variables describing the environment. Based on the current state, the agent (a) selects an action to perform, and (b) receives feedback on its selection in the form of a reward. The agent's objective is thus to maximize the cumulative reward over a given time.

These concepts apply naturally to fuzzing. In particular, the fuzzer's seed scheduler must select an input (to mutate) from a pool of constantly-changing possibilities. Ideally, this selection maximizes the discovery of new code, or (ideally) a bug. *The seed scheduler must balance exploring the input queue and exploiting the input uncovering the most code.* This requires a careful trade-off between making intelligent input prioritization decisions and maintaining the fuzzer's iteration rate. We satisfy this trade-off by formulating seed scheduling as a *multi-armed bandit*.

2.3.1 Multi-Armed Bandit. The multi-armed bandit (MAB) is a well-explored RL problem focusing on the trade-off between exploration and exploitation [24]. Given a state, the agent selects an action $a_k \in \mathcal{A}$ at each time step $t \in [1, \ldots, T]$. The agent's goal is to maximize the cumulative reward (by performing a sequence of actions) over T.

The classic MAB involves K slot machines ("bandits"), where each $k \in [1, \ldots, K]$ has an unknown probability θ_k of paying out when played. At each time step t, the player (i.e., agent) selects a slot machine to play. Once played, the player is either rewarded with a payout (with probability θ_k) or receives nothing (with probability $1 - \theta_k$). Naturally, any rational player would focus on the bandit paying out the most (thus achieving their goal of maximizing the cumulative reward). Unfortunately, *this information is unknown to the player*. Consequently, the player must trade-off between *exploiting* the bandit with the (current) highest expected payout and *exploring* different bandits to learn more about the probability θ_k (in the hope of finding a higher payout). What is the best strategy for selecting between exploration and exploitation?

Thompson sampling [37] is a popular approach for addressing this exploration/exploitation trade-off. This popularity is due to simplicity, fast execution time, and optimality guarantees (ensuring errors grow sublinearly over time [2]). Consequently, we adopt Thompson sampling in T-Scheduler, our RL-based seed scheduler.

3 APPROACH

Our seed scheduler, T-Scheduler, formulates greybox fuzzing as a Beta-Bernoulli bandit, which we solve with Thompson sampling. We first provide a high-level description of the T-Scheduler algorithm (Section 3.2) and a motivating example (Section 3.2.1), followed by a mathematical formulation of our Beta-Bernoulli bandit model (Sections 3.3 to 3.5).

3.1 Notation and Definitions

A fuzzer measures its progress fuzzing target $\mathcal{P}$ in a coverage map $C(\mathcal{P}, \iota) \in \mathbb{N}^K$ of size K. Typically, each *feature* $x \in C(\mathcal{P}, \iota)$ records the number of times a particular edge in the target's control-flow graph (CFG)[2] is executed by an input $\iota \in \mathcal{I}$, where $\mathcal{I}$ is the set of inputs representing the union of the initial corpus and the set of inputs generated by the fuzzer. We refer to this count as a *hit count* (given by the function `hit_count`).

Feature rareness prioritizes features covered less by $\mathcal{I}$. We adopt the definition by Wang et al. [40], where feature rareness is the

[2] Some fuzzers eschew edge coverage for other coverage metrics (e.g., context-sensitive edge or data flow). Our approach is agnostic to the underlying coverage metric.

inverse of x's hit count. Feature hit counts and rareness are defined for each feature in the coverage map and the inputs generated so far. These definitions enable a seed scheduler that prioritizes both newly-discovered and hard-to-reach code.

3.2 The T-Scheduler Algorithm

Algorithm 1: T-Scheduler.

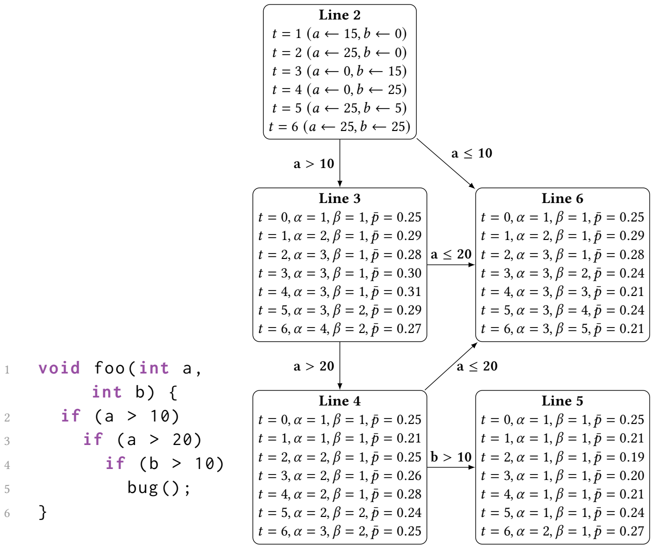
```
1  $\boldsymbol{\alpha} \leftarrow \{1 \,|\, k \in [1, \dots, K]\}$
2  $\boldsymbol{\beta} \leftarrow \{1 \,|\, k \in [1, \dots, K]\}$
3  Function UpdatePosterior($C(\mathcal{P}, \iota)$, $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$)
4      for $k, x \in C(\mathcal{P}, \iota)$ do
5          if $x \neq 0$ then
6              if IsInteresting($\iota$) then
7                  $\alpha_k \leftarrow \alpha_k + 1$
8              else
9                  $\beta_k \leftarrow \beta_k + 1$
10     return $\boldsymbol{\alpha}, \boldsymbol{\beta}$
11 Function SelectInput($\boldsymbol{\alpha}, \boldsymbol{\beta}$)
12     for $k \leftarrow [1, \dots, K]$ do
13         $\hat{\theta}_k \sim \mathrm{Beta}(\alpha_k, \beta_k)$
14         $\psi_k \sim \mathrm{Beta}(\alpha_k + \beta_k, \alpha_k^2)$
15     $a_t \leftarrow \arg\max[\psi_1 \hat{\theta}_1, \dots, \psi_K \hat{\theta}_K]$
16     $\mathcal{I}^{(t+1)} \leftarrow$ FavoredInputs($a_t$)
17     return $\mathcal{I}^{(t+1)}$
```

We present the T-Scheduler algorithm in Algorithm 1. It consists of two functions: UpdatePosterior and SelectInput.

The UpdatePosterior function—called each time an input ι is executed—uses two K-length vectors—$\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ (Lines 7 and 9)—to store the number of times a coverage map feature is hit or missed, respectively. Each element α_k and β_k (where $k \in [1, \dots, K]$) represents the number of times ι hits or misses $x \in C(\mathcal{P}, \iota)$. These vectors are used to compute a probability distribution for each coverage map feature, modeling the probability of an input inducing new behaviors in $\mathcal{P}$.

The SelectInput function is called when the queue has been exhausted. It samples $\boldsymbol{\theta} = [\theta_1, \dots, \theta_K]$ from K Beta distributions (Line 13) to determine the "best" input to fuzz (see Section 3.3). However, using only $\boldsymbol{\theta}$ leads the fuzzer to repeatedly select the same inputs, because it implicitly penalizes rarely-covered features in the coverage map. Thus, we introduce a *correction factor* $\boldsymbol{\psi} = [\psi_1, \dots, \psi_K]$ (Line 14) based on feature rareness to penalize frequently-covered coverage map features (see Section 3.4).

Finally, the next input to fuzz is chosen from FavoredInputs using $\hat{\theta}_k$ and ψ_k (Lines 15 to 16; see Section 3.5). FavoredInputs stores a single "best" input corresponding to each feature in the coverage map. Per Section 2.2, the fuzzer determines this input by a combination of execution time, input size, and the number of times the input has been fuzzed.



```
1  void foo(int a,
       int b) {
2    if (a > 10)
3      if (a > 20)
4        if (b > 10)
5          bug();
6  }
```

Line 2
$t = 1$ ($a \leftarrow 15, b \leftarrow 0$)
$t = 2$ ($a \leftarrow 25, b \leftarrow 0$)
$t = 3$ ($a \leftarrow 0, b \leftarrow 15$)
$t = 4$ ($a \leftarrow 0, b \leftarrow 25$)
$t = 5$ ($a \leftarrow 25, b \leftarrow 5$)
$t = 6$ ($a \leftarrow 25, b \leftarrow 25$)
a > 10
a ≤ 10
Line 3
$t = 0, \alpha = 1, \beta = 1, \bar{p} = 0.25$
$t = 1, \alpha = 2, \beta = 1, \bar{p} = 0.29$
$t = 2, \alpha = 3, \beta = 1, \bar{p} = 0.28$
$t = 3, \alpha = 3, \beta = 1, \bar{p} = 0.30$
$t = 4, \alpha = 3, \beta = 1, \bar{p} = 0.31$
$t = 5, \alpha = 3, \beta = 2, \bar{p} = 0.29$
$t = 6, \alpha = 4, \beta = 2, \bar{p} = 0.27$
a ≤ 20
Line 6
$t = 0, \alpha = 1, \beta = 1, \bar{p} = 0.25$
$t = 1, \alpha = 2, \beta = 1, \bar{p} = 0.29$
$t = 2, \alpha = 3, \beta = 1, \bar{p} = 0.28$
$t = 3, \alpha = 3, \beta = 2, \bar{p} = 0.24$
$t = 4, \alpha = 3, \beta = 3, \bar{p} = 0.21$
$t = 5, \alpha = 3, \beta = 4, \bar{p} = 0.24$
$t = 6, \alpha = 3, \beta = 5, \bar{p} = 0.21$
a > 20
a ≤ 20
Line 4
$t = 0, \alpha = 1, \beta = 1, \bar{p} = 0.25$
$t = 1, \alpha = 1, \beta = 1, \bar{p} = 0.21$
$t = 2, \alpha = 2, \beta = 1, \bar{p} = 0.25$
$t = 3, \alpha = 2, \beta = 1, \bar{p} = 0.26$
$t = 4, \alpha = 2, \beta = 1, \bar{p} = 0.28$
$t = 5, \alpha = 2, \beta = 2, \bar{p} = 0.24$
$t = 6, \alpha = 3, \beta = 2, \bar{p} = 0.25$
b > 10
Line 5
$t = 0, \alpha = 1, \beta = 1, \bar{p} = 0.25$
$t = 1, \alpha = 1, \beta = 1, \bar{p} = 0.21$
$t = 2, \alpha = 1, \beta = 1, \bar{p} = 0.19$
$t = 3, \alpha = 1, \beta = 1, \bar{p} = 0.20$
$t = 4, \alpha = 1, \beta = 1, \bar{p} = 0.21$
$t = 5, \alpha = 1, \beta = 1, \bar{p} = 0.24$
$t = 6, \alpha = 2, \beta = 1, \bar{p} = 0.27$

(a) The program. The parameters a and b are derived from user input.

(b) CFG showing the changing parameters α_k and β_k at each time step t. The mean probability for each time step is shown as $\bar{p}$.

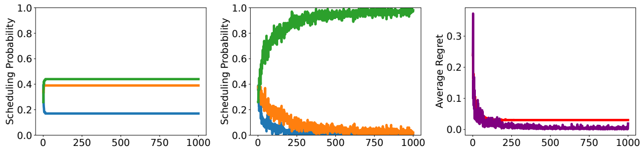
Figure 2: An example showing how T-Scheduler updates its parameters after each input is executed. Six different values for a and b (corresponding to six different inputs) are shown in the root node. The parameter α is incremented when the input discovers new program behavior, otherwise β is incremented. The probability $\bar{p}_k$ is the normalized mean of the Beta distribution in Line 13 in Algorithm 1 using Eq. (1).

3.2.1 Motivating Example. We use the example in Fig. 2 to illustrate T-Scheduler's approach. T-Scheduler is an *adaptive* scheduler, meaning the probability of selecting a given input (for mutation) changes as the scheduler receives coverage feedback from the fuzzer. In this example, coverage feedback is used to update α (Line 7 in Algorithm 1) and β (Line 9), storing the positive and negative feedback of past scheduling decisions, respectively. The scheduler uses the updated parameters α and β to assign a probability for selecting an input (Line 13). We show the mean of this probability distribution

$$\bar{p}_k = \frac{\mathbb{E}[\mathrm{Beta}(\alpha_k, \beta_k)]}{\sum_{k=1}^{K} \mathbb{E}[\mathrm{Beta}(\alpha_k, \beta_k)]}, \tag{1}$$

because of the difficulty visualizing $\hat{\theta}_k$ in Fig. 2. Importantly, Algorithm 1 does not need to compute $\bar{p}_k$ and is only used to visualize how the mean of the probability distribution changes over time.

Figure 2 uses six example inputs to show how the parameters are updated. Inputs discovering new program behaviors are stored in the corresponding node and $\bar{p}$ is the probability of the input being selected by the scheduler. This assumes that prior inputs that discovered new program behaviors are more likely to discover new program behaviors. At $t = 1$, the input covers lines 3 and 6 (Section 3.2.1) for the first time, so α is incremented and $\bar{p}$ increases. At $t = 2$, the input covers lines 3, 4, and 6. Line 4 is covered for the first time and α is incremented for lines 3, 4, and 6. Notably, $\bar{p}$ in line 4 increases but $\bar{p}$ decreases at lines 3 and 6 because we favor rarer paths. The input covers line 6 at $t = 3$ and $t = 4$. Line 6 has already been covered, so β is incremented and $\bar{p}$ decreases (because we penalize inputs that do not uncover new program behaviors). Similarly, at $t = 5$ the input covers lines 3, 4, and 6, which have also been covered by previous inputs. Thus, $\bar{p}$ decreases at these lines



(a) Greedy. **(b) T-Scheduler.** **(c) Regret.**

Figure 3: An example scheduler demonstrating the probability of an input being selected over 1,000 iterations (the x-axis is iteration count). The regret quantifies the error by taking the difference between the optimal decision and the input selected. The inputs ι_1, ι_2, and $\iota 3$ have latent probabilities of 0.7, 0.8, and 0.9, respectively. The greedy algorithm has a constant regret, while the T-Scheduler algorithm's regret approaches zero over time. Highlight colors reflect those in the plots.

while $\bar{p}$ increases for lines 2 and 6. Finally, line 5 is covered for the first time at $t = 6$. This results in α increasing at lines 3, 4, and 5 and $\bar{p}$ changing disproportionally to favor rarer paths.

The scheduler selects an input to executed based on the probability that mutating the input will lead to new program behaviors. This probability is unknown. Thus, inputs with a higher probability for discovering new program behaviors must be estimated and prioritized by the scheduler. Figure 3 illustrates the impact different scheduling algorithms have on these estimates. Here, we assume that a scheduler has three inputs to select from—ι_1, ι_2, and ι_3—with probabilities 0.7, 0.8, and 0.9 that a mutation will discover new program behavior (which is unknown and needs to be estimated by the scheduler in a fuzzing campaign), respectively to demonstrate the behavior of the algorithm.[3] Here, we can see input ι_3 should be selected because it has the highest probability to discover new program behavior. But the scheduler does not know these values and is required to estimate them to assign a scheduling probability to each input. Figures 3a and 3b show the convergence of $\bar{p}$ for two scheduling approaches—a greedy algorithm and T-Scheduler—when selecting inputs using the scheduling probabilities ($\bar{p}$ or $\hat{\theta}$). T-Scheduler allows for sub-optimal decisions at a given time by sampling from a distribution (Line 13 in Algorithm 1) meaning the input with the highest $\bar{p}$ is not always selected. These sub-optimal decisions allow the model to "explore" early on in the fuzzing campaign and "exploit" inputs when it has more information. The greedy approach shown in Fig. 3a always tries to select the highest scheduling probability which does not lead to the optimal solution. Figure 3c illustrates the regret of T-Scheduler converging to zero over time leading to the optimal solution while a constant regret remains for the greedy approach. In this example, this can only be achieved if the scheduler always picks ι_3. Current state-of-the-art schedulers (e.g., AFLFast [7], Entropic [5], and TortoiseFuzz [43]) use a greedy approach to make decisions. T-Scheduler makes more optimal decisions (compared to greedy algorithms), particularly in long fuzzing campaigns.

With the T-Scheduler algorithm presented, we now turn our attention to the probabilistic modeling that underpins our approach.

3.3 Adapting the Beta-Bernoulli Bandit

Thompson sampling frames exploration and exploitation as a Bayesian posterior estimation, choosing an action that maximizes a reward based on a randomly-drawn prior belief (Section 2.3.1). We assume that information derived from previous inputs can be used to improve input scheduling in the future. The reward function provides feedback to the fuzzer after each input ι is executed. In particular, the fuzzer is (positively) rewarded for discovering new target behaviors, and penalized otherwise.

We begin our formulation with a K-armed bandit; i.e., there are K actions for the fuzzer to choose from. Here, K is the size of $C(\mathcal{P}, \iota)$, and each entry of the coverage map corresponds to a favored input in a many-to-one relationship (Section 2.2). We define a fuzzing campaign with respect to a time step $t \in [1, \ldots, T]$, where T is the length of the campaign (i.e., the number of executed inputs). Each input $\iota^{(t)}$ at time step t will produce a coverage map $C(\mathcal{P}, \iota)^{(t)}$. After performing action $k \in [1, \ldots K]$ the fuzzer is rewarded by:

$$r_k^{(t)} := \begin{cases} 1, & \text{if } x_k^{(t)} \neq 0 \text{ and } \iota^{(t)} \text{ is interesting,} \\ 0, & \text{if } x_k^{(t)} \neq 0, \end{cases} \tag{2}$$

where $x_k^{(t)} \in C(\mathcal{P}, \iota)^{(t)}$ is the coverage map feature at index k. For each time step t the reward is represented as the vector $\mathbf{r}^{(t)} = [r_1^{(t)}, \ldots, r_K^{(t)}]$. The fuzzer is rewarded for inducing interesting behaviors in the target (e.g., uncovering new code). Intuitively, this ensures the scheduler selects inputs that are more likely to induce new behaviors in $\mathcal{P}$.

3.3.1 *Estimating Probabilities and Rewarding the Fuzzer.*

The probability that the fuzzer generates an input (by mutating the current input) inducing new behaviors is $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_K]$. Importantly, $\boldsymbol{\theta}$ is unknown and must be estimated over time through experimentation. Per Section 2.3, learning $\boldsymbol{\theta}$ requires a careful balance between exploration and exploitation to maximize the cumulative reward over T. The estimated $\boldsymbol{\theta}$ guides the seed scheduler to select the next (best) input to fuzz.

We design the reward function in Eq. (2) such that θ_k (the *posterior distribution*) can be estimated using the Beta-Bernoulli bandit. In this setting, θ_k is the probability that the input induces new behaviors in $\mathcal{P}$, and is estimated by a Bernoulli distribution with observations $r_k^{(t)}$ (the *likelihood*) and a Beta distribution over $\hat{\theta}_k^{(t)}$ (the *prior distribution*); i.e.,

$$r_k^{(t)} \sim p(r_k^{(t)} | \hat{\theta}_k^{(t)}) = \text{Bernoulli}(\hat{\theta}_k^{(t)}) \tag{3}$$

and

$$\hat{\theta}_k^{(t)} \sim p(\hat{\theta}_k^{(t)}; \alpha_k, \beta_k^{(t)}) = \text{Beta}(\alpha_k^{(t)}, \beta_k^{(t)}). \tag{4}$$

The prior distribution is modeled as a Beta distribution with parameters $\boldsymbol{\alpha}^{(t)} = [\alpha_1^{(t)}, \ldots, \alpha_K^{(t)}]$ and $\boldsymbol{\beta}^{(t)} = [\beta_1^{(t)}, \ldots, \beta_K^{(t)}]$ (Lines 7 and 9, Algorithm 1), while the parameters $\alpha_k^{(0)}$ and $\beta_k^{(0)}$ are initialized to 1 at $t = 0$ (Lines 1 and 2, Algorithm 1). The parameter $\boldsymbol{\alpha} - 1$ represents the number of times each $x \in C(\mathcal{P}, \iota)$ has been covered (for all $\mathcal{I}$ executed so far) by an interesting input,

[3] In a fuzzing campaign these probabilities change as new program behaviors are discovered.

while $\boldsymbol{\beta} - 1$ represents the number of times each $x \in C(\mathcal{P}, \iota)$ was covered by input that was *not* considered interesting.

Using Bayes rule, the posterior distribution $p(\hat{\theta}|r_k^{(t)})$ is

$$\begin{aligned} p(\hat{\theta}_k^{(t)}|r_k^{(t)}) &= \frac{p(r_k^{(t)}|\hat{\theta}_k^{(t)})p(\hat{\theta}_k^{(t)};\alpha_k^{(t)},\beta_k^{(t)})}{p(r_k^{(t)})} \\ &\propto p(r_k^{(t)}|\hat{\theta}_k^{(t)})p(\hat{\theta}_k^{(t)};\alpha_k^{(t)},\beta_k^{(t)}). \end{aligned} \tag{5}$$

The prior distribution $p(\hat{\theta}_k^{(t)};\alpha_k^{(t)},\beta_k^{(t)})$ represents the distribution at the previous time step; i.e.,

$$p(\hat{\theta}_k^{(t)};\alpha_k^{(t)},\beta_k^{(t)}) := p(\hat{\theta}_k^{(t-1)}|r_k^{(t-1)}), \tag{6}$$

while the likelihood $p(r_k^{(t)}|\hat{\theta}_k^{(t)})$ takes into account the coverage achieved at the current time step. Equation (6) provides a mechanism for the fuzzer to update its model to ensure it selects the "best" seed. The recursive structure defined in Eqs. (5) and (6) means the model is dependent on previously selected inputs. The fuzzer continuously updates the model by receiving feedback from the reward function (Eq. (2)) at each timestep. This approximates the posterior distribution $p(\hat{\theta}_k^{(t)}|r_k^{(t)})$, which is now the probability that a fuzzer-generated input will cover this feature in the coverage map.

3.3.2 Improving Performance. Directly sampling from Eq. (5) to compute $\boldsymbol{\theta}$ is computationally expensive. However, using the Beta distribution (which is a *conjugate prior* to the Bernoulli distribution) avoids expensive numerical computations (that are typical in Bayesian inference), leading to simpler updates after each time step. In particular, solving Eq. (5) with Eqs. (3) and (4) gives the following update rule for the posterior distribution:

$$(\alpha_k^{(t)},\beta_k^{(t)}) = (\alpha_k^{(t-1)},\beta_k^{(t-1)}) + (r_k^{(t)}, 1 - r_k^{(t)}). \tag{7}$$

This is the approach taken in `UpdatePosterior` (Algorithm 1): the parameters $\alpha_k^{(t)}$ and $\beta_k^{(t)}$ are updated incrementally (at each time step) with each observation of success and failure, respectively. The update rule has been illustrated in Fig. 2, where α_k is incremented for the path of the input if the input discovered new program behavior, otherwise β_k is incremented. For example, at $t = 1$, input covered lines 3 and 6 for the first time so α was incremented for both of these nodes, and at $t = 3$, the input covered Line 6 but did not discover new code coverage, so *beta* was incremented for Line 6.

Following these updates, Eq. (5) is sampled by drawing from a Beta distribution with parameters α_k and β_k,

$$\hat{\theta}_k^{(t)} \sim p(\theta = \hat{\theta}_k^{(t)}|\tilde{r}_k^{(t+1)} = 1) = \mathrm{Beta}(\alpha_k^{(t)},\beta_k^{(t)}), \tag{8}$$

where θ_k—the probability of covering feature $x \in C(\mathcal{P}, \iota)$—is dependent on the inputs seen so far.

Unfortunately, using $\boldsymbol{\theta}$ (at Line 13, Algorithm 1) to select the next input will likely result in the fuzzer repeatedly selecting the same few inputs. This is because incrementing $\boldsymbol{\alpha}$ in Eq. (7) rewards the model, ultimately skewing the probability density function (PDF) towards one and making it more likely that the fuzzer will select this input. Similarly, incrementing $\boldsymbol{\beta}$ penalizes the model, skewing the PDF towards zero and making it less likely that the fuzzer will select this input. However, this has the side-effect of penalizing actions that are not selectable. We describe this issue and how we correct for it in Section 3.4.

3.4 Rareness Correction

Section 3.3 introduced the standard MAB setting, where we assume all K actions are selectable. However, this is not the case in practice: the fuzzer cannot select inputs corresponding to unexercised coverage map features. Moreover, the update rule in Eq. (7) penalizes rarely-covered features in C.

We use *feature rareness* to penalize frequently-covered coverage map features, introducing a *correction factor* to account for the update rule's penalty. This ensures favored inputs corresponding to less-covered features in the coverage map have a greater chance of being selected, prioritizing newly-discovered and hard-to-reach code. We apply this penalty using the chain rule on the joint probability between the reward at the next time step $\tilde{r}_k^{(t+1)}$ and the probability of the fuzzer selecting an input covering feature k, $\hat{\theta}_k^{(t)}$:

$$p(\tilde{r}_k^{(t+1)} = 1|\theta = \hat{\theta}_k^{(t)}) \propto p(\theta = \hat{\theta}_k^{(t)}|\tilde{r}_k^{(t+1)} = 1)p(\tilde{r}_k^{(t+1)} = 1), \tag{9}$$

where $\tilde{r}_k^{(t+1)}$ is the predicted reward at the next time step.

Eq. (9) is a binary classification with dependent variables. Here, the conditional probability with dependent variable $\tilde{r}_k^{(t+1)} = 1$ is drawn from Eq. (8), and the constraint for rareness applied by the marginal probability of $\tilde{r}_k^{(t+1)} = 1$ is

$$\psi_k^{(t)} \sim p(\tilde{r}_k^{(t+1)} = 1) = \mathrm{Beta}\left(\alpha_k^{(t)} + \beta_k^{(t)}, \left(\alpha_k^{(t)}\right)^2\right). \tag{10}$$

The conditional probability $p(\hat{\theta}_k^{(t)}|\tilde{r}_k^{(t+1)} = 1)$ represents the probability action a_k selects an ι that will discover new behaviors. The marginal probability $p(\tilde{r}_k^{(t+1)} = 1)$ applies a constraint penalizing features in the coverage map with a high hit count, thus prioritizing under-explored code. Eq. (11) shows this:

$$\phi_k^{(t)} = \mathbb{E}[p(\tilde{r}_k^{(t+1)} = 1)] = \frac{\alpha_k^{(t)} + \beta_k^{(t)}}{\left(\alpha_k^{(t)}\right)^2 + \alpha_k^{(t)} + \beta_k^{(t)}}. \tag{11}$$

For $\alpha_k >> \beta_k$, then $\phi_k^{(t)} \to \frac{1}{\alpha_k}$; i.e., a penalty is applied if the input is selected too frequently. Similarly, for $\beta_k >> \alpha_k$, then $\phi_k^{(t)} \to 1$; i.e., the penalty is removed if the input is infrequently chosen. This prioritizes rare features in C.

The correction factor is self-balancing: if less-explored favored inputs fail to discover any new inputs then $\beta_k \to \infty$ and $\phi_k^{(t)} \to 1$. After each time step, ϕ_k progressively removes the penalty if the fuzzer fails to discover any new interesting behavior. This allows θ_k to dominate the seed scheduling process. Similarly, ϕ_k dominates if new behaviors are quickly discovered.

3.5 Input Selection

A fuzzer must balance exploration and exploitation when selecting the "best" input to fuzz. Thompson sampling draws a value from K Beta distributions (Eq. (8)), then selects the next input to fuzz from the posterior distribution (Eq. (5)). Importantly, sampling a distribution generates a new distribution, enabling both exploration and exploitation.

Here, we present two approaches for selecting the next input to fuzz: input selection without and with rareness correction (Sections 3.3 and 3.4, respectively). This enables us to evaluate rareness correction's impact on fuzzing outcomes (Section 4.5).

We express input selection *without* rareness correction by sampling directly from Eq. (8); i.e.,

$$a^{(t+1)} = \arg\max[\hat{\theta}_1^{(t)}, \ldots, \hat{\theta}_K^{(t)}]. \tag{12}$$

After applying rareness correction, we express input selection as

$$a^{(t+1)} = \arg\max[\psi_1^{(t)}\hat{\theta}_1^{(t)}, \ldots, \psi_K^{(t)}\hat{\theta}_K^{(t)}], \tag{13}$$

where $\psi_k^{(t)}$ is computed by Eq. (10). The action $a^{(t+1)}$ is then used to select the next input from the set of favored inputs (Line 15, Algorithm 1). Intuitively the parameter $a^{(t+1)}$ represents the index of an edge with the highest scheduling probability (θ_k or $\bar{p}$) for each time-step (e.g., for $t = 4$ in Fig. 2, $a^{(t+1)}$ will select an input that has covered Line 3 because $\bar{p} = 0.31$ is the highest). An alternate formulation can be made by setting $\psi_k^{(t)}$ to the expected value $\phi_k^{(t)}$ in Eq. (11).

4 EVALUATION

We evaluate T-Scheduler over 35 CPU-yr of fuzzing, comparing it to four fuzzers and 11 schedulers across 35 programs from the Magma [17] and FuzzBench [28] benchmarks. Our evaluation aims to answer the following research questions:

RQ 1 Does T-Scheduler improve bug discovery? (Section 4.2)
RQ 2 Does T-Scheduler improve code coverage? (Section 4.3)
RQ 3 What are the run-time costs of T-Scheduler? (Section 4.4)
RQ 4 How do T-Scheduler's design choices impact fuzzing outcomes? (Section 4.5)

4.1 Methodology

Fuzzer Selection. We evaluate three T-Scheduler variants:

Rare$^-$ No rareness correction (Eq. (12)).
Rare$^+$ Rareness correction via the expected value (i.e., with $\psi_k^{(t)} := \phi_k^{(t)}$ in Eq. (13)).
Sample Rareness correction via sampling (Eq. (13)).

All three variants are implemented in AFL++ (v4.01a) [15]. We build on AFL++—rather than AFL [47]—because it incorporates state-of-the-art fuzzing improvements that T-Scheduler can leverage. Rare$^+$ deterministically selects the next input by computing the model's expected value. In contrast, Sample probabilistically selects the next input by drawing samples from the model's distribution.

To emphasize T-Scheduler's generality, we evaluate it against four fuzzers using two instrumentation schemes: LLVM compiler and QEMU binary. While QEMU significantly reduces fuzzer iteration rates, it is important to understand fuzzer performance when source code is not available. Importantly, this reduction is consistent across all fuzzers, so does not (dis)advantage any particular fuzzer. We select fuzzers using LLVM to answer RQ 1 and fuzzers using QEMU to answer RQ 2. These fuzzers are:

AFL++ (v4.01a) [15] The current state-of-the-art greybox fuzzer. We run AFL++ with eight supported power schedules [15]: the six AFLFast [7] schedules (EXPLORE, FAST, COE, QUAD, LIN, and EXPLOIT), and AFL++'s MMOPT (increases the score for new inputs to focus on newly-discovered paths) and RARE (ignores the input's run-time and focuses on inputs covering rarely-discovered features). We use AFL++ to answer both RQs 1 and 2. For RQ 1 we use LLVM's link-time optimization and also fuzz with an additional "CmpLog"-instrumented target (for logging comparison operands [4]).

`K-Scheduler` [35] Schedules inputs based on *Katz centrality* [21] analysis of the CFG. Katz centrality measures the "influence" of an input. This analysis helps seed scheduling by revealing the potential coverage gains from mutating a particular input. The AFL based `K-Scheduler` is implemented using LLVM's CFG analysis, so we use it to answer RQ 1.

TortoiseFuzz [43] Introduces three new coverage metrics for input scheduling operating on the function, loop, and basic block levels. TortoiseFuzz uses prior information on memory operations to gain further insights in prioritizing seeds leading to memory corruption bugs. This focus on memory corruption bugs and reliance on LLVM analyses means we use the AFL based TortoiseFuzz to answer RQ 1.

AFL-Hier [40] Combines a hierarchy of coverage metrics (ranging from coarse-grained to fine-grained) and an RL-based hierarchical seed scheduler for managing clusters of inputs (preventing fine-grained coverage metrics from flooding the queue). The AFL based AFL-Hier's hierarchy of coverage metrics is implemented in QEMU, and thus we use it to answer RQ 2.

Benchmark Selection. We evaluate these fuzzers on the Magma and FuzzBench benchmarks. At the time of writing, FuzzBench's `libxml` and `libpcap` failed to download. Thus, we omit these two targets from our evaluation. `K-Scheduler` also failed to construct CFGs for (and thus fuzz) *php* and *poppler*.

Experimental Setup. Each target is fuzzed for 72 h and repeated ten times to ensure statistically-sound results. We bootstrap each target with the default seeds provided by the benchmark. We conduct all experiments on a server with a 48-core Intel® Xeon® Gold 5118 2.30 GHz CPU, 512 GiB of RAM, and running Ubuntu 18.04.

4.2 Bug Discovery (RQ 1)

The ultimate goal of fuzzing is to find bugs. To this end, we evaluate the LLVM-based fuzzers presented in Section 4.1 on the Magma benchmark (>16 CPU-yr of fuzzing). Magma distinguishes between bugs *reached* and *triggered.* A bug is reached when "*the faulty line of code is executed*" (i.e., control-flow constraints are met) and triggered when "*the fault condition is satisfied*" (i.e., data-flow constraints are met). We focus on triggering bugs (not just reaching them) and say that fuzzer/scheduler $\mathcal{F}_1$ outperforms $\mathcal{F}_2$ on a given bug if (a) $\mathcal{F}_1$ finds the bug and $\mathcal{F}_2$ does not, or (b) $\mathcal{F}_1$ finds the bug faster than $\mathcal{F}_2$.

Table 1 shows that Sample was the best-performing scheduler across four out of five (80 %) of our bug-finding metrics ("total", "unique", "fastest", and "missed"), while Rare$^-$ scored the highest on the remaining metric ("best"). Of the eight AFL++ schedulers, FAST found the most bugs ("total" = 472) and was the second-best performer across the "best", "unique", "fastest", and "missed" metrics (after MMOPT, COE, EXPLORE, and COE, respectively). These results reinforce the AFL++ developers' decision to make FAST the default scheduler. Notably, the EXPLOIT scheduler (the

Table 1: Summary of Magma bug-finding results (across 10 trials). Count = number of bugs found in a target. Total = number of bugs found across all targets. Best = number of times a fuzzer found the most bugs in a given target. Unique = number of bugs found in any of the ten trials. Fastest = number of times a fuzzer found a bug first (per the restricted mean survival time [3] and log-rank test [26]). Missed = number of times a fuzzer failed to find a bug across all ten trials. Consistency = mean number of unique bugs found per trial (i.e., total / unique / # trials). The best-performing fuzzer(s) for each metric is in green. Targets that failed to build or run with the given fuzzer are marked with ✗. Full results are presented in Appendix A.

			Fuzzer												
			AFL++										T-Scheduler		
	Target	Driver	EXPLORE	FAST	COE	QUAD	LIN	EXPLOIT	MMOPT	RARE	K-Sched	Tortoise	Rare^-	Rare^+	Sample
Count	*libpng*	`libpng_read_fuzzer`	23	29	25	24	17	30	24	23	11	11	19	18	18
	libsndfile	`sndfile_fuzzer`	70	70	70	70	70	70	70	70	28	20	70	70	70
	libtiff	`tiff_read_rgba_fuzzer`	36	34	35	31	32	35	37	34	20	17	37	33	35
		`tiffcp`	49	49	48	43	43	50	49	52	41	34	53	48	49
	libxml2	`xml_read_memory_fuzzer`	31	34	34	30	34	30	40	31	10	10	30	34	35
		`xmllint`	27	25	28	22	25	20	27	24	13	10	27	28	31
	lua	`lua`	10	10	10	10	6	9	9	10	10	9	13	11	11
	openssl	`asn1`	18	20	19	18	18	20	20	19	11	10	20	20	20
		`client`	10	10	10	10	10	10	10	10	10	3	10	10	10
		`server`	10	10	10	10	10	10	10	10	20	20	18	18	16
		`x509`	0	1	1	0	0	3	0	4	0	8	0	0	0
	php	`exif`	16	19	18	14	14	26	18	20	✗	30	30	30	30
	poppler	`pdf_fuzzer`	32	27	27	25	25	31	30	30	✗	20	33	32	34
		`pdfimages`	35	42	40	29	25	32	38	32	✗	16	34	35	38
		`pdftoppm`	42	42	46	28	30	32	46	37	✗	20	38	40	37
	sqlite3	`sqlite3_fuzz`	45	50	38	29	49	33	42	39	2	0	36	41	43
Total			454	472	459	393	408	441	471	445	176	238	468	468	477
Best			2	5	3	2	2	4	6	2	2	3	7	4	6
Unique			63	63	65	54	61	59	63	60	24	29	62	63	65
Fastest			11	10	4	6	8	7	9	4	1	3	7	5	11
Missed			15	15	13	24	17	19	15	18	54	49	16	15	13
Consistency			0.72	0.75	0.71	0.73	0.67	0.75	0.75	0.74	0.73	0.82	0.75	0.73	0.73

original AFL's scheduler) was one of the worst performers (e.g., it was the third worst performer in missed bugs).

COE found more unique bugs (65) than Rare^- (62) and Rare^+ (63). However, the higher "total" results across the T-Scheduler variants suggest that T-Scheduler produces more-consistent bug-finding results. We reinforce this result with a "consistency" metric, defined as the total number of bugs divided by the number of unique bugs averaged across all ten trials. Based on this metric, Rare^- outperformed or performed as well as all AFL++ schedulers. That is, the number of bugs discovered by T-Scheduler remains consistent as the number of trials decreases.

All three T-Scheduler variants outperformed `K-Scheduler` and TortoiseFuzz—two state-of-the-art schedulers—across four of the five (80 %) bug-finding metrics ("total", "best", "unique", and "missed"). Sample (the best-performing T-Scheduler variant) outperformed or performed as well as `K-Scheduler` on 11 out of 12 drivers (we exclude *php* and *poppler* because they failed to build). Similarly, Sample outperformed or performed as well as TortoiseFuzz on 14 out of 16 drivers. We examine these results in the following sections.

4.2.1 `K-Scheduler` *Comparison.*

Of the 35 bugs found by Sample or `K-Scheduler`, the former: (i) outperformed `K-Scheduler` on 27 bugs; (ii) performed as well as `K-Scheduler` on six bugs; and (iii) was outperformed by `K-Scheduler` on two bugs. Notably, `K-Scheduler` failed to discover 16 of the 27 (59 %) bugs where T-Scheduler outperformed `K-Scheduler`. Moreover, `K-Scheduler` was, on average, ~50× slower at finding the remaining ten bugs. We attribute these results to `K-Scheduler` prioritizing "exploration" over "exploitation". Prior work on directed greybox fuzzing [6, 41, 49] has shown that fuzzers must both "explore" interesting code and "exploit" specific data-flow conditions to trigger bugs.[4] Concentrating on CFG expansion means that `K-Scheduler` does not focus on this exploitation phase, potentially harming bug discovery.

Finding: TIF009. One exception to these results is TIF009, which `K-Scheduler` found after 3.29 h (mean time over ten trials). In comparison, Rare^-, Rare^+, and Sample found it after 14.31, 33.37 and 33.77 h, respectively (4–10× slower). TIF009 (CVE-2019-7663 [29]) is a `NULL` pointer dereference located in `TIFFWriteDirectoryTagTransferfunction`. This vulnerable function is reachable via `TIFFWriteDirectory` when the `TransferFunction` field is set in a TIFF directory entry [1], and the bug is triggered when the transfer function pointers are `NULL`.

`K-Scheduler` and T-Scheduler trigger the bug within 3–120 s of reaching it, suggesting the bug does not require satisfying complex data-flow constraints (which pure mutational fuzzers—e.g., AFL++—may have difficulty satisfying). Moreover, multiple inputs from the initial seed set reach `TIFFWriteDirectory` without *any* mutation. Ultimately, `K-Scheduler`'s centrality score led it to the vulnerable function faster because it prioritized inputs that explored `TIFFWriteDirectory`, which had a relatively high centrality score (0.288). In comparison, neighboring CFG nodes had significantly lower centrality scores (0.047; while the median centrality score was 0.22). The relatively-simple data-flow constraints (a struct field set to `NULL`) meant that `K-Scheduler`'s prioritization of exploration over exploitation was an advantage.

[4] This is an unfortunate overloading of terms, and should not be confused with a MAB's exploration and exploitation phases.

4.2.2 TortoiseFuzz Comparison. 37 of the 55 (67 %) bugs discovered across all 13 fuzzers/schedulers were memory safety bugs (e.g., stack/heap buffer overflow/over-read, `NULL` pointer dereference). Thus, we expected TortoiseFuzz to demonstrate superior results in discovering these bugs (due to its design targeting memory safety bugs). However, TortoiseFuzz was outperformed by the other fuzzers/schedulers (in particular, T-Scheduler): it failed to trigger 20 of the 37 (54 %) memory safety bugs in *any* trial, and was slower at triggering another eight. TortoiseFuzz was outperformed by Rare$^-$, Rare$^+$, and Sample on 51, 53, and 58 bugs, respectively.

Notably TortoiseFuzz and `K-Scheduler` use a different set of heuristics based on the assumption that inputs reaching specific target sites are more likely to induce new program behaviors. TortoiseFuzz prioritizes memory-sensitive bugs allowing it to detect several memory-sensitive bugs earlier such as `TIF009`, `SSL009`, and `PDF010`. In contrast, T-Scheduler heuristics are derived from the assumption that (a) information derived from previous test cases can be used to improve input scheduling in the future, and (b) it is beneficial to spend resources on paths less explored to gain more information about the program to improve input scheduling. A potential improvement is to extend the model to be a directed greybox fuzzing where they have a different set of assumptions where the scheduler should prioritize specific sites such as memory-sensitive functions which are more likely to induce new program behaviors.

Finding: sqlite3. TortoiseFuzz failed to find any bugs in *sqlite3*. In contrast, Rare$^-$ found seven bugs, while Rare$^+$ and Sample found eight. This was due to the limited coverage expanded by TortoiseFuzz: it achieved only ~7 % line coverage, while Sample achieved >50 % line coverage. We attribute this low coverage to TortoiseFuzz's iteration rate: only 58 input/s. In contrast, T-Scheduler achieved an iteration rate >450 input/s. This reinforces the importance of reducing run-time overheads to maximize throughput.

Result 1

Sample was the (equal) best performer across the most (four out of five) bug-finding metrics: it found the most unique bugs, was the fastest, and missed the fewest bugs. Rare$^-$ outperformed in the "best" metric.

4.3 Code Coverage (RQ 2)

Bugs are sparse, making it challenging to evaluate fuzzers fairly using bug-centric metrics. Fuzzer evaluations commonly use code coverage as a proxy for evaluating fuzzer performance (a bug cannot be found in code never executed). We repeat this practice here.

We compare coverage using QEMU binary instrumentation on 19 FuzzBench targets (>19 CPU-yr of fuzzing). Comparisons are made across two measures: final coverage and coverage area under curve (AUC). Coverage is measured using Clang's source-based coverage instrumentation [13]. We use AUC because we found several targets had maximized coverage before the end of a 72 h trial. A higher AUC indicates that the fuzzer uncovers behaviors at a faster rate, which is important if the length of a fuzzing campaign is constrained. We use the Mann-Whitney U test to determine statistical significance [22].

Per Table 2, AFL++'s COE achieved the (equal) highest coverage on 16 of the 19 targets (84 %). Of the T-Scheduler variants, Rare$^-$ achieved the highest coverage on 13 targets (68 %) and was the next best performer after COE. Rare$^+$ and Sample performed similarly (equal best on five targets). Rare$^+$'s smaller 95 % bootstrap CI indicates less variance (across trials); we attribute this to Rare$^+$'s deterministic approach for rareness correction. AFL-Hier was the third worst performing fuzzers (only beating QUAD and LIN, and tying with EXPLORE). Curiously, it was the best performer on `lcms`, achieving twice as much coverage as the next best fuzzer.

COE was again the (equal) best performer for AUC (Table 3). However, this time it also tied with Rare$^-$. Following Böhme et al. [8], we use Cohen's kappa [14] to measure the agreement between total coverage and AUC of the best-performing fuzzer(s). We found the results in Tables 2 and 3 are in weak agreement ($\kappa = 0.56$). Outliers including `jsoncpp`, `openssl`, and `systemd` contributed to this weak agreement. For example, Rare$^-$ outperformed Rare$^+$ and Sample on `openssl` in Table 2, while the opposite is true in Table 3. These results were due to inaccurate AUC measurements: coverage measurements occur at 15 min intervals, and all fuzzers had saturated within the first 15 min.

Result 2

COE achieved the (equal) highest coverage on 16 of the 19 targets (84 %). Rare$^-$ tied with COE when using AUC.

4.4 Scheduler Overheads (RQ 3)

Scheduler overhead impacts a fuzzer's iteration rate. Prior work [18, 45] has shown that increased iteration rates lead to improved fuzzing outcomes. Here we investigate the scheduler's impact on iteration rates and fuzzing outcomes. To measure this impact, we: (i) instrument the scheduler to compute run-time overhead, recording the time the fuzzer spends updating the queue and selecting an input to fuzz; (ii) count the number of times the queue is updated (i.e., the number of times the fuzzer performs "filtering and favoring"); and (iii) examine the iteration rate reported by the fuzzer. We adopt the process used by She et al. [35] and run our instrumentation across 19 FuzzBench targets for 24 h, repeating this experiment ten times to minimize variance.

Table 4 shows that, despite an orders-of-magnitude increase in overhead, T-Scheduler achieves iteration rates comparable to (and, in most cases, higher than) the eight AFL++ schedulers. Of the eight AFL++ schedulers, FAST had the highest overhead (85 s), which we attribute to the high number of queue updates (873 over a single 24 h trial). However, this again did not impact iteration rates; FAST's iteration rate of 210 input/s was the highest. The `libpng` and `zlib` targets dominated this result, achieving iteration rates over 500 and 700 input/s, respectively.

Unlike T-Scheduler, AFL-Hier's relatively high update time (106 ms) and overhead (3 min) appeared to impact its iteration rate, which was the third lowest. We attribute these results to the hierarchical tree structure AFL-Hier uses to abstract the queue.

4.4.1 Scalability. Queue update times should remain constant (per trial) as the fuzzer expands coverage and the queue grows. This is particularly important for large fuzzing campaigns (e.g., Google's OSS-Fuzz [34]), where the queue can grow to tens of thousands of inputs. To this end, we investigate a scheduler's scalability by

Table 2: FuzzBench coverage, presented as mean coverage with 95 % bootstrap CI. The best-performing fuzzer(s) for each target (per the Mann-Whitney U test) is in green (larger is better). "Best" is the number of targets a fuzzer achieved the highest coverage.

Target	Fuzzer											
	AFL++								AFL-Hier	T-Scheduler		
	EXPLORE	FAST	COE	QUAD	LIN	EXPLOIT	MMOPT	RARE		Rare^{-}	Rare^{+}	Sample
bloaty	5,884.70 ± 109.18	5,767.60 ± 97.83	5,998.50 ± 74.35	5,514.40 ± 131.59	5,794.30 ± 108.91	6,069.50 ± 61.48	5,944.10 ± 141.67	5,935.50 ± 88.13	6,113.20 ± 126.46	5,918.20 ± 41.84	6,001.70 ± 47.54	6,069.20 ± 47.18
curl	18,968.10 ± 46.26	16,778.90 ± 35.11	19,375.40 ± 45.64	18,738.90 ± 149.82	18,660.50 ± 64.69	18,204.40 ± 148.61	19,164.40 ± 72.55	18,712.20 ± 88.77	13,370.10 ± 9.31	16,795.20 ± 124.08	16,776.30 ± 53.54	16,786.00 ± 79.30
freetype2	14,396.00 ± 197.46	14,585.80 ± 356.16	16,419.70 ± 181.27	13,836.00 ± 272.73	14,246.40 ± 204.66	15,609.50 ± 275.23	15,053.80 ± 399.48	15,364.90 ± 201.14	14,598.80 ± 924.03	15,860.40 ± 171.62	16,018.10 ± 219.93	15,734.60 ± 176.92
harfbuzz	7,642.40 ± 118.41	7,550.50 ± 104.96	7,777.10 ± 58.23	7,333.60 ± 144.77	7,222.30 ± 115.52	7,470.00 ± 71.30	7,640.00 ± 101.47	7,560.30 ± 70.59	6,616.10 ± 317.23	7,727.60 ± 31.45	7,577.60 ± 35.36	7,663.00 ± 24.29
jsoncpp	639.00 ± 0.00	639.00 ± 0.00	639.00 ± 0.00	639.00 ± 0.00	639.00 ± 0.00	636.00 ± 0.00	639.00 ± 0.00	639.00 ± 0.00	545.10 ± 56.91	639.00 ± 0.00	639.00 ± 0.00	639.00 ± 0.00
lcms	1,227.20 ± 0.58	1,223.80 ± 1.18	1,317.10 ± 17.45	1,235.80 ± 8.95	1,370.10 ± 136.16	1,262.00 ± 16.67	1,224.70 ± 1.83	1,235.30 ± 6.69	2,293.22 ± 135.79	1,254.70 ± 11.71	1,233.40 ± 2.09	1,251.40 ± 13.38
libjpeg-turbo	3,561.10 ± 55.74	3,656.60 ± 49.42	3,745.90 ± 41.05	3,451.40 ± 44.76	3,442.40 ± 21.84	3,140.70 ± 36.43	3,749.80 ± 31.54	3,057.90 ± 11.70	2,909.60 ± 75.03	3,764.40 ± 29.95	3,497.00 ± 47.71	3,517.60 ± 54.57
libpng	2,089.40 ± 4.80	2,084.10 ± 5.07	2,084.50 ± 7.94	2,076.10 ± 5.29	2,082.40 ± 6.32	2,091.40 ± 4.53	2,097.40 ± 5.72	2,056.20 ± 7.82	1,619.00 ± 32.11	2,079.20 ± 6.11	2,074.30 ± 6.13	2,064.10 ± 7.27
mbedtls	7,698.00 ± 10.83	7,642.30 ± 19.18	8,147.80 ± 17.13	7,659.90 ± 13.89	7,701.80 ± 13.86	7,749.20 ± 13.00	7,652.20 ± 5.68	7,693.10 ± 16.09	7,467.10 ± 100.39	8,025.70 ± 26.42	7,652.70 ± 21.38	7,631.20 ± 24.86
openssl	13,731.20 ± 3.34	13,745.80 ± 3.13	13,750.80 ± 3.72	13,720.10 ± 2.15	13,730.30 ± 4.85	13,754.30 ± 2.10	13,738.80 ± 4.11	13,728.60 ± 4.80	13,695.30 ± 4.64	13,738.40 ± 4.33	13,731.80 ± 3.71	13,735.80 ± 4.42
openthread	5,643.20 ± 38.42	5,764.80 ± 20.37	5,703.10 ± 31.87	5,364.10 ± 42.34	5,390.20 ± 59.08	5,706.60 ± 15.50	5,740.70 ± 29.01	5,750.50 ± 32.94	5,424.22 ± 157.82	5,802.70 ± 18.32	5,594.80 ± 79.15	5,617.40 ± 55.67
php	42,342.10 ± 106.95	42,459.60 ± 100.18	42,961.10 ± 61.12	41,627.00 ± 61.70	41,505.80 ± 38.58	41,907.40 ± 109.88	42,364.50 ± 136.80	42,344.90 ± 99.31	41,117.30 ± 534.67	44,736.60 ± 70.01	42,190.00 ± 143.45	44,198.10 ± 73.78
proj4	5,610.70 ± 141.28	5,699.10 ± 76.92	6,579.70 ± 91.73	5,132.90 ± 80.43	5,112.60 ± 54.38	5,310.30 ± 132.76	5,807.10 ± 121.24	5,129.00 ± 86.67	2,320.80 ± 369.80	6,466.50 ± 106.58	6,092.50 ± 205.82	6,132.00 ± 67.97
re2	3,506.10 ± 3.59	3,498.50 ± 4.15	3,518.00 ± 4.18	3,486.00 ± 7.05	3,504.00 ± 1.69	3,475.70 ± 4.83	3,502.60 ± 5.71	3,502.70 ± 6.87	3,055.30 ± 22.87	3,530.10 ± 5.40	3,486.70 ± 7.58	3,484.50 ± 4.90
sqlite3	21,191.10 ± 151.57	21,651.90 ± 306.67	23,210.00 ± 366.46	21,281.40 ± 113.28	21,558.00 ± 159.46	23,197.10 ± 386.04	22,097.90 ± 194.40	22,479.50 ± 127.28	19,041.60 ± 177.51	23,301.90 ± 577.27	22,028.80 ± 78.95	22,021.20 ± 143.47
systemd	637.60 ± 0.92	639.40 ± 0.56	640.00 ± 0.00	640.00 ± 0.00	638.20 ± 1.21	633.40 ± 1.57	638.80 ± 1.14	639.40 ± 0.57	613.56 ± 13.21	640.00 ± 0.00	640.00 ± 0.00	640.00 ± 0.00
vorbis	2,098.40 ± 16.27	1,991.70 ± 39.81	2,140.40 ± 8.86	2,030.90 ± 26.69	2,061.70 ± 32.15	2,043.70 ± 17.80	2,087.10 ± 18.38	1,986.90 ± 23.53	1,549.00 ± 0.00	2,156.30 ± 4.42	2,073.20 ± 8.68	2,062.70 ± 22.25
woff2	1,769.80 ± 14.61	1,694.40 ± 13.88	1,841.30 ± 9.47	1,773.10 ± 7.82	1,744.60 ± 14.93	1,731.40 ± 5.95	1,775.10 ± 6.69	1,678.00 ± 8.61	1,498.56 ± 49.29	1,860.90 ± 9.41	1,717.60 ± 5.69	1,699.10 ± 7.14
zlib	945.40 ± 4.12	942.20 ± 5.24	953.80 ± 3.91	932.80 ± 3.71	932.20 ± 3.42	940.20 ± 4.81	947.20 ± 5.06	948.50 ± 5.02	920.30 ± 3.69	960.10 ± 1.65	954.90 ± 4.67	959.40 ± 3.49
Best	3	5	16	2	2	5	6	4	3	13	5	5

examining how much queue update times vary (across a single 24 h trial). Table 4 shows these results under "update variance".

Like queue update time, queue update time variance is negligible. T-Scheduler has effectively no variance, making it ideal for long-running fuzzing campaigns. AFL-Hier and AFL++'s RARE have the highest variance. This is unsurprising; RARE focuses on queue entries that hit rare coverage map elements, requiring additional computation to find those rare elements, while operations on AFL-Hier's hierarchical tree structure have $O(n)$ complexity.

> **Result 3**
>
> T-Scheduler maintains high iteration rates, despite increased scheduling overheads. T-Scheduler is also scalable: queue update times remain constant over fuzzing campaigns.

4.5 Ablation Study (RQ 4)

We undertake an ablation study to understand better the impacts the individual T-Scheduler components have on our results. We study: (i) AFL++ FAST; (ii) Rare^{-}, replacing the FAST scheduler with a MAB; and (iii) Sample, adding a correction for rareness. We select FAST because it is AFL++'s default scheduler, and Sample because it generally outperforms Rare^{+} (RQs 1 and 2).

Figure 4 visualizes the unique bug counts from Table 1. FAST found 63 unique bugs, of which five were only found by FAST. Replacing FAST with Rare^{-} decreased the number of unique bugs to 62. However, Rare^{-} found two bugs missed by the other two schedulers. When adding rareness correction (via Sample) to Rare^{-}, the number of bugs increased to 65. Moreover, four of these 65 bugs were only found by Sample, and were found (a) late into the fuzzing process (after ~60 h), and (b) by only a small number of fuzzers/schedulers, suggesting they are difficult to trigger.

Table 3: FuzzBench AUC, presented as mean AUC with 95 % bootstrap CI. The best-performing fuzzer(s) for each target (per the Mann-Whitney U test) is in green (larger is better). "Best" is the number of times a fuzzer achieved the highest AUC for the evaluated targets.

Target	Fuzzer											
	AFL++								AFL-Hier	T-Scheduler		
	EXPLORE	FAST	COE	QUAD	LIN	EXPLOIT	MMOPT	RARE		Rare$^-$	Rare$^+$	Sample
bloaty	1,855.49 ± 36.81	1,798.74 ± 29.02	1,881.75 ± 21.23	1,739.14 ± 37.05	1,832.82 ± 32.55	1,901.46 ± 18.05	1,860.15 ± 39.57	1,882.21 ± 26.01	720.66 ± 88.80	1,876.27 ± 12.40	1,903.64 ± 12.70	1,917.34 ± 12.21
curl	5,935.13 ± 32.79	5,297.44 ± 8.34	6,128.42 ± 21.96	5,876.09 ± 49.73	5,845.46 ± 32.48	5,685.59 ± 42.23	5,981.73 ± 11.66	5,899.21 ± 16.57	22.31 ± 4.41	5,270.48 ± 31.77	5,284.60 ± 15.11	5,278.60 ± 23.71
freetype2	4,485.76 ± 63.39	4,524.82 ± 102.24	5,026.24 ± 53.00	4,316.93 ± 83.98	4,433.65 ± 52.09	4,794.04 ± 71.91	4,617.88 ± 108.91	4,691.47 ± 37.17	1,596.44 ± 349.75	4,892.82 ± 58.15	4,911.98 ± 46.05	4,821.05 ± 42.39
harfbuzz	2,367.14 ± 34.39	2,326.66 ± 31.08	2,407.32 ± 16.32	2,270.86 ± 41.14	2,237.21 ± 30.87	2,292.35 ± 21.88	2,358.83 ± 30.49	2,340.57 ± 21.70	1,317.19 ± 211.41	2,425.86 ± 11.06	2,372.57 ± 13.43	2,396.13 ± 7.54
jsoncpp	203.63 ± 0.07	203.23 ± 0.17	203.71 ± 0.05	203.54 ± 0.02	203.69 ± 0.01	202.66 ± 0.02	203.46 ± 0.11	203.49 ± 0.01	80.35 ± 15.07	202.70 ± 0.28	203.39 ± 0.17	203.27 ± 0.15
lcms	389.70 ± 0.33	387.03 ± 0.57	393.74 ± 0.81	390.46 ± 0.96	406.59 ± 15.81	392.60 ± 1.47	388.93 ± 0.49	390.42 ± 0.46	572.34 ± 48.40	394.88 ± 1.99	389.15 ± 0.57	390.45 ± 1.55
libjpeg-turbo	1,085.49 ± 9.90	1,096.15 ± 11.33	1,160.83 ± 11.89	1,071.70 ± 8.03	1,072.38 ± 3.32	973.90 ± 4.95	1,121.60 ± 12.41	954.81 ± 6.06	750.88 ± 38.03	1,173.16 ± 11.36	1,064.67 ± 15.36	1,067.75 ± 16.61
libpng	663.27 ± 1.60	660.20 ± 1.99	663.96 ± 2.50	657.04 ± 2.20	660.73 ± 2.07	661.04 ± 1.43	665.76 ± 1.78	650.26 ± 2.72	442.56 ± 28.84	658.86 ± 2.12	657.18 ± 1.97	652.64 ± 2.12
mbedtls	2,436.35 ± 2.24	2,419.82 ± 5.30	2,541.22 ± 5.61	2,436.52 ± 3.85	2,446.75 ± 3.71	2,455.33 ± 3.24	2,420.35 ± 3.02	2,436.75 ± 3.06	1,941.63 ± 231.06	2,471.21 ± 9.09	2,428.81 ± 6.12	2,418.06 ± 7.74
openssl	4,369.59 ± 3.07	4,370.03 ± 4.42	4,382.95 ± 1.09	4,372.61 ± 0.94	4,376.14 ± 1.51	4,378.27 ± 0.70	4,373.62 ± 1.99	4,374.11 ± 2.01	2,414.84 ± 513.73	4,355.91 ± 7.69	4,377.09 ± 1.14	4,368.88 ± 4.03
openthread	1,748.75 ± 17.22	1,762.49 ± 12.70	1,791.13 ± 11.93	1,668.19 ± 13.06	1,660.39 ± 19.64	1,746.66 ± 8.34	1,790.89 ± 10.25	1,784.31 ± 6.85	1,254.70 ± 180.57	1,825.09 ± 6.25	1,729.62 ± 21.97	1,713.15 ± 16.16
php	13,390.60 ± 31.39	13,388.88 ± 32.36	13,584.21 ± 23.81	13,194.36 ± 9.33	13,177.75 ± 9.90	13,183.46 ± 25.63	13,389.12 ± 36.87	13,337.44 ± 23.16	1,151.67 ± 280.18	14,016.82 ± 19.42	13,414.13 ± 38.98	14,016.68 ± 26.92
proj4	1,647.87 ± 43.23	1,670.72 ± 28.16	2,020.40 ± 28.88	1,491.32 ± 30.43	1,467.89 ± 23.12	1,426.01 ± 41.85	1,671.35 ± 46.43	1,327.46 ± 30.11	436.16 ± 76.83	1,977.17 ± 39.66	1,770.04 ± 66.98	1,785.75 ± 25.78
re2	1,107.15 ± 1.24	1,102.90 ± 1.06	1,116.69 ± 0.94	1,097.35 ± 1.67	1,099.13 ± 1.17	1,082.91 ± 2.76	1,103.85 ± 1.47	1,097.64 ± 1.79	719.14 ± 32.05	1,121.36 ± 1.38	1,100.35 ± 2.12	1,095.72 ± 2.18
sqlite3	6,546.16 ± 37.64	6,549.26 ± 84.99	6,966.93 ± 70.91	6,659.31 ± 23.54	6,684.83 ± 42.71	6,776.79 ± 62.16	6,770.59 ± 31.57	6,917.85 ± 23.47	3,163.01 ± 228.67	7,032.32 ± 78.15	6,862.27 ± 20.75	6,815.92 ± 31.30
systemd	202.70 ± 0.37	202.93 ± 0.20	204.00 ± 0.02	202.95 ± 0.25	202.97 ± 0.41	200.68 ± 0.33	202.86 ± 0.45	202.53 ± 0.28	148.28 ± 26.43	203.38 ± 0.28	203.29 ± 0.16	203.16 ± 0.26
vorbis	621.37 ± 4.06	597.84 ± 3.77	655.33 ± 2.96	613.50 ± 4.78	618.62 ± 4.97	627.95 ± 4.83	624.02 ± 5.24	616.59 ± 5.06	2.07 ± 0.22	669.33 ± 1.99	625.59 ± 4.40	625.32 ± 4.44
woff2	540.76 ± 1.79	528.67 ± 2.04	564.95 ± 2.12	549.08 ± 1.01	543.36 ± 3.23	540.87 ± 1.03	541.73 ± 1.71	526.76 ± 1.75	197.68 ± 73.95	580.63 ± 2.47	538.27 ± 1.04	533.78 ± 1.61
zlib	299.20 ± 1.05	298.56 ± 1.48	301.50 ± 1.22	296.07 ± 0.77	295.10 ± 0.50	291.20 ± 0.75	300.56 ± 1.34	299.10 ± 1.35	273.63 ± 10.12	302.06 ± 1.34	301.07 ± 1.26	302.36 ± 1.04
Best	4	2	13	0	0	2	4	3	1	13	4	3

Table 4: Scheduler overheads and iteration rates with 95 % bootstrap CI. Update count = number of times the queue is updated in a single trial (geometric mean). Update time = time spent (ms) on each queue update (arithmetic mean). Update variance = how much the queue update time varies (ms^2) in a single trial (arith. mean). Overhead = total time (s) the fuzzer spends selecting an input to fuzz in a trial (arith. mean). Iteration rate = number of inputs executed per second (arith. mean).

	Fuzzer											
	AFL++								AFL-Hier	T-Scheduler		
	EXPLORE	FAST	COE	QUAD	LIN	EXPLOIT	MMOPT	RARE		Rare$^-$	Rare$^+$	Sample
Update count (#)	269.97 ± 44.62	873.86 ± 156.92	283.12 ± 46.29	328.12 ± 57.84	590.47 ± 135.07	80.03 ± 7.91	281.25 ± 46.43	124.42 ± 17.73	88.84 ± 17.17	672.38 ± 87.68	649.44 ± 86.14	635.84 ± 81.63
Update time (ms)	14.96 ± 3.47	9.29 ± 1.32	9.52 ± 1.30	15.17 ± 3.26	16.77 ± 7.29	29.88 ± 7.25	15.62 ± 2.83	39.77 ± 7.99	106.55 ± 23.59	41.76 ± 0.09	42.05 ± 0.11	79.83 ± 0.23
Update variance (ms^2)	0.40 ± 0.37	0.01 ± 0.00	0.00 ± 0.00	0.09 ± 0.04	0.12 ± 0.10	0.11 ± 0.06	0.04 ± 0.02	0.52 ± 0.32	0.75 ± 0.51	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
Overhead (s)	57.84 ± 9.32	85.35 ± 11.90	49.34 ± 7.66	58.18 ± 9.37	62.00 ± 9.23	49.18 ± 7.02	56.38 ± 8.71	64.13 ± 10.14	207.68 ± 47.91	2,488.36 ± 319.63	2,419.26 ± 317.35	4,443.65 ± 559.37
Iteration rate (inputs/s)	83.52 ± 13.33	210.19 ± 39.23	85.06 ± 12.20	89.26 ± 16.73	87.61 ± 15.89	99.86 ± 16.94	89.42 ± 13.30	58.52 ± 9.79	84.84 ± 17.28	96.79 ± 16.75	94.46 ± 16.36	93.60 ± 16.67

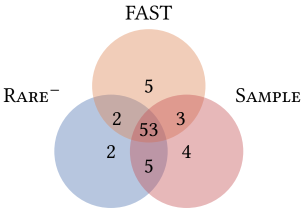


Figure 4: Venn diagram of unique bug count between AFLFast, Rare⁻, and Sample (74 unique bugs found across all drivers).

Sample's bug-finding performance improved towards the end of the fuzzing campaign (here, after 45 h of fuzzing), triggering nine bugs that were missed by FAST. Three of these bugs (`XML001`, `XML002`, and `PDF011`) are buffer overflow bugs in functions that are frequently covered by FAST. FAST assigns higher priority to an input depending on the number of times it covers an edge, while penalizing inputs that cover paths with high frequency [7]. This is a limitation of FAST, as a high-frequency path does not necessarily mean a lower chance of discovering new program behavior. T-Scheduler does not apply a penalty to high-frequency paths if previously scheduled inputs have discovered new program behaviors.

Code coverage increased when replacing the FAST scheduler with Rare⁻'s MAB-based scheduler. Notably, Rare⁻ achieved the (equal) highest coverage on most FuzzBench targets (Tables 2 and 3). In particular, `harfbuzz` was the only target where FAST achieved the most coverage and did not tie with Rare⁻ (Table 2). This is in contrast to bug finding, where Sample outperformed Rare⁻ (except for the "best" metric in Table 1). We attribute this to Rare⁻'s single objective (in the RL algorithm) to maximize code coverage. In contrast, Sample uses a multi-objective optimization that balances exploring rare paths and maximizing code coverage.

Iteration rates decreased by ~50 % when replacing FAST with Rare⁻ (Table 4). However, this decrease was also true of the other seven AFL++ schedulers (and was sometimes even more pronounced; up to 72 %). Sample increased the scheduler overhead from 40 min to 74 min (an 84 % increase). We attribute this increase to costs associated with sampling from the Beta distribution twice (Eq. (13)). In contrast, Rare⁻ only samples from it once. However, this has negligible impact on iteration rates: a reduction of ~1 input/s. Similarly, the variance across queue update times remains zero when replacing Rare⁻ with Sample, demonstrating its scalability.

To Sample or not to Sample? Our ablation study focused on Sample, which probabilistically samples for rareness correction. However, it is worth revisiting Rare⁺—which deterministically computes an expected value for rareness correction—to understand what impact the probabilistic approach has on fuzzing outcomes. Sample outperformed Rare⁺ across all bug-finding metrics (Table 1). Sample also slightly outperformed Rare⁺ (by a single target) on total coverage (Table 2) and performed the same on coverage AUC (Table 3). While probabilistically sampling resulted in higher scheduler overhead, this had a negligible impact on the fuzzer's iteration rate (Table 4). In most applications, the probabilistic approach should outperform a deterministic approach. Thus, we recommend Sample for general use based on our results.

Result 4

Accounting for rare coverage elements (with Sample) improves bug-finding performance (over Rare⁻). However, Rare⁻'s single objective of maximizing coverage leads to higher coverage. Both improve upon FAST.

5 RELATED WORK

Woo et al. [44] were one of the first to formulate fuzzing as a MAB. Their work focused on *blackbox* fuzzing, where there is no coverage feedback to guide the fuzzer. Woo et al. [44] proposed using the exp3 algorithm [33], with the fuzzer rewarded based on the number of unique bugs found. However, bugs are sparse, so assigning a reward based on their discovery is impractical. In contrast, AFLFast [7] focused on *greybox* fuzzing, introducing the power schedule for seed scheduling (see Section 2.2). Entropic [5] expanded this with an entropy-based power schedule for prioritizing *information gain*.

Prior work has applied MAB to greybox fuzzing. Karamcheti et al. [20], Koike et al. [23] use a MAB over the fuzzer's mutation operators, while EcoFuzz [46], AFL-Hier [40], and MobFuzz [48] use a MAB for seed scheduling.

EcoFuzz [46] eschews a traditional MAB for an *adversarial bandit*. While an adversarial bandit removes assumptions about the bandits' probability distributions, this generality requires more hyperparameters to tune. In particular, EcoFuzz has two hyperparameters—exploration and decay—and uses entropy to compute the probability of an input's (potential) information gain.

AFL-Hier [40] combines a multiplayer MAB with a hierarchical tree structure to balance exploration/exploitation across coverage metrics. Like EcoFuzz, AFL-Hier has two hyperparameters (exploration and decay). However, unlike EcoFuzz, AFL-Hier rewards rare code paths, encouraging the scheduler to select inputs that cover less-explored paths.

Finally, MobFuzz [48] uses a multi-objective optimization formulated as a multiplayer MAB, maximizing three objectives: execution speed, memory consumption, and the length of memory comparisons. MobFuzz uses a weighted average of these three objectives to compute the reward for each input.

6 CONCLUSIONS

We present T-Scheduler, an RL-based fuzzer seed scheduler. Unlike prior RL-based schedulers, T-Scheduler does not require hyperparameter tuning. We empirically demonstrate T-Scheduler's effectiveness over 35 CPU-yr of fuzzing. On Magma, T-Scheduler found the most bugs and missed the fewest. On FuzzBench, it was the fastest at expanding coverage on the most targets. T-Scheduler also maintains consistently-high iteration rates (even as the queue grows). Given our results, we recommend the adoption of our Sample variant. Interestingly, our results also show that AFL++'s default FAST scheduler was generally outperformed by the MMOPT scheduler (across both bug-finding and coverage-expansion metrics). We encourage others to adopt and build upon T-Scheduler, available at https://github.com/asiaccs2024-t-scheduler.

ACKNOWLEDGMENTS

This work was supported by the Defence Science and Technology Group Next Generation Technologies Fund (Cyber) program via the *KRONOS* Data61 Collaborative Research Project.

REFERENCES

[1] Adobe. 1992. TIFF, Revision 6.0. https://www.loc.gov/preservation/digital/formats/fdd/fdd000022.shtml

[2] Shipra Agrawal and Navin Goyal. 2017. Near-Optimal Regret Bounds for Thompson Sampling. *Jorunal of the ACM* 64, 5, Article 30 (2017), 24 pages. https://doi.org/10.1145/3088510

[3] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *International Conference on Software Engineering (ICSE)*. ACM, 1–10. https://doi.org/10.1145/1985793.1985795

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Network and Distributed System Security (NDSS)*. The Internet Society, 15 pages. https://doi.org/10.14722/ndss.2019.23371

[5] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*. ACM, 678–689. https://doi.org/10.1145/3368089.3409748

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Computer and Communications Security (CCS)*. ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Computer and Communications Security (CCS)*. ACM, 1032–1043. https://doi.org/10.1145/2976749.2978428

[8] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *International Conference on Software Engineering (ICSE)*. ACM, 1621–1633. https://doi.org/10.1145/3510003.3510230

[9] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep Reinforcement Fuzzing. In *Security and Privacy Workshops (SPW)*. IEEE, 116–122. https://doi.org/10.1109/SPW.2018.00026

[10] Oliver Chang. 2023. Taking the next step: OSS-Fuzz in 2023. https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html

[11] Yaohui Chen, Mansour Ahmadi, Reza Mirzazade farkhani, Boyu Wang, and Long Lu. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In *Research in Attacks, Intrusions and Defenses (RAID)*. USENIX, 77–92.

[12] Liang Cheng, Yang Zhang, Yi Zhang, Chen Wu, Zhangtan Li, Yu Fu, and Haisheng Li. 2019. Optimizing Seed Inputs in Fuzzing with Machine Learning. In *International Conference on Software Engineering: Companion (ICSE)*. IEEE, 244–245. https://doi.org/10.1109/ICSE-Companion.2019.00096

[13] Clang Team. 2022. Source-based Code Coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

[14] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104

[15] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Workshop on Offensive Technologies (WOOT)*. USENIX, 12 pages.

[16] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Automated Software Engineering (ASE)*. IEEE, 50–59.

[17] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Measurement and Analysis of Computing Systems* 4, 3, Article 49 (2020), 29 pages. https://doi.org/10.1145/3428334

[18] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. Seed Selection for Successful Fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 230–243. https://doi.org/10.1145/3460319.3464795

[19] Adrian Herrera, Mathias Payer, and Antony L. Hosking. 2022. Registered Report: datAFLow Towards a Data-Flow-Guided Fuzzer. In *Fuzzing Workshop (FUZZING)*. The Internet Society, 11 pages. https://doi.org/10.14722/fuzzing.2022.23001

[20] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. 2018. Adaptive Grey-Box Fuzz-Testing with Thompson Sampling. In *Artificial Intelligence and Security (AISec)*. ACM, 37–47. https://doi.org/10.1145/3270101.3270108

[21] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* (1953), 39–43. https://doi.org/10.1007/BF02289026

[22] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Computer and Communications Security (CCS)*. ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[23] Yuki Koike, Hiroyuki Katsura, Hiromu Yakura, and Yuma Kurogome. 2022. SLOPT: Bandit Optimization Framework for Mutation-Based Fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*. ACM, 519–533. https://doi.org/10.1145/3564625.3564659

[24] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit algorithms*. Cambridge University Press.

[25] Siqi Li, Xiaofei Xie, Yun Lin, Yuekang Li, Ruitao Feng, Xiaohong Li, Weimin Ge, and Jin Song Dong. 2022. Deep Learning for Coverage-Guided Fuzzing: How Far are We? *Transactions on Dependable and Secure Computing* (2022), 1–13. https://doi.org/10.1109/TDSC.2022.3200525

[26] Nathan Mantel. 1966. Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemotherapy Reports* 50, 3 (1966), 163–170.

[27] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[28] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*. ACM, 1393–1403. https://doi.org/10.1145/3468264.3473932

[29] NIST. 2019. CVE-2019-7663. https://nvd.nist.gov/vuln/detail/CVE-2019-7663

[30] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *USENIX Security (SEC)*. USENIX, 861–875.

[31] Gary J. Saavedra, Kathryn N. Rodhouse, Daniel M. Dunlavy, and W. Philip Kegelmeyer. 2019. A Review of Machine Learning Applications in Fuzzing. *arXiv Preprint* abs/1906.11133 (2019), 12 pages. https://doi.org/10.48550/arXiv.1906.11133

[32] Joseph Scott, Federico Mora, and Vijay Ganesh. 2020. BanditFuzz: A Reinforcement-Learning Based Performance Fuzzer for SMT Solvers. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*. Springer-Verlag, 68–86. https://doi.org/10.1007/978-3-030-63618-0_5

[33] Yevgeny Seldin, Csaba Szepesvári, Peter Auer, and Yasin Abbasi-Yadkori. 2013. Evaluation and Analysis of the Performance of the EXP3 Algorithm in Stochastic Environments. In *Proceedings of Machine Learning Research (PMLR, Vol. 24)*. PMLR, 103–116. http://proceedings.mlr.press/v24/seldin12a.html

[34] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security (SEC)*. USENIX.

[35] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *Security and Privacy (S&P)*. IEEE, 2194–2211. https://doi.org/10.1109/SP46214.2022.9833761

[36] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction.* MIT press.

[37] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3-4 (1933), 285–294.

[38] Jonas Benedict Wagner. 2017. *Elastic Program Transformations: Automatically Optimizing the Reliability/Performance Trade-off in Systems Software.* Ph. D. Dissertation. EPFL. https://doi.org/10.5075/epfl-thesis-7745

[39] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *USENIX Security (SEC)*. USENIX, 2741–2758.

[40] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 17 pages. https://doi.org/10.14722/ndss.2021.24486

[41] Pengfei Wang and Xu Zhou. 2020. SoK: The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing. *CoRR* abs/2005.11907 (2020).

[42] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. 2020. A systematic review of fuzzing based on machine learning techniques. *PLOS ONE* 15, 8 (2020), 1–37. https://doi.org/10.1371/journal.pone.0237749

[43] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Network and Distributed Systems Security (NDSS)*. The Internet Society, 17 pages. https://doi.org/10.14722/ndss.2020.24422

[44] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-Box Mutational Fuzzing. In *Computer and Communications Security (CCS)*. ACM, 511–522. https://doi.org/10.1145/2508859.2516736

[45] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Computer and Communications Security (CCS)*. ACM, 2313–2328. https://doi.org/10.1145/3133956.3134046

[46] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *USENIX Security (SEC)*. USENIX, 2307–2324.

[47] Michał Zalewski. 2015. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/

[48] Gen Zhang, Pengfei Wang, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, and Kai Lu. 2022. MobFuzz: Adaptive Multi-objective Optimization in Gray-box Fuzzing. In *Network and Distributed Systems Security (NDSS)*. The Internet Society, 18 pages. https://doi.org/10.14722/ndss.2022.24314

[49] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2022. FishFuzz: Throwing Larger Nets to Catch Deeper Bugs. *CoRR* abs/2207.13393 (2022). https://doi.org/10.48550/arXiv.2207.13393

[50] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-box Fuzzing through Deep Learning. In *USENIX Security (SEC)*. USENIX, 2255–2269.

A MAGMA SURVIVAL ANALYSIS

Following prior work [3, 17–19, 38], we model bug finding using survival analysis. This allows us to reason about censored data; i.e., the case where a fuzzer does not find a bug. Table 5 presents the restricted mean survival time (RMST) of a given bug; i.e., the mean time the bug "survives" being discovered by a fuzzer across ten repeated 72 h campaigns. Lower RMSTs imply a fuzzer finds a bug "faster", while a smaller confidence interval (CI) means the bug is found more consistently. Applying the log-rank test [26] under the null hypothesis that two fuzzers share the same survival function allows us to statistically compare survival times. Thus, two fuzzers have statistically equivalent bug survival times if the log-rank test's p-value > 0.05. The survival analysis results in Table 5 augment those presented in the main paper.

Table 5: Magma bugs triggered, presented as the restricted mean survival time (RMST; in hours) with 95 % bootstrap CI. Bugs never found by a particular fuzzer have an RMST of ⊤ (to distinguish bugs with a 72 h RMST). Targets that fail to build with a given fuzzer are marked with ✗. The best-performing fuzzer (fuzzers if the bug survival times are statistically equivalent per the log-rank test) for each bug is highlighted in green (smaller is better).

Target	Driver	Bug	Fuzzer												
			AFL++								K-Sched	Tortoise	T-Scheduler		
			EXPLORE	FAST	COE	QUAD	LIN	EXPLOIT	MMOPT	RARE			Rare^{-}	Rare^{+}	Sample
libpng	`libpng_read_fuzzer`	PNG001	71.51 ± 1.67	⊤	⊤	⊤	⊤	70.53 ± 5.00	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PNG003	14.40 ± 25.24	0.01 ± 0.00	0.01 ± 0.01	7.21 ± 14.11	28.80 ± 30.92	0.01 ± 0.01	7.21 ± 18.93	7.21 ± 17.28	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01
		PNG006	14.45 ± 17.84	0.08 ± 0.05	0.04 ± 0.02	7.24 ± 12.76	28.83 ± 24.43	0.05 ± 0.03	7.25 ± 12.75	7.26 ± 12.75	⊤	⊤	⊤	⊤	⊤
		PNG007	39.28 ± 19.37	35.25 ± 13.41	51.15 ± 19.63	38.13 ± 21.00	47.51 ± 20.87	26.85 ± 14.18	42.04 ± 19.27	52.84 ± 18.18	68.36 ± 12.35	70.22 ± 6.03	28.31 ± 15.21	30.63 ± 14.04	28.02 ± 16.30
libsndfile	`sndfile_fuzzer`	SND001	0.64 ± 0.24	0.41 ± 0.11	0.46 ± 0.21	1.29 ± 0.52	1.43 ± 0.31	2.46 ± 1.64	0.56 ± 0.36	0.45 ± 0.17	34.02 ± 0.53	⊤	0.24 ± 0.08	0.21 ± 0.11	0.32 ± 0.08
		SND005	0.97 ± 0.27	0.78 ± 0.32	1.09 ± 0.42	3.92 ± 1.48	2.88 ± 1.07	6.57 ± 3.59	1.51 ± 0.68	1.02 ± 0.43	⊤	2.82 ± 1.20	0.41 ± 0.10	0.55 ± 0.24	0.48 ± 0.13
		SND006	1.11 ± 0.86	1.10 ± 1.23	0.85 ± 0.51	0.98 ± 0.46	5.69 ± 7.29	6.36 ± 2.68	1.00 ± 0.44	0.34 ± 0.14	68.24 ± 12.76	⊤	0.40 ± 0.14	0.45 ± 0.19	0.36 ± 0.08
		SND007	0.70 ± 0.32	0.85 ± 0.30	0.46 ± 0.27	1.27 ± 0.53	1.57 ± 0.61	2.86 ± 1.42	1.27 ± 0.56	0.66 ± 0.27	56.23 ± 15.46	⊤	0.60 ± 0.26	0.80 ± 0.18	0.79 ± 0.31
		SND017	0.34 ± 0.19	0.47 ± 0.31	0.57 ± 0.23	0.89 ± 0.69	1.67 ± 1.19	0.59 ± 0.15	0.57 ± 0.20	0.74 ± 0.41	1.94 ± 0.12	0.67 ± 0.13	1.35 ± 0.90	0.36 ± 0.31	0.34 ± 0.22
		SND020	0.75 ± 0.30	0.80 ± 0.29	1.06 ± 0.21	1.40 ± 0.49	2.18 ± 0.83	2.03 ± 0.74	1.12 ± 0.25	1.14 ± 0.27	⊤	⊤	2.96 ± 0.93	3.36 ± 1.52	2.63 ± 0.96
		SND024	0.59 ± 0.27	0.38 ± 0.27	0.30 ± 0.14	0.98 ± 0.46	0.93 ± 0.37	2.62 ± 1.27	0.97 ± 0.43	0.34 ± 0.14	60.41 ± 15.52	⊤	0.38 ± 0.15	0.45 ± 0.19	0.35 ± 0.08
libtiff	`tiff_read_rgba_fuzzer`	TIF002	60.02 ± 15.10	60.46 ± 18.66	60.19 ± 10.33	65.84 ± 20.91	66.72 ± 11.50	62.47 ± 14.48	56.93 ± 15.73	58.95 ± 17.79	⊤	⊤	58.99 ± 13.38	66.96 ± 8.80	64.17 ± 12.92
		TIF007	0.07 ± 0.04	0.08 ± 0.03	0.04 ± 0.02	0.12 ± 0.14	0.06 ± 0.03	0.05 ± 0.02	0.03 ± 0.02	0.04 ± 0.03	1.66 ± 0.40	4.45 ± 1.58	0.03 ± 0.02	0.04 ± 0.03	0.02 ± 0.01
		TIF008	67.16 ± 9.80	64.98 ± 23.84	⊤	⊤	⊤	66.81 ± 11.22	63.17 ± 17.41	67.89 ± 13.95	⊤	⊤	66.63 ± 14.58	⊤	64.90 ± 14.50
		TIF012	1.52 ± 0.56	1.92 ± 1.01	1.25 ± 0.34	3.05 ± 1.04	1.75 ± 0.35	1.44 ± 0.72	1.35 ± 0.36	1.84 ± 0.49	2.42 ± 0.54	51.10 ± 18.80	1.37 ± 0.66	0.97 ± 0.34	0.90 ± 0.39
		TIF014	5.63 ± 2.44	2.72 ± 1.17	4.17 ± 1.69	4.12 ± 2.89	3.11 ± 2.39	2.49 ± 1.27	3.68 ± 2.52	1.59 ± 0.65	⊤	64.30 ± 19.23	2.15 ± 1.41	3.85 ± 2.27	2.04 ± 0.98
	`tiffcp`	TIF002	⊤	68.29 ± 12.58	⊤	⊤	⊤	69.71 ± 7.78	70.72 ± 4.35	66.34 ± 10.97	⊤	⊤	65.47 ± 15.84	⊤	66.71 ± 10.45
		TIF005	69.44 ± 8.68	65.94 ± 20.57	65.84 ± 20.90	⊤	61.04 ± 22.01	66.74 ± 10.31	⊤	⊤	⊤	⊤	68.74 ± 11.05	⊤	⊤
		TIF006	22.19 ± 8.76	22.62 ± 13.97	13.46 ± 5.32	51.00 ± 17.15	46.21 ± 22.09	31.89 ± 14.87	16.42 ± 13.61	12.05 ± 4.82	64.89 ± 24.15	41.90 ± 17.22	14.92 ± 7.82	20.82 ± 12.40	20.53 ± 9.97
		TIF007	0.05 ± 0.03	0.06 ± 0.03	0.17 ± 0.16	0.14 ± 0.09	0.05 ± 0.03	0.07 ± 0.04	0.05 ± 0.03	0.05 ± 0.03	0.23 ± 0.11	9.52 ± 2.80	0.04 ± 0.02	0.04 ± 0.03	0.03 ± 0.02
		TIF008	65.04 ± 23.61	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF009	28.49 ± 19.49	30.93 ± 20.82	25.45 ± 19.99	37.69 ± 17.37	33.09 ± 22.14	23.03 ± 18.04	18.79 ± 11.26	19.39 ± 14.14	3.29 ± 2.11	10.62 ± 1.53	14.31 ± 3.47	33.37 ± 15.03	33.77 ± 17.58
		TIF012	1.26 ± 0.30	0.86 ± 0.31	1.33 ± 0.51	7.77 ± 5.61	2.41 ± 1.05	1.36 ± 0.45	0.89 ± 0.22	1.37 ± 0.57	7.30 ± 5.82	54.88 ± 15.72	2.43 ± 0.99	1.53 ± 0.79	1.15 ± 0.39
		TIF014	4.06 ± 1.99	3.18 ± 1.49	1.80 ± 0.60	9.53 ± 7.82	3.93 ± 2.29	2.48 ± 1.06	1.32 ± 0.43	1.05 ± 0.33	5.68 ± 2.66	61.01 ± 15.90	1.29 ± 0.61	0.93 ± 0.44	0.87 ± 0.39

Table 5: Magma bugs (cont.).

Target	Driver	Bug	Fuzzer												
			AFL++								K-Sched	Tortoise	T-Scheduler		
			EXPLORE	FAST	COE	QUAD	LIN	EXPLOIT	MMOPT	RARE			RARE$^-$	RARE$^+$	SAMPLE
libxml2	xml_read_memory_fuzzer	XML001	⊤	⊤	67.43 ± 8.15	⊤	⊤	⊤	43.49 ± 14.41	⊤	⊤	⊤	⊤	65.80 ± 8.42	65.02 ± 13.91
		XML002	⊤	⊤	⊤	⊤	71.33 ± 2.27	⊤	65.73 ± 21.28	67.52 ± 15.20	⊤	⊤	⊤	68.72 ± 11.15	61.70 ± 20.67
		XML003	5.49 ± 2.49	2.78 ± 2.09	2.59 ± 0.92	1.94 ± 1.16	2.63 ± 0.80	8.58 ± 5.46	9.29 ± 12.41	3.58 ± 1.82	⊤	⊤	4.93 ± 2.74	1.69 ± 0.83	2.84 ± 1.21
		XML009	1.11 ± 0.23	1.52 ± 0.48	1.43 ± 0.46	2.45 ± 0.92	5.16 ± 2.16	4.83 ± 1.73	8.16 ± 12.59	1.82 ± 0.88	⊤	⊤	1.55 ± 0.90	1.64 ± 0.91	1.20 ± 0.46
		XML012	69.16 ± 9.65	60.42 ± 11.63	70.18 ± 6.19	⊤	63.83 ± 12.93	⊤	48.18 ± 18.08	⊤	⊤	⊤	⊤	⊤	71.61 ± 1.33
		XML017	0.02 ± 0.02	0.02 ± 0.02	0.02 ± 0.02	0.04 ± 0.06	0.06 ± 0.04	0.02 ± 0.02	7.21 ± 16.00	0.03 ± 0.02	0.02 ± 0.02	0.03 ± 0.03	0.02 ± 0.02	0.02 ± 0.01	0.03 ± 0.02
	xmllint	XML001	58.72 ± 11.70	62.41 ± 9.50	63.36 ± 7.42	68.58 ± 11.62	60.06 ± 16.06	⊤	54.85 ± 11.93	65.02 ± 10.46	⊤	⊤	62.34 ± 8.09	52.02 ± 11.68	57.17 ± 8.30
		XML002	65.11 ± 14.82	71.07 ± 3.16	68.13 ± 13.14	⊤	66.00 ± 20.38	⊤	⊤	66.75 ± 17.82	⊤	⊤	69.56 ± 8.29	66.25 ± 11.28	65.02 ± 23.70
		XML009	1.47 ± 0.72	2.03 ± 0.92	2.01 ± 0.80	5.89 ± 2.55	6.37 ± 2.64	6.17 ± 2.18	2.30 ± 1.27	2.70 ± 1.53	66.68 ± 9.16	⊤	1.11 ± 0.40	0.93 ± 0.46	0.64 ± 0.21
		XML012	⊤	⊤	65.92 ± 12.90	65.67 ± 21.48	66.99 ± 17.02	⊤	65.99 ± 20.39	⊤	⊤	⊤	⊤	⊤	67.14 ± 14.06
		XML017	0.03 ± 0.02	0.05 ± 0.05	0.04 ± 0.03	0.07 ± 0.07	0.06 ± 0.04	0.02 ± 0.02	0.03 ± 0.02	0.02 ± 0.02	0.01 ± 0.02	0.13 ± 0.09	0.04 ± 0.03	0.03 ± 0.02	0.03 ± 0.02
lua	lua	LUA002	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	67.10 ± 6.58	69.76 ± 7.61	71.10 ± 3.04
		LUA004	5.68 ± 2.17	8.15 ± 2.27	5.75 ± 2.87	14.95 ± 5.97	36.47 ± 20.63	35.36 ± 9.31	5.89 ± 3.57	10.19 ± 4.25	9.93 ± 4.11	7.21 ± 17.28	9.69 ± 2.90	6.24 ± 2.08	10.03 ± 2.58
openssl	asn1	SSL001	35.11 ± 12.55	25.39 ± 7.22	28.46 ± 9.54	44.71 ± 11.97	47.63 ± 13.78	8.58 ± 3.50	19.74 ± 6.45	38.69 ± 9.26	66.85 ± 17.47	⊤	5.72 ± 2.27	5.45 ± 2.84	6.53 ± 3.68
		SSL003	0.06 ± 0.07	0.06 ± 0.06	0.06 ± 0.06	0.06 ± 0.06	0.06 ± 0.06	0.06 ± 0.05	0.06 ± 0.05	0.06 ± 0.05	0.16 ± 0.00	0.26 ± 0.00	0.06 ± 0.04	0.07 ± 0.08	0.07 ± 0.07
	client	SSL002	0.08 ± 0.06	0.17 ± 0.20	0.07 ± 0.05	0.08 ± 0.06	0.08 ± 0.06	0.08 ± 0.05	0.07 ± 0.05	0.08 ± 0.05	0.17 ± 0.00	50.42 ± 37.31	0.09 ± 0.08	0.08 ± 0.06	0.09 ± 0.06
	server	SSL002	0.11 ± 0.08	0.11 ± 0.08	0.12 ± 0.08	0.16 ± 0.09	0.11 ± 0.08	0.12 ± 0.08	0.16 ± 0.09	0.11 ± 0.08	0.22 ± 0.00	0.35 ± 0.00	0.11 ± 0.08	0.11 ± 0.08	0.12 ± 0.09
		SSL020	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	18.62 ± 4.02	16.42 ± 3.27	29.93 ± 16.92	37.10 ± 14.20	46.80 ± 16.06
	x509	SSL009	⊤	71.49 ± 1.74	66.82 ± 17.60	⊤	⊤	64.89 ± 12.55	⊤	54.42 ± 19.80	⊤	27.31 ± 17.28	⊤	⊤	⊤
php	exif	PHP004	57.62 ± 28.19	70.00 ± 6.80	49.60 ± 23.07	57.61 ± 28.20	⊤	48.32 ± 16.34	65.14 ± 23.29	51.48 ± 27.52	✗	2.77 ± 0.06	5.61 ± 3.11	5.48 ± 5.15	2.88 ± 2.54
		PHP009	56.61 ± 17.72	30.29 ± 17.40	49.65 ± 24.01	68.83 ± 8.99	61.50 ± 14.04	15.25 ± 7.36	27.63 ± 19.74	33.01 ± 20.78	✗	3.51 ± 0.22	1.22 ± 0.76	0.64 ± 0.28	0.98 ± 0.57
		PHP011	2.55 ± 1.37	1.67 ± 1.89	3.16 ± 2.88	1.54 ± 1.14	3.80 ± 3.16	0.70 ± 0.41	1.42 ± 1.03	1.11 ± 0.94	✗	2.23 ± 0.03	0.13 ± 0.06	0.21 ± 0.07	0.22 ± 0.09
sqlite3	sqlite3_fuzz	SQL002	1.28 ± 0.50	2.28 ± 0.88	2.62 ± 1.98	9.57 ± 2.10	3.56 ± 0.99	3.70 ± 1.32	1.31 ± 0.63	1.21 ± 0.41	62.10 ± 19.45	⊤	2.83 ± 1.26	5.19 ± 1.63	2.77 ± 1.09
		SQL003	⊤	68.65 ± 11.38	⊤	68.44 ± 12.09	66.47 ± 18.78	⊤	⊤	⊤	⊤	⊤	⊤	69.81 ± 7.45	71.67 ± 1.13
		SQL010	⊤	⊤	⊤	⊤	68.12 ± 13.19	⊤	70.78 ± 4.15	⊤	⊤	⊤	66.87 ± 17.42	⊤	⊤
		SQL012	48.45 ± 14.52	56.60 ± 10.60	63.50 ± 13.57	⊤	54.90 ± 20.57	⊤	61.02 ± 9.13	60.32 ± 13.18	⊤	⊤	67.25 ± 9.35	63.18 ± 15.02	54.53 ± 23.44
		SQL013	⊤	67.15 ± 8.35	69.68 ± 7.89	⊤	69.31 ± 7.06	⊤	⊤	⊤	⊤	⊤	71.16 ± 2.86	67.38 ± 9.07	62.88 ± 13.30
		SQL014	8.63 ± 4.36	8.64 ± 2.56	17.78 ± 6.82	44.40 ± 13.27	18.42 ± 9.06	17.90 ± 9.91	19.91 ± 11.24	30.75 ± 10.16	⊤	⊤	13.94 ± 4.39	29.72 ± 10.17	15.60 ± 7.58
		SQL015	70.67 ± 4.50	64.43 ± 14.97	67.36 ± 15.75	⊤	57.17 ± 22.20	⊤	66.12 ± 12.17	64.72 ± 14.34	⊤	⊤	⊤	69.17 ± 9.61	66.67 ± 14.13
		SQL018	4.60 ± 1.56	3.98 ± 1.64	8.58 ± 4.84	19.84 ± 10.26	4.72 ± 1.11	12.69 ± 4.12	3.40 ± 1.66	3.90 ± 1.99	⊤	⊤	5.64 ± 2.30	5.41 ± 1.50	6.21 ± 1.59
		SQL020	42.36 ± 12.23	46.39 ± 14.82	60.29 ± 15.71	69.81 ± 7.45	40.07 ± 14.72	55.64 ± 21.97	55.97 ± 18.59	67.64 ± 7.93	⊤	⊤	61.24 ± 21.57	59.17 ± 15.05	64.01 ± 11.64

Table 5: Magma bugs (cont.).

Target	Driver	Bug	Fuzzer												
			AFL++								K-Sched	Tortoise	T-Scheduler		
			EXPLORE	FAST	COE	QUAD	LIN	EXPLOIT	MMOPT	RARE			RARE$^-$	RARE$^+$	SAMPLE
poppler	`pdf_fuzzer`	PDF001	⊤	65.08 ± 23.48	⊤	⊤	⊤	⊤	⊤	⊤	✗	⊤	⊤	⊤	⊤
		PDF010	1.15 ± 0.53	1.82 ± 0.50	1.89 ± 1.34	5.25 ± 2.81	5.63 ± 2.40	2.07 ± 2.03	1.96 ± 1.22	1.61 ± 0.56	✗	0.10 ± 0.10	0.99 ± 0.47	1.23 ± 0.52	1.24 ± 0.69
		PDF011	65.59 ± 21.76	⊤	66.53 ± 18.57	60.79 ± 21.97	⊤	65.88 ± 20.79	⊤	⊤	✗	⊤	67.01 ± 12.96	65.70 ± 21.39	55.79 ± 21.95
		PDF016	0.04 ± 0.02	0.05 ± 0.03	0.06 ± 0.04	0.07 ± 0.09	0.03 ± 0.02	0.04 ± 0.02	0.04 ± 0.02	0.07 ± 0.04	✗	0.25 ± 0.00	0.04 ± 0.02	0.04 ± 0.02	0.05 ± 0.03
		PDF018	37.84 ± 22.46	40.38 ± 20.71	38.25 ± 19.84	⊤	⊤	33.83 ± 13.80	29.91 ± 16.76	20.92 ± 12.37	✗	⊤	12.75 ± 6.18	9.40 ± 4.68	10.99 ± 5.44
		PDF019	⊤	⊤	⊤	⊤	69.39 ± 8.85	62.62 ± 21.37	⊤	⊤	✗	⊤	⊤	⊤	⊤
		PDF021	52.56 ± 19.39	⊤	⊤	62.32 ± 13.10	55.67 ± 18.47	⊤	60.34 ± 23.04	65.11 ± 23.38	✗	⊤	70.08 ± 6.50	68.57 ± 11.63	68.76 ± 10.99
	`pdfimages`	PDF002	⊤	⊤	65.84 ± 20.92	⊤	⊤	⊤	⊤	⊤	✗	⊤	⊤	65.56 ± 21.87	65.57 ± 21.84
		PDF003	10.42 ± 5.69	11.24 ± 4.53	7.80 ± 2.47	13.40 ± 5.75	9.72 ± 3.65	32.29 ± 18.22	31.47 ± 16.99	31.91 ± 18.48	✗	⊤	23.56 ± 11.40	5.98 ± 2.64	9.75 ± 4.05
		PDF011	67.30 ± 15.96	47.78 ± 23.75	50.65 ± 21.96	64.93 ± 24.01	⊤	70.10 ± 6.46	59.23 ± 18.00	56.30 ± 22.15	✗	48.95 ± 13.93	55.77 ± 22.48	65.02 ± 15.35	35.84 ± 17.81
		PDF016	0.03 ± 0.02	0.01 ± 0.01	0.03 ± 0.02	0.02 ± 0.01	0.03 ± 0.02	0.02 ± 0.01	0.03 ± 0.02	0.02 ± 0.01	✗	0.09 ± 0.06	0.04 ± 0.03	0.03 ± 0.02	0.02 ± 0.02
		PDF018	15.29 ± 9.90	10.03 ± 5.12	12.76 ± 3.98	62.63 ± 14.63	68.55 ± 9.41	17.24 ± 8.60	5.49 ± 3.25	7.89 ± 8.87	✗	⊤	4.86 ± 1.36	5.23 ± 1.38	3.85 ± 1.57
		PDF019	59.02 ± 25.54	46.57 ± 21.60	59.70 ± 24.13	64.94 ± 23.96	⊤	65.11 ± 23.39	65.89 ± 9.77	67.23 ± 10.93	✗	⊤	59.00 ± 25.48	59.37 ± 24.76	⊤
		PDF021	68.11 ± 7.83	56.31 ± 22.81	57.63 ± 20.14	53.10 ± 19.80	64.80 ± 11.22	60.48 ± 17.74	60.53 ± 16.80	⊤	✗	⊤	⊤	⊤	⊤
	`pdftoppm`	PDF002	⊤	69.18 ± 9.57	⊤	⊤	⊤	66.84 ± 17.53	70.95 ± 3.55	⊤	✗	⊤	⊤	⊤	⊤
		PDF004	⊤	⊤	66.15 ± 12.04	⊤	⊤	⊤	⊤	⊤	✗	⊤	⊤	⊤	⊤
		PDF006	37.74 ± 16.73	47.02 ± 17.98	39.42 ± 19.31	⊤	67.36 ± 15.77	62.16 ± 19.31	43.73 ± 15.04	57.99 ± 27.47	✗	⊤	65.15 ± 13.44	68.07 ± 7.54	69.96 ± 6.93
		PDF010	3.21 ± 1.70	2.98 ± 1.53	2.51 ± 0.90	3.79 ± 1.56	4.14 ± 2.63	2.79 ± 1.96	3.01 ± 1.40	2.08 ± 0.82	✗	0.11 ± 0.08	0.87 ± 0.82	0.81 ± 0.41	1.15 ± 0.48
		PDF011	61.79 ± 20.01	⊤	51.66 ± 27.29	68.18 ± 12.97	54.37 ± 24.48	64.07 ± 16.67	59.46 ± 19.30	62.30 ± 19.04	✗	⊤	66.46 ± 18.79	61.80 ± 20.31	55.98 ± 22.97
		PDF016	0.07 ± 0.04	0.03 ± 0.02	0.03 ± 0.02	0.02 ± 0.02	0.03 ± 0.02	0.04 ± 0.02	0.03 ± 0.02	0.03 ± 0.02	✗	0.19 ± 0.00	0.04 ± 0.04	0.07 ± 0.07	0.04 ± 0.03
		PDF018	29.16 ± 14.25	22.78 ± 16.31	21.64 ± 6.97	65.46 ± 22.20	65.66 ± 21.51	61.72 ± 17.43	24.27 ± 12.33	22.05 ± 8.44	✗	⊤	8.02 ± 5.23	7.30 ± 2.37	8.73 ± 2.30
		PDF019	66.98 ± 17.05	⊤	69.24 ± 9.37	⊤	65.84 ± 12.95	⊤	⊤	64.85 ± 24.28	✗	⊤	66.97 ± 17.06	69.87 ± 7.24	⊤
		PDF021	49.11 ± 22.90	48.91 ± 12.70	56.02 ± 16.93	47.02 ± 16.10	64.56 ± 11.22	54.78 ± 20.24	42.11 ± 18.53	66.85 ± 11.22	✗	⊤	52.93 ± 18.80	63.05 ± 13.16	56.91 ± 21.43