EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit

Tai Yue, Pengfei Wang, Yong Tang*, Enze Wang, Bo Yu, Kai Lu, Xu Zhou
National University of Defense Technology
{yuetai17, pfwang, ytang, wangenze18, yubo0615, kailu, zhouxu}@nudt.edu.cn

Abstract

Fuzzing is one of the most effective approaches for identifying security vulnerabilities. As a state-of-the-art coverage-based greybox fuzzer, AFL is a highly effective and widely used technique. However, AFL allocates excessive energy (i.e., the number of test cases generated by the seed) to seeds that exercise the high-frequency paths and can not adaptively adjust the energy allocation, thus wasting a significant amount of energy. Moreover, the current Markov model for modeling coverage-based greybox fuzzing is not profound enough. This paper presents a variant of the Adversarial Multi-Armed Bandit model for modeling AFL's power schedule process. We first explain the challenges in AFL's scheduling algorithm by using the reward probability that generates a test case for discovering a new path. Moreover, we illustrated the three states of the seeds set and developed a unique adaptive scheduling algorithm as well as a probability-based search strategy. These approaches are implemented on top of AFL in an adaptive energy-saving greybox fuzzer called EcoFuzz. EcoFuzz is examined against other six AFL-type tools on 14 real-world subjects over 490 CPU days. According to the results, EcoFuzz could attain 214% of the path coverage of AFL with reducing 32% test cases generation of that of AFL. Besides, EcoFuzz identified 12 vulnerabilities in GNU Binutils and other software. We also extended EcoFuzz to test some IoT devices and found a new vulnerability in the SNMP component.

1 Introduction

Fuzzing is an automated software testing method that is popular and effective for detecting vulnerabilities in software, which was first devised by Barton Miller in 1989 [23, 32]. Since then, fuzzing has been developed rapidly [22]. As one of the most effective techniques, Coverage-based Greybox Fuzzing (CGF) has attracted several researchers' attention [6]. Combined with genetic algorithms, CGF obtains the path coverage generated by the instrumentation tools and uses it to select good seeds. This technique helps the fuzzing to proceed in a direction that constantly improves the coverage, and more coverage being achieved leads to more bugs for triggering [9]. As Miller's report, a 1% increase in code coverage increases the percentage of bugs found by 0.92% [24].

One of the most popular and widely-adopted CGF is American Fuzzy Lop (AFL) [40]. AFL is an efficient method for file application fuzzing and has identified numerous high-impact vulnerabilities [39]. However, when AFL was used to fuzz real-world programs, it displayed certain shortcomings. The main challenge is that the majority of the test cases exercise the same few paths, thus causing a significant amount of energy wasted on the high-frequency paths [6]. Especially in the later stages of fuzzing, the seeds that exercise high-frequency paths can no longer help in improving the discovery of new paths. However, AFL's constant power schedule is unable to allocate energy to the seeds reasonably. Typically, AFL assigns too much energy to the seeds exercising high-frequency paths. Such problems reflect the insufficient performance of AFL's schedule algorithm. More importantly, the schedule algorithm of AFL is not built on a scientific theoretical model.

Some methods and techniques have been proposed to increase the performance of scheduling algorithms. AFLFast modeled the *transition probability* of mutating a seed for generating a test case exercising another path with the transition probability in a Markov chain [6]. Then, AFLFast implemented a monotonous power schedule to assign energy [6]. This can rapidly approach the minimum energy required for discovering a new path. However, AFLFast cannot flexibly adjust the allocation strategy according to the fuzz process, thereby increasing the average energy cost of discovering a new path. Besides, though AFLFast proposed the transition probability in fuzzing and determined the method for assigning energy as per the transition probability [6], it was unable to provide a detailed analysis of the transition probability. It is not possible to calculate the transition probability from a discovered path to an undiscovered path. In fact, in this con-

*Corresponding author

text, selecting the next seed and assigning energy to the seed is the classic "exploration vs. exploitation" trade-off problem from game theory, not a simple probability problem.

This paper proposes a *variant of the Adversarial Multi-Armed Bandit* (VAMAB) model to model CGF. We modeled each seed as a "bandit" of VAMAB, which is a classical concept from MAB, and explained the trade-off between exploration and exploitation in CGF as per the VAMAB model. Moreover, the Markov chain was used for understanding the details from a probabilistic perspective. As opposed to AFLFast [6], our model's perspective for regarding the process of power schedules is derived from game theory, which helps in better understanding the challenges in schedule algorithm compared to the Markov chain. Further, an *adaptive average-cost-based power schedule* algorithm as well as a *self-transition-based probability estimation method* were developed according to the VAMAB model and were implemented on AFL in a tool named EcoFuzz, which is an *adaptive energy-saving greybox fuzzer*. Compared to AFL's constant schedule and AFLFast's monotonous schedule, EcoFuzz implements an adaptive schedule that can effectively reduce energy wastage, which maximizes the path coverage in the finite times of executions. EcoFuzz is particularly well-suited in situations that have limited performance, such as fuzzing the IoT devices and fuzzing the binary programs via QEMU. In this paper, EcoFuzz was evaluated with six state-of-the-art AFL-type fuzzers such as AFLFast, FairFuzz and MOPT on 14 real-world software [6,17,21]. We also compared EcoFuzz with other four tools like Angora on LAVA-M [10,12]. The following are the contributions made in this paper.

- An Variant of the Adversarial Multi-Armed Bandit (VAMAB). We proposed a VAMAB model to model the CGF, as well as proposed the *reward probability* which is the probability of the seed to discover new paths. We presented the variations of reward probability in detail and introduced the attenuation of this probability. Further, we explained AFL's challenges, classified CGF into three states, and put forth strategies that could enhance AFL's performance in each state.
- Self-transition-based Probability Estimation Method (SPEM). We designed a method to estimate the reward probability for selecting seeds in the exploitation state. This method is more accurate than AFL's search strategy for selecting the next seed with a high reward probability.
- Adaptive Average-Cost-based Power Schedule (AAPS). We recommended an adaptive power schedule that assigns energy to each seed by utilizing the average-cost as the baseline, and then monotonously increases the energy. Compared to AFLFast, AAPS can adjust the next energy allocation by assessing previous allocations.
- Tool. We implement our approaches on AFL, an adaptive energy-saving fuzzer named EcoFuzz. EcoFuzz was then assessed as per 14 real-world software and LAVA-M compared to certain state-of-the-art tools. Results showed that EcoFuzz could find more paths compared to other AFL-type fuzzers with the same number of executions. Moreover, EcoFuzz detected more bugs than others on LAVA-M, and found 12 vulnerabilities in some software, obtaining 2 CVEs. EcoFuzz was also adopted for testing the SNMP component and found a vulnerability. We have published EcoFuzz on Github (https://github.com/MoonLight-SteinsGate/EcoFuzz).

2 Background

2.1 American Fuzzy Lop

As a state-of-the-art CGF, AFL is favored by numerous researchers [6,13,17,43]. AFL uses lightweight instrumentation to capture basic block transitions and determine a unique identifier for the path exercised by a test case, and employs genetic algorithms to discover test cases that are likely to trigger new paths [42]. Its efficiency is affected by some factors.

Search strategy for seeds. AFL keeps a seed queue, dequeues seeds one by one, and fuzzes them. AFL marks some seeds as favored seeds and gives these seeds preference over the non-favored ones [26]. In detail, AFL determines a seed as a favored seed according to the *fav factor* calculated by the seed's execution time and length.

Mutation strategies and power schedules. AFL has two categories of mutation strategies, which are deterministic and indeterministic [42]. The deterministic strategies operate at every bit/byte of each input. And they are only used when it is the first time for fuzzing the seed. In deterministic strategies, AFL assigns energy to the seed according to its length.

After implementing deterministic strategies, AFL effectuates the indeterministic strategies, including *havoc* and *splice*. In this stage, AFL mutates the seed by randomly selecting a sequence of mutation operators and applies them to random locations in the seed file. AFL assigns energy to the seed according to its *score*, which is based on coverage (prioritize inputs that cover more of the program), execution time (prioritize inputs that execute faster), and discovery time (prioritize inputs discovered later) [15]. Particularly, if the test case exercises a new path, AFL will double the assigned energy.

Numerous researchers prefer AFL as its high speed of mutation and execution. AFL also supports source code instrumentation as well as binary instrumentation via QEMU [4], thus making AFL easy to start. However, its performance can be further enhanced. AFL is unable to adjust its energy allocation adaptively and constantly assigns more than the minimum energy required to discover a new path on some seeds, resulting in significant energy wastage [6]. Additionally, AFL has a simple search strategy that is inefficient, leading to AFL taking more turns to select valuable seeds. Finally, the deterministic strategies are also not as effective as random strategies [41].

2.2 Coverage-based Greybox Fuzzing as Markov Chain

Böhme *et al.* [6] modeled CGF as a systematic exploration of the state space of a Markov chain. More importantly, they proposed the transition probability in CGF and modeled it as that in the Markov chain [25].

A Markov chain is a stochastic process that transitions from one state to another. Formally, a Markov chain refers to a sequence of random variables $\{X_0, X_1, ..., X_n\}$ where X_i denotes the state of the process at time i. The value of X_i is taken from a set of states $S = \{1, 2, ..., N\}$ for some $N \in \mathbb{N}$. Further, the transition probability p_{ij} indicates the chain's state transition probability from state i at time t to state j at time $t+1$, which is signified as the conditional probability,

$$p_{ij} = P(X_{t+1} = j | X_t = i) \tag{1}$$

Particularly, if the transition probability p_{ij} depends only on the state i and j, and not on the time t, the Markov chain is called time-homogeneous. To model CGF as a time-homogeneous Markov chain, Böhme *et al.* defined the Markov chain's state space as the discovered paths and their immediate neighbors [6]. That is, given a set of seeds T, S^+ indicates the set of discovered paths that are exercised by T while S^- is the set of undiscovered paths [6] that are exercised by inputs generated by randomly mutating any seed from T. The set of states S is defined as

$$S = S^+ \cup S^- \tag{2}$$

The transition probability is defined as follows. For path $i \in S^+$, p_{ij} is the probability of generating a test case exercising the path j through the mutation of the seeds $t_i \in T$ that exercises the path i.

According to this model, Böhme *et al.* [6] proposed that a more efficient CGF can discover an undiscovered state in a low-density region while assigning the least amount of total energy. That is, defining $E[X_{ij}]$ is the expectation of the minimum energy that should be assigned to seed $t_i \in T$ for discovering the new state j, CGF must choose t_i for fuzzing such that $\exists j \in S^-$ where the probability of executing path j is low and $E[X_{ij}]$ is minimal. Moreover, the energy assigned to t_i should be $E[X_{ij}]$, which is deduced as $1/p_{ij}$ in [6].

Unfortunately, when fuzzing real-world programs, it is impossible to calculate the transition probability of discovering a new path from the current seed precisely, and thus, a completely accurate approach cannot be determined for selecting the next seed and assigning energy to it. However, there is a seed $t_i \in T$ that has the highest probability of finding a new path. AFLFast [6] recommended selecting the next favored seed that is chosen from the queue with the smallest number of times and that exercises a path with the least amount of fuzz. However, the efficiency of this search strategy depends on the information about all seeds. If there is a queue of seeds Q where some seeds from Q have been fuzzed while others are not, there may be more accurate recognition for seeds that have been fuzzed than those that have not. For choosing the next seed t_i where the probability of executing path i is the minimum, it is necessary to conduct an examination for fuzzing seeds that have not been fuzzed, which is a classic "exploration vs. exploitation" trade-off problem.

2.3 Multi-Armed Bandits Problem

The Multi-Armed Bandit problem is important as one of the simplest non-trivial problems wherein the conflict between exploitation and exploration [7, 35]. This problem resulted from the slot machine with multiple *arms*. In this case, the player plays one of the arms and obtains a *reward*. The player's main goal is maximizing the rewards in finite trials [35].

Formally, as shown in Fig. 1, there are N parallel arms, indexed $i \in K = \{1, 2, ..., N\}$, and each time only a single arm is allowed to be selected to play. The state of arm i at time t is denoted as $x_i(t)$, while the expectation of reward of the arm i at time t is $R_i(x_i(t))$ [35]. However, there is no indication about the reward expectations related to each arm. Thus, the problem is how to allocate the trials over arms sequentially in time to maximize the expected total reward. It should be noted that an increasing number of trails being allocated to an arm i will lead to more accurate information being deduced regarding the reward expectation of i, which is the process of *exploration*. If all the reward expectations of all arms are known, then we only select those arms with the highest expectation to gain the highest reward, which is the process of *exploitation*. Therefore, our goal is achieved by having a trade-off between exploration (trying out some arms) and exploitation (choosing an arm with the highest reward). Exploitation helps maximize the expected rewards for a single step, whereas the combination of exploration and exploitation helps achieve higher rewards in the long run [26].

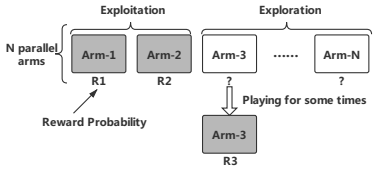

Figure 1: The schematic diagram of the MAB problem, where the grey color block symbolizes that this arm has been played for some times.

In the classic MAB problem, there are two assumptions that the distribution of rewards for each arm is time-invariant, and the number of arms is constant. Thus, solutions concerning the MAB problem have almost relied on these assumptions [2]. However, these assumptions limit the MAB model's applicability. For modeling CGF as the MAB-type model, it is natural to regard an arm as a seed. However, during fuzzing, the number of seeds (i.e., arms) is increasing and the probability of

finding a new path (i.e., reward probability) is decreasing, which are not constant. Particularly, Auer *et al.* proposed the MAB problem variant that includes no-statistical assumptions about generating rewards as the Adversarial Multi-Armed Bandit (AMAB) problem [3]. We consider modeling CGF by the variant of the AMAB model, not the MAB model.

3 A Variant of the Adversarial Multi-Armed Bandit Model

In this section, we model the process of searching seeds and assigning energy as a variant of the AMAB problem, thus enabling exposing the essence of the CGF. Moreover, we explain the exploration and exploitation during fuzzing according to this model, and point out certain challenges in enhancing AFL.

3.1 Coverage-based Greybox Fuzzing as the Variant of the Adversarial Multi-Armed Bandit Model

In this subsection, we define some assumptions and terms, then build our VAMAB model. Assuming that we are fuzzing program A, several assumptions are stated below.

Assumption 3.1 *The number of total paths and unique crashes that can be executed of program A are finite, denoted as n_p and n_c, respectively.*

This assumption helps to consider the mathematical model in the finite state space, which could simplify the problem.

Assumption 3.2 *The program A is stateless. That is, the path of each execution depends only on the input generated by fuzzer.*

This assumption ensures that the reward probability is independent in VAMAB model, only determined by the seed.

The following are some important definitions.

Definition 3.1 *The set of total paths of program A is signified as $S = \{1, 2, ..., n_p\}$ and the corresponding seeds set is denoted as $T = \{t_1, t_2, ..., t_{n_p}\}$.*

Definition 3.2 *We followed the definitions of **transition probability** p_{ij} and the **minimum energy** $E[X_{ij}]$ in [6]. p_{ij} is the probability of generating a test case exercising path j from the seed t_i. $E[X_{ij}]$ is the expectation of minimum energy (i.e., the number of test cases generated by t_i) of this process, deduced as $1/p_{ij}$ in [6].*

Definition 3.3 *Based on Definition 3.2, we define the **transition frequency** f_{ij} as the frequency of path transition from path i to path j, as*

$$f_{ij} = \frac{f_i(j)}{s(i)} \tag{3}$$

*$f_i(j)$ indicates the number of test cases exercising path j generated by seed t_i. Particularly, f_{ii} is defined as the **self-transition frequency**. $s(i)$ is the number of trials conducted to seed t_i, satisfying*

$$s(i) = \sum_{j=1}^{n_p} f_i(j) \tag{4}$$

Definition 3.4 *We define the probability of mutating t_i for generating inputs executing other paths as p_{i*}, deduced as*

$$p_{i*} = 1 - p_{ii} = \sum_{j=1}^{n_p} p_{ij} - p_{ii} = \sum_{j=1, j\neq i}^{n_p} p_{ij} \tag{5}$$

Providing the queue with n seeds is T_n, $|T_n| = n$, $1 \leq n < n_p$, some of the seeds in T_n that have been fuzzed are denoted as T_n^+ and the others are marked as T_n^-. Additionally, the number of trials being conducted thus far is m.

When fuzzing the program A, the aim might be maximizing the number of discovered crashes and paths of A as well as assuming them as the arms in the MAB model. However, Woo *et al.* [36] pointed out that focusing on one seed may trigger the same crashes, thus impacting the selection in exploitation. Thus, **our model regards the seeds as the arms and aims to maximize path coverage in finite trials**. Therefore, we define the **reward** of each trial as generating an input that triggers new path. Each trial to play an arm i denotes mutating a corresponding seed t_i and executing the generated test case.

Now we have conducted the trials for m times. $\forall t_i \in T_n$, we denote earn a reward in next trial as,

$$R_i(m+1, T_n) = 1 \tag{6}$$

The probability of the arm i to earn a reward (i.e., discovering a new path) in this trial is deduced as

$$\begin{aligned} P(R_i(m+1, T_n) = 1) &= \sum_{j=n+1}^{n_p} p_{ij} \\ &= 1 - \sum_{j=1}^{n} p_{ij} \end{aligned} \tag{7}$$

We define this probability as the **reward probability**. According to Equation (7), we can deduce that: (1) the reward probability $P(R_i(m+1, T_n) = 1)$ depends only on the seed t_i and the seeds set T_n of discovered paths, and is not related to the number of trials being conducted (i.e., m). Thus, the reward probability is simplified as $P_{R_{i,n}}$; (2) with a rise in the number of discovered seeds n, there is a decrease in the number of undiscovered paths $(n_p - n)$ which leads to a reduction in the probability of arm i to find new paths. These are following the general results in most evaluation that as more paths are found, the discovery of new paths decelerates monotonically [6].

Therefore, it is evident that the distribution of the reward of each arm is not invariant. Actually, the probability decreases

once a reward is gained in some trials. This is called *probability attenuation*. As a result, the process of fuzzing is not modeled as the classic MAB model, which is closer to the AMAB model. Moreover, according to the mechanism of CGF, once a reward is earned, it leads to a new and interesting path. New seed will also be added into the queue of seeds, with the seeds set T_n transferring into T_{n+1} and the number of arms increasing to $n+1$, as shown in Fig. 2. Based on these differences, this problem is defined as a VAMAB.

As opposed to the traditional MAB model, the number of arms of the VAMAB model will increase, and the reward probability will decrease if rewards are earned until all paths of program A are found. Therefore, before discovering all paths, there is always a trade-off between exploration (fuzzing seeds that have been not fuzzed) and exploitation (selecting the fuzzed seeds to get more rewards).

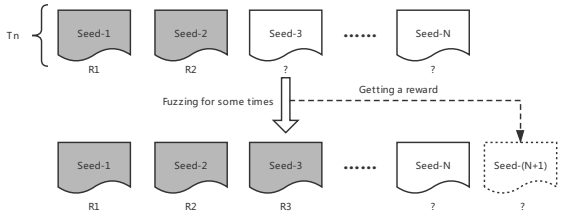

Figure 2: The figure illustrating VAMAB model , in which the grey color block symbolizes that this seed has been fuzzed.

3.2 Exploration vs Exploitation in VAMAB Model

Providing we could calculate the reward probability of seeds after conducting some trials on them, for the seeds set T_n, we can determine the reward probability $P_{R_{i,n}}$ of the seed t_i from T_n^+, which is the set of fuzzed seeds. Then we can calculate the minimum energy the seed requires to find new paths following Definition 3.2. For gaining more rewards in a short period, it may be better to select the seeds from T_n^+ with the highest reward probability, as "exploitation". In contrast, focusing on the unfuzzed seeds in T_n^- and allocating them enough energy can help to calculate their reward probability. Seeds with higher reward probability may be found from T_n^- compared to those from T_n^+, as "exploration".

Thus, based on the level of testing on the seeds, as shown in Fig. 3, the states of T_n were classified into three categories:

(1) **Initial State**. The initial state refers to the first stage of the fuzzing process, where all seeds are unfuzzed. After beginning the fuzzing of the seeds, the initial state transitions to the exploration or exploitation state, as indicated by Curve 1 and Curve 2 in Fig. 3.

(2) **Exploration State**. In this state, some seeds in T_n are fuzzed, while some are not. Therefore, energy should be assigned to the seeds that have not been fuzzed to earn rewards and estimate their reward probability. After attaining a reward, T_n transits to T_{n+1}. Once all seeds in T_m are fuzzed, the exploration state transitions into the exploitation state, as shown by Curve 3 in Fig. 3.

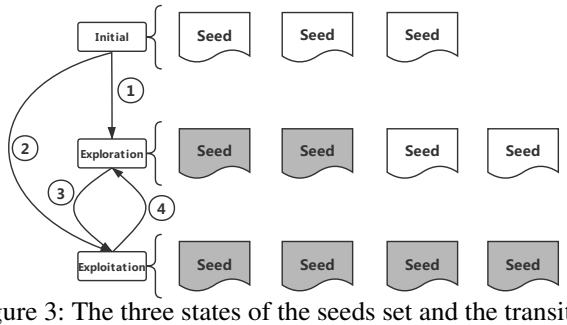

Figure 3: The three states of the seeds set and the transition relationship between them, in which the grey color block symbolizes that this seed has been fuzzed.

(3) **Exploitation State**. In this state, all seeds have been fuzzed. It is crucial to select those seeds with the highest reward probability to test for discovering new paths. Once a test case exercises an undiscovered path, the transition from the exploitation to exploration occurs until all paths have been found, as shown by Curve 4 in Fig. 3.

For these three states, it is necessary to implement different strategies to maximize rewards. As previously discussed, it is risky to focus only on exploitation and skip exploration. Therefore, we considered the strategy of testing each seed in the initial and exploration stage and selecting the high-quality seeds with high reward probabilities in the exploitation stage.

3.3 Challenges in VAMAB Model

Although we have proposed how to improve the efficiency of the scheduling algorithm, some challenges persisted.

The first challenge is **how to determine the reward probability of each seed to select the next seed in the exploitation stage**. Given $t_i \in T_n$, its reward probability $P_{R_{i,n}}$ is certain. According to Equation (7), the reward probability depends on transition probability. In [6], Böhme *et al.* calculated the transition probability between seeds in an example. However, determining the transition probability p_{ij} relies on the path constraints of path i and j, which can only be inferred through manual analysis with source code, not accessed by CGF. Therefore, we could not accurately calculate the reward probability of seeds despite conducting several trials on the seeds. We can only estimate it. A common method is to estimate the transition probability through transition frequency. That is, for p_{ij}, it is possible to approximate it as f_{ij} for $1 \le i, j \le n$. However, based on Equation (3), (4) and (7), we may estimate the reward probability $P_{R_{i,n}}$ as

$$P_{R_{i,n}} \approx 1 - \sum_{j=1}^{n} f_{ij} = 1 - \sum_{j=1}^{n} \frac{f_i(j)}{s(i)} = 0 \tag{8}$$

This is useless for CGF to select seeds. Consequently, it is important to find other criteria or parameters for approximating the reward probability to select the seeds to fuzz.

The second challenge pertains to **how to assign suitable energy to each arm to balance the trade-off between exploration and exploitation**. Especially in the exploration stage, assigning too much energy to an unfuzzed seed in T_n^- is very risky. Researchers proposed some algorithms for resolving the problem of trade-off in the Adversarial MAB problem (e.g., Exp3) [3]. However, this algorithm is based on the assumption that the number of arms is constant. Our model differs from the traditional AMAB problem on the variability of the number of arms. Therefore, some current algorithms are not suitable for our model.

Therefore, to maximize the path coverage, we need to establish efficient mechanisms, which use existing information to estimate the reward probability of each seed for searching seeds in the exploitation stage and allocate appropriate energy to seeds for reducing energy waste.

4 Implementation

In this section, we implemented a prototype tool called EcoFuzz. We introduce the framework and algorithm of EcoFuzz firstly. After that, we detail the search strategy and energy schedule algorithm implemented in EcoFuzz.

4.1 Main Framework of EcoFuzz

EcoFuzz is based on AFL 2.52b, which follows the framework and most of the mechanisms of AFL, including the feedback-driven coverage and crash-filter mechanisms. Based on these, we developed a scheduling algorithm called AAPS and a search strategy called SPEM. The state determination mechanism was added. EcoFuzz is based on the VAMAB model to determine which state the seeds queue stays at. Moreover, EcoFuzz runs without the deterministic strategies, while our algorithm eliminated the mechanism in AFL that doubling energy when a new path is found. Fig. 4 presents an overview of EcoFuzz. Further details are given in Algorithm 1. The three states of EcoFuzz are introduced below:

Initial State. EcoFuzz only stays at this state before fuzzing. In this state, EcoFuzz chooses the first seed to fuzz. Then, EcoFuzz turns to the exploration or exploitation state.

Exploration State. In this state, EcoFuzz selects the next seed based on the index order of the seeds which are not fuzzed, without skipping the seeds that are not preferred, and assigns energy by AAPS. If all seeds in the queue have been fuzzed, EcoFuzz transfers into the exploitation state.

Exploitation State. In this state, as all seeds have been fuzzed, EcoFuzz implements SPEM for estimating the reward probability of all seeds and prioritizes the seeds with high reward probability for testing. Each seed is selected at most once until all seeds have been selected or a new path is found.

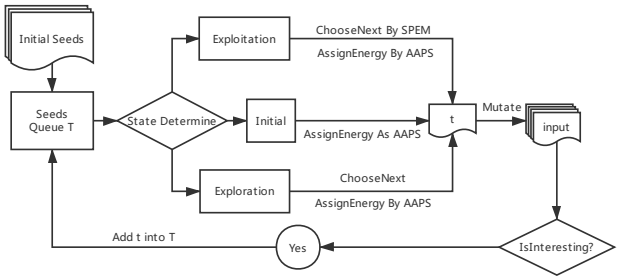

Figure 4: The overview of EcoFuzz, where the SPEM and AAPS denote the search strategy and energy schedule we propose in Section 4.2 and Section 4.3, respectively.

If all seeds have been selected in this state, EcoFuzz will re-select the seeds until finding paths. After a new path is found, EcoFuzz transfers from exploitation to exploration.

Algorithm 1 The algorithm of EcoFuzz

Require: Initial Seeds Set S
  $total\_fuzz = 0$
  $rate = 1$
  $Q = S$
  repeat
    $queued\_path = |Q|$
    $average\_cost$ = CalculateCost($total\_fuzz$, $queued\_path$)
    $state$ = StateDetermine(Q)
    if $state$ == Exploitation **then**
      s = ChooseNextBySPEM(Q)
    else
      s = ChooseNext(Q)
    end if
    $Energy$ = AssignEnergy(s, $state$, $rate$, $average\_cost$)
    for i from 1 to $Energy$ **do**
      t = Mutate(s, Indeterministic)
      $total\_fuzz$ += 1
      res = Execute(t)
      if res == CRASH or IsInteresting(res) **then**
        $regret = i$ / $Energy$
        $s.last\_found$ += 1
        if IsInteresting(res) **then**
          add t to Q
        else
          add t to T_c
        end if
      end if
    end for
    $rate$ = UpdateRate($regret$, $rate$)
    $s.last\_energy = Energy$
  until timeout reached or abort-signal
Ensure: T_c

Additionally, according to [11], we add a static analysis module for extracting some magic bytes to a dictionary for certain programs. In detail, the static analysis module extracts some hardcode and magic bytes in the target binary by searching from its disassembly information, which is efficient and uncomplicated.

4.2 Self-transition-based Probability Estimation Method

In Section 3, we introduced the reward probability of each seed and proved that it is not possible to determine the reward probability accurately. Fortunately, our model aims to select the seeds with high reward probability in the exploitation state. Therefore, there is a greater focus on the magnitude relationship but not on the specific value of the reward probability.

From Equation (5) (7), we can deduce that

$$P_{R_{i,n}} = p_{i*} - \sum_{j=1, j\neq i}^{n} p_{ij} \tag{9}$$

For $i \in \{1,2,...,n\}$, the probability p_{i*} is constant and $\sum_{j=1,j\neq i}^{n} p_{ij}$ depends only on the set T_n. Based on the discussion in Section 3.3, we considered using $(1-f_{ii})$ as an approximate estimation of p_{i*}. However, for $\sum_{j=1,j\neq i}^{n} p_{ij}$, as it is the reason for probability attenuation, the earlier the seed is discovered, the more its reward probability attenuates. Hence, the index of the seed was used to illustrate the probability attenuation qualitatively. Following is the estimation method:

$$P_{R_{i,n}} \approx 1 - \frac{f_{ii}}{\sqrt{i}} \tag{10}$$

According to Equation (10), our method prefers to select the seeds with lower self-transition frequency and larger index. However, the estimation method is only used to qualitatively estimate the magnitude relationship of the reward probability between the seeds. Thus, we could not calculate the minimum energy of the selected seed. For this, an adaptive average-cost-based power scheduling algorithm was proposed.

4.3 Adaptive Average-Cost-based Power Schedule

As the lowest energy to find a new path can not be calculated, a scheduling algorithm was developed to approximate it monotonically. Compared to AFL, which allocated redundant and constant energy each time, our algorithm aims to be economical and flexible, particularly in the exploration stage.

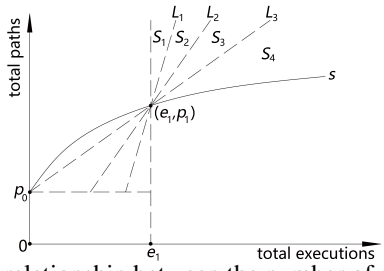

Figure 5: A relationship between the number of paths and the number of total executions during the fuzzing process.

Considering a typical fuzzing process, as shown in Fig. 5, Curve s represents the relationship $p(e)$ between the number of paths p and the number of total executions e when the CGF is fuzzing a target. Further, Fig. 5 shows that the derivative of $p(e)$ decreases with an increase in the number of executions e, meaning that the CGF found new paths more efficiently in an early stage than a later stage. Particularly, the point $(0,p_0)$ denotes the initial state of fuzzing and the point (e_1,p_1) shows that the CGF found (p_1-p_0) unique paths with the e_1 executions. The average-cost of finding a path is defined as

$$C(p_1,e_1,p_0) = \frac{e_1}{p_1-p_0} \tag{11}$$

This represents the average number of executions required for discovering a new path when the CGF has executed e_1 test cases, which is the reciprocal of the slope of Line L_3 in Fig. 5. Notice that, the average-cost decreases with an increase in the executions. Therefore, the next point (e_2,p_1+1) is likely to appear in Area S_4 in Fig. 5. However, if the CGF generates test cases less than $C(p_1,e_1,p_0)$ to find a new path, the next point will appear in Area $S_1 \cup S_2 \cup S_3$, above Line L_3.

It was expected that CGF could find as many new paths within the average-cost of energy as possible. Thus, we considered using the average-cost C as the basic line for allocating energy, which is economical for the CGF, to design the AAPS algorithm, as shown in Algorithm 2.

For the seed s, we allocate energy no more than average-cost to s in the exploration stage. In addition, less energy allocation was considered for the seeds exercising high-frequency paths than those exercising low-frequency path, which is realized by the function CalculateCoefficient(). In detail, we calculate the ratio r of the total number of test cases exercising the same path with s (i.e., $s.exec\_num$) and $average\_cost$. For the ratio r in $(0,0.5]$, $(0.5,1]$ and $(1,+\infty)$, we set the coefficient k as the empirical values: 1, 0.5 and 0.25, respectively, allocated energy $k \times C$ corresponding to the reciprocal of the slope of Line L_3, L_2 and L_1 in Fig. 5.

Algorithm 2 The AAPS algorithm

Require: s, $state$, $rate$, $average\_cost$
  $Energy = 0$
  if $state$ == Exploration **then**
    k = CalculateCoefficient($s.exec\_num$, $average\_cost$)
    $Energy = average\_cost \times k \times rate$
  else if $state$ == Exploitation **then**
    if $s.last\_found > 0$ **then**
      $Energy$ = Min($s.last\_energy$, M) $\times$ $rate$
    else
      $Energy$ = Min($s.last\_energy \times 2$, M) $\times$ $rate$
    end if
  else
    $Energy = 1024 \times rate$
  end if
Ensure: $Energy$

Furthermore, the *regret* concept in certain solutions of the classic MAB problem were combined for establishing

a context-adaptive energy allocation mechanism [1]. This mechanism aims to improve the coefficient of energy utilization. If more energy is allocated than the seed need to find a path, this mechanism reduces energy assigned the next time.

Moreover, the regret is calculated according to the energy assigned to the seed and the energy it uses if it finds new paths. Based on a previous assessment of energy allocations, the coefficient *rate* was updated to adjust the next allocation. Particularly, to avoid wasting too much energy on a seed in the exploitation stage, we set M as the upper bound for one turn of energy allocation and assign the empirical value $16 \times average\_cost$ to M.

5 Evaluation

5.1 Configuration of Evaluation

Real-World Programs. We evaluated EcoFuzz as per 14 real-world utility programs. These programs were selected from those evaluated by other AFL-type tools [17,21]. All the evaluation was conducted without dictionaries. The configuration of all programs is listed in Table 1. For each case, we ran the fuzzing with one seed provided by AFL.

Table 1: The configuration of target programs

Subjects	Version	Format
nm -C @@	Binutils-2.32	elf
objdump -d @@	Binutils-2.32	elf
readelf -a @@	Binutils-2.32	elf
size @@	Binutils-2.32	elf
c++filt @@	Binutils-2.32	elf
djpeg @@	libjpeg-turbo-1.5.3	jpeg
xmllint @@	libxml2-2.9.9	xml
gif2png @@	gif2png-2.5.13	gif
readpng @@	libpng-1.6.37	png
tcpdump -nr @@	tcpdump-4.9.2	pcap
infotocap @@	ncurses-6.1	text
jhead @@	jhead-3.03	jpeg
magick convert @@ /dev/null	ImageMagick-7.0.8-65	png
bsdtar -xf @@ /dev/null	libarchive-3.4.0	tar

Baseline. We compared EcoFuzz against other six AFL-type fuzzers, including AFL, FidgetyAFL, AFLFast, AFLFast.new, FairFuzz and MOPT-AFL [6,17,21,41].

We executed the AFLFast and AFLFast.new with the fast model, which is the fastest schedule strategy of AFLFast [6], and ran MOPT-AFL with the parameter "-L 30" to launch the MOPT scheme.

Platform. We fuzzed each case for 24 hours (on a single core) and repeated each experiment 5 times to reduce the effects of randomness according to [16]. The experiments were conducted on a 64-bit machine with 40 cores (2.8 GHz Intel R Xeon R E5-2680 v2), 64GB of RAM, and Ubuntu 16.04 as server OS. The experiments ran for 490 CPU days.

5.2 Evaluation of Path Exploration and Energy-Saving

Evaluation Metrics. We choose the total number of paths discovered by different techniques, the total number of test cases generated, and the average-cost as the measurements.

The reason is derived from the model design. The VAMAB model aims to maximize the number of paths in the least number of test cases generated. According to the definition of average-cost, our scheduling algorithm uses the average-cost as the basic line for allocating energy and measuring the efficiency of each allocation. Thus, EcoFuzz intended to achieve the same number of paths with other tools in the least number of fuzz, namely, the least average-cost.

Path Coverage. For each subject and technique, Fig. 6 plots the average number of paths discovered throughout five runs at each average number of executions point in 24 hours.

Fig. 6 shows that EcoFuzz outperforms other six AFL-type fuzzers on most programs while achieving the upper bound on the number of paths on `nm`, `objdump`, `size` , `gif2png`, `readpng`, `tcpdump`, `jhead`, `magick` and `bsdtar` in the least executions. The path coverage achieved by EcoFuzz on the other five programs is approximately the same as that of FidgetyAFL or AFLFast.new, and is more than that of FairFuzz and MOPT-AFL. Particularly, except `readelf` and `djpeg`, EcoFuzz finds the most paths with the same executions than other tools. More analysis is detailed in Appendix 8.1.

Average-Cost. As FidgetyAFL, AFLFast.new, and FairFuzz outperform the other three tools in path exploration, we focused on comparing their efficiency with that of EcoFuzz. Table 2 presents the number of total paths, total executions, and the average-cost of these techniques on each subject.

From Table 2, EcoFuzz generates fewer test cases than the other three state-of-the-art tools on eight subjects, and finds more paths than others on nine programs. Moreover, EcoFuzz's average-cost is observed to be significantly lower than that of others on most programs. On `size`, `djpeg` and `gif2png`, though FairFuzz has the lowest average-cost, the number of paths it found is also the least. In contrast, EcoFuzz finds more paths than others on `size` and `gif2png`, with a lower average-cost than that of AFLFast.new and FidgetyAFL. Particularly, on `jhead`, EcoFuzz attained more paths upper bound than other techniques in the early stage with fewer executions. Therefore, EcoFuzz outperforms other tools in energy-saving. More analysis is detailed in Appendix 8.1.

Statistical Analysis. Following the guidance of [16], we conducted statistical analyses to ensure that the evaluation is comprehensive. We used p value and extremum to evaluate the performance of these tools. For p value, p_1 represents the difference between the performances of EcoFuzz and AFL. Further, p_2, p_3, p_4, p_5, and p_6 denote the differences between the performances of EcoFuzz and FidgetyAFL, AFLFast, AFLFast.new, FairFuzz, and MOPT-AFL, respectively. The number of paths and average-cost were considered for calculating the p value. All the results and more analysis are shown in Table 6 and 7 in Appendix 8.1.

From these results, EcoFuzz and AFLFast.new outperform the other five tools significantly in the extremum of discovered paths. On the path coverage, p_1 is smaller than 10^{-4} in

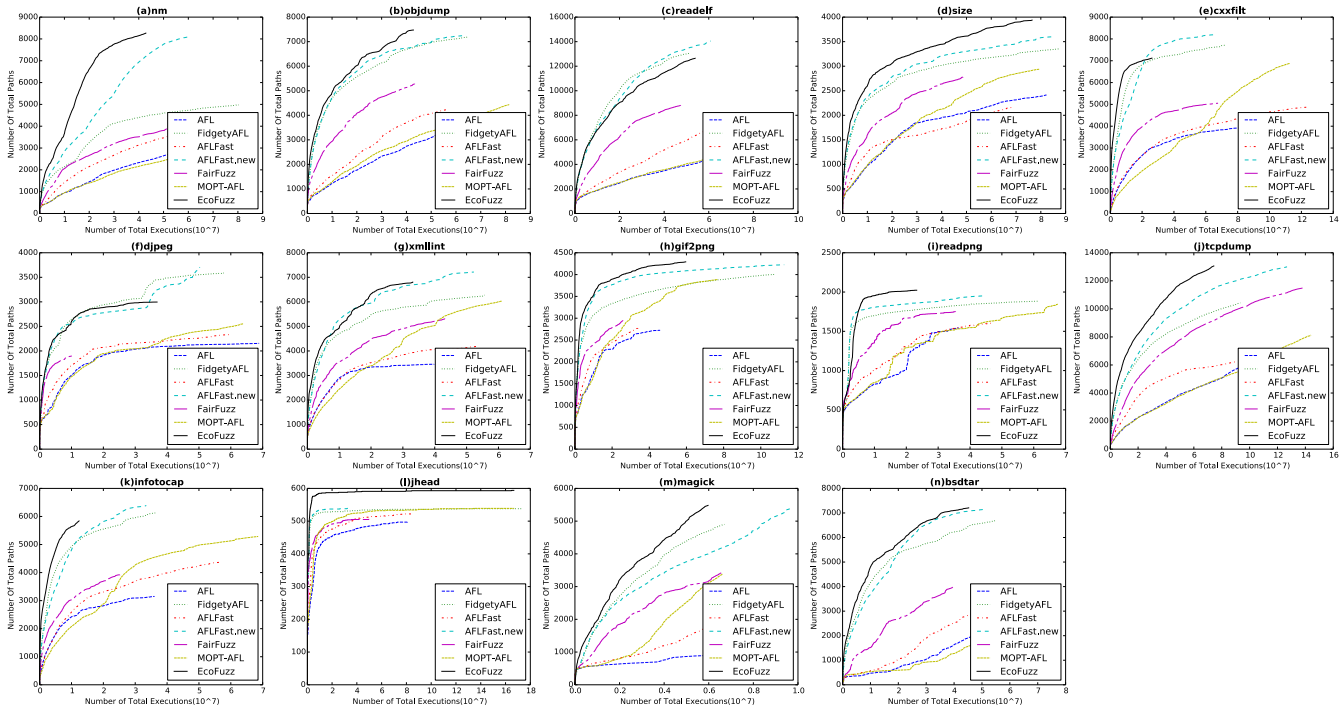

Figure 6: Number of total paths discovered by different AFL techniques averaged over 5 runs, where the X axis represents the number of total executions in 24 hours, which is scaled in units of 10^7.

Table 2: The average-cost of each fuzzer on each subject

Subjects	Number of total paths / Number of executions finding these paths				Average-cost			
	FidgetyAFL	AFLFast.new	FairFuzz	EcoFuzz	FidgetyAFL	AFLFast.new	FairFuzz	EcoFuzz
nm	4,975 / 80.34M	8,127 / 60.95M	3,890 / 51.42M	8,266 / 42.88M	16,152	7,500	13,222	**5,188**
objdump	7,186 / 65.03M	7,241 / 62.45M	5,287 / 43.34M	7,474 / 42.78M	9,051	8,626	8,200	**5,724**
readelf	13,063 / 51.73M	14,048 / 60.90M	8,813 / 47.47M	12,649 / 53.90M	**3,960**	4,335	5,387	4,261
size	3,352 / 87.12M	3,601 / 85.31M	2,782 / 48.90M	3,939 / 76.45M	25,998	23,698	**17,581**	19,412
cxxfilt	7,715 / 72.37M	8,192 / 64.90M	5,054 / 67.59M	7,119 / 26.19M	9,381	7,923	13,377	**3,679**
djeg	3,587 / 57.77M	3,706 / 50.29M	1,902 / 10.45M	2,996 / 36.78M	16,109	13,572	**5,498**	12,280
xmllint	6,269 / 55.69M	7,214 / 52.12M	5,322 / 43.21M	6,803 / 33.11M	8,884	7,225	8,120	**4,868**
gif2png	4,004 / 107.46M	4,226 / 112.38M	2,952 / 25.88M	4,292 / 59.53M	26,844	26,600	**8,769**	13,873
readpng	1,884 / 61.36M	1,952 / 44.39M	1,753 / 35.48M	2,023 / 22.66M	32,585	22,755	20,253	**11,205**
tcpdump	10,432 / 93.37M	12,993 / 126.74M	11,489 / 137.89M	13,059 / 74.27M	8,951	9,755	12,003	**5,688**
infotocap	6,125 / 36.23M	6,389 / 33.47M	3,921 / 25.23M	5,840 / 12.36M	5,917	5,239	6,436	**2,117**
jhead	538 / 120.60M	539 / 32.16M	506 / 49.69M	594 / 164.86M	224,575	**59,775**	98,402	278,005
magick	4,903 / 6.70M	5,375 / 9.63M	3,419 / 6.56M	5,483 / 5.97M	1,367	1,793	1,919	**1,089**
bsdtar	6,685 / 54.84M	7,143 / 51.15M	3,981 / 39.55M	7,209 / 45.17M	8,204	7,162	9,936	**6,266**

* The number of executions finding these paths denotes the number of test cases are generated when the fuzzers have reached these paths, of which the unit is M(10^6). Bold fonts represent the best performance.

all evaluations, indicating that the distribution of total paths found by EcoFuzz and AFL differ significantly. Compared to AFLFast.new, though EcoFuzz achieves the path coverage approximate to AFLFast.new, the energy depletion and average-cost of EcoFuzz are significantly lower than AFLFast.new.

Overall. EcoFuzz performs better than other AFL-type techniques in the average-cost. Moreover, compared to AFL, AFLFast, FairFuzz, and MOPT-AFL, more paths were found by EcoFuzz on tested programs. EcoFuzz finds 214% of the paths discovered by AFL and generates only 68% test cases of AFL, while reducing 65% average-cost of AFL. EcoFuzz also generates only 65% test cases of FidgetyAFL and finds 110% of the paths found by FidgetyAFL, and 65% test cases of AFLFast.new, along with determining the same number of paths. In addition, EcoFuzz reduces the average-cost of approximately 39% of FidgetyAFL and 33% of AFLFast.new.

5.3 Evaluating the Search Strategy and Power Schedule

This subsection focuses on the efficiency of SPEM and AAPS algorithm.

Evaluation Metrics. We define the *utilization ratio* of energy, which is the ratio of the energy consumed for finding the newest path to the total energy allocated in each turn, to evaluate the scheduling algorithms of different techniques.

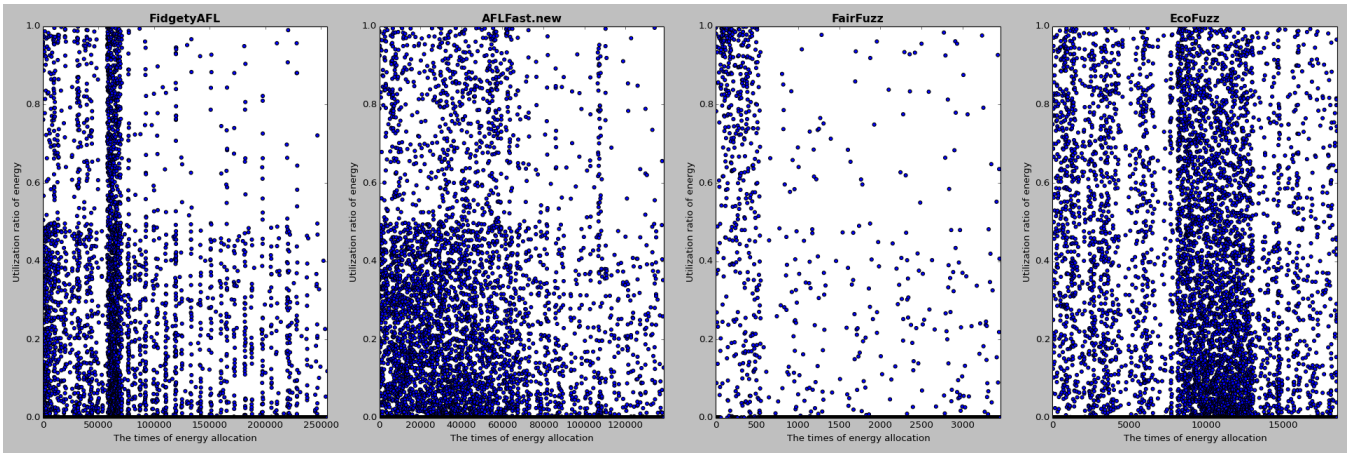

Figure 7: The utilization ratio in each time of allocation as the times of energy distribution during these four fuzzers test the nm.

We recorded the turns of allocation and energy consumed in indeterministic strategies. Because all fuzzers except EcoFuzz implement the splice strategy, and as the mechanism of splice strategy is very similar to that of havoc strategy, each allocation in splice strategy was regarded as a time of energy allocation. Particularly, if the fuzzer did not find new paths in one turn of energy allocation, the ratio was recorded as 0. Thus, the value of ratio ranges from 0 to 1.

Based on the utilization ratio, certain indicators for multi-faceted assessments, including the *average utilization ratio* and the *effective allocation*, were defined. The index of allocation times was denoted as i, ranging from 1 to N, while the corresponding utilization ratio was denoted as r_i. In addition, the number of paths found in this energy allocation is n_i, and the first indicator is *average utilization ratio*, calculated as

$$\bar{r} = \frac{\sum_{i=1}^{i=N} r_i}{N} \tag{12}$$

The frequency p of allocation finding new paths (we call this *effective allocation*) is the second measurement, denoted as

$$p = \frac{|\{i | n_i > 0, 1 \leq i \leq N\}|}{N} \tag{13}$$

We choose each best run of EcoFuzz, FidgetyAFL, FairFuzz, and AFLFast.new on fuzzing `nm` to start our evaluation.

Evaluation of AAPS Algorithm. Fig. 7 plots the utilization ratio in each turn of the energy distribution of these four tools during fuzzing `nm`. The utilization ratio of a point being closer to 1.0 indicates less energy being wasted. Further, the degree of density of points represents the path coverage.

As shown in Fig. 7, EcoFuzz utilizes energy more efficiently than the other three tools, as its distribution of points is closer to 1.0 than others. EcoFuzz also found the most paths among all tools, which was significantly more than that found by FairFuzz and FidgetyAFL, with the densest distribution of points. Further, for the distributions of FidgetyAFL and AFLFast.new, the majority of the points are located in the interval with the ratio being between 0 and 0.5, and only a few points' ratios are higher than 0.5. In contrast, EcoFuzz's distribution of points is much closer to 1.0 than those of other techniques, with approximately half the points concentrated in an area with the ratio above 0.5, thus proving that the AAPS algorithm assigns energy more efficiently.

Why the utilization ratio of most points in FidgetyAFL and AFLFast.new is under 0.5? As stated in Section 2.1, if AFL finds a new path in random strategies, AFL will double the energy assigned to this seed. FidgetyAFL and AFLFast both follow this mechanism. However, Fig. 7 shows that this mechanism can create unnecessary energy depletion as, often during allocation, fuzzers do not find new paths after doubling energy. Thus, the remaining energy is wasted. On the other hand, our AAPS algorithm eliminates this mechanism that doubles the assigned energy and introduces an adaptive mechanism. If more energy has been assigned compared to the seeds that need to find new paths for some time, the AAPS algorithm helps reduce the next energy allocation to decrease energy depletion. Therefore, the distribution of points in EcoFuzz is more even compared to that in other tools.

Table 3: The evaluation of power schedule

Techniques	Average utilization ratio	Effective allocation	Average-cost
EcoFuzz	**0.121**	**0.290**	**4,314**
FidgetyAFL	0.005	0.013	9,078
AFLFast.new	0.010	0.031	7,046
FairFuzz	0.107	0.204	4,930

In detail, we calculated some indicators to evaluate the AAPS algorithm. Table 3 shows that the efficiency of different scheduling algorithms on `nm`. EcoFuzz demonstrates the best performance with the least average-cost, highest average utilization, and highest frequency of effective allocation. EcoFuzz's effective allocation frequency is more than FidgetyAFL, while its average-cost is half of FidgetyAFL.

We also evaluated the adaptive mechanism in AAPS. The adaptive mechanism was implemented on FidgetyAFL. This new FidgetyAFL + Adaptive fuzzer was run on `nm` and

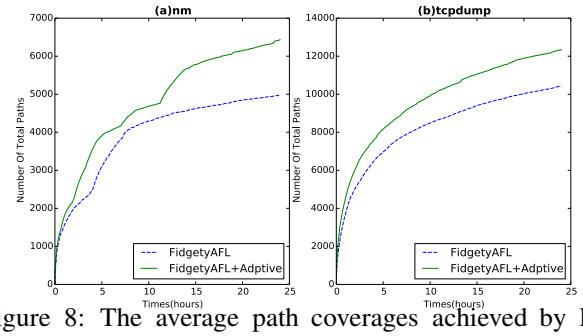

Figure 8: The average path coverages achieved by FidgetyAFL and FidgetyAFL + Adaptive.

`tcpdump` for 24 hours for 5 times. Fig. 8 shows the results. FidgetyAFL + Adaptive found more paths than FidgetyAFL on `nm` and `tcpdump`. It can be concluded the adaptive mechanism can improve the efficiency of AFL's power schedule.

Evaluation of SPEM Algorithm. As shown in Fig. 7, in the later stage of fuzzing where EcoFuzz transitions into the exploitation stage frequently, EcoFuzz's point distribution is denser than that of the other three tools. This qualitatively illustrates that the SPEM algorithm is effective.

More quantitatively, we calculate the frequency of effective allocation for the seeds chosen repeatedly in the exploitation stage to estimate the efficiency of the search strategies. The results are shown in Table 4. EcoFuzz's measured 0.069, which is more than FidgetyAFL at 0.031 and AFLFast.new at 0.026, thus proving that the SPEM algorithm is efficient.

Table 4: The evaluation of search strategy

Techniques	Allocation with New Finding	Repeated Chosen	Ratio
EcoFuzz	**705**	10,174	**0.069**
FidgetyAFL	364	**11,703**	0.031
AFLFast.new	54	2,066	0.026
FairFuzz	0	0	-

5.4 The Validity on Detecting Vulnerabilities

As most tested software are the latest version, it is difficult for these tools to find crashes in them using the seeds provided by AFL. However, EcoFuzz still found 5 vulnerabilities. For further evaluating EcoFuzz's efficiency in detecting vulnerabilities, we attempted to select the seeds for the latest version of the software by considering crashes in its previous version.

Unique Crashes. We tested GNU Binutils-2.31 programs with EcoFuzz and found few crashes in `nm` and `size` of GNU Binutils-2.31. Some crashes were selected as the initial seeds for testing the `nm` and `size` from GNU Binutils-2.32. As AFLFast.new outperforms the other five tools, we compared EcoFuzz with it. After 24 hours of testing, EcoFuzz found 53 and 63 unique crashes in `nm` and `size`, respectively, while AFLFast.new found 17 and 76 unique crashes.

Analysis of Vulnerabilities. EcoFuzz found more unique crashes than AFLFast.new in `nm` and fewer crashes than AFLFast.new in `size`. We used AddressSanitizer for further vulnerability analysis [31]. After analysis, EcoFuzz and AFLFast.new both detect the vulnerability in `nm` when calling the *d_expression_1* function in *cp-demangle.c*, which has been confirmed as the CVE-2019-9070 by others. Moreover, two 0-day heap buffer overflow vulnerabilities exist in `size` that are only found by EcoFuzz. One is trigged when calling the *bfd_hash_hash* function and the other is triggered when calling the *_bfd_doprnt* function. Although AFLFast.new found more crashes in `size` than EcoFuzz, it failed to trigger these two bugs. We submitted the bugs for requiring CVEs, and the heap buffer overflow in *_bfd_doprnt* has been affirmed as CVE-2019-12972. Besides, when testing GNU Binutils-2.31, EcoFuzz found four stack-overflow in *xmalloc.c* and *cplus-dem.c*. They were reported to the Binutils group and have been patched. Table 8 in Appendix 8.2 presents the analysis of all vulnerabilities. These results show that EcoFuzz can detect vulnerabilities efficiently in some real-world programs.

5.5 Evaluation on LAVA-M

The LAVA-M dataset is proposed as a benchmark for assessing the fuzzers' performance [12]. The dataset contains four programs that are `base64`, `md5sum`, `uniq`, and `who`. Each program was generated by injecting some bugs into the source code. Recently, several fuzzers (e.g., VUzzer, Steelix, Angora, and T-Fuzz [10, 19, 27, 29]) used this benchmark in evaluation.

Baseline. In addition to tools in Section 5.2, we compared EcoFuzz with other state-of-the-art tools on LAVA-M, including Angora and VUzzer [10, 29].

Configuration. Since our platform in Section 5.2 was not connected to the Internet, for installing and running Angora as well as VUzzer, we deployed them on our cloud server, a ubuntu 16.04 server os with 8 cores (Intel Xeon Platinum 8163 CPU @ 2.50GHz) and 16GB of RAM. A similar experiment was also conducted by executing each program for 5 hours, such that the configuration was the same as that in VUzzer and Angora. Each experiment was repeated 5 times. Further, EcoFuzz was run with the static analysis module, and the dictionary that this module generated is provided for all AFL-type fuzzers. Table 5 lists the total bugs found by all fuzzers during the five runs.

Discovered Bugs. As shown in Table 5, EcoFuzz found the most bugs and outperformed others on LAVA-M. On `base64`, `md5sum`, and `uniq`, EcoFuzz found all listed as well as unlisted bugs. On `who`, as there were numerous bugs in `who`, the efficiency of detecting bugs of each fuzzer can be evaluated distinctly. It was observed that EcoFuzz found the most bugs on `who` than the other fuzzers, with 1,252 listed and 200 unlisted bugs. Moreover, AFLFast.new performed the best in other techniques, but it was not better than EcoFuzz. Angora found 1,012 listed and 155 unlisted bugs, which is less than those found by EcoFuzz.

Moreover, the result showed that AFL-type fuzzers could also find numerous bugs on LAVA-M in the dictionary model, with finding almost all bugs in `base64`, `md5sum`, and `uniq`.

Table 5: The number of total bugs discovered in LAVA-M

Program	Bugs	AFL	AFLFast	FidgetyAFL	AFLFast.new	FairFuzz	MOPT-AFL	Angora	VUzzer	EcoFuzz
base64	44	44(+4)	44(+4)	44(+4)	44(+4)	44(+4)	44(+4)	43(+1)	1(+0)	**44(+4)**
md5sum	57	57(+1)	57(+3)	57(+4)	57(+4)	57(+3)	57(+0)	57(+4)	16(+0)	**57(+4)**
uniq	28	28(+1)	28(+1)	28(+1)	28(+1)	28(+1)	28(+1)	28(+1)	28(+1)	**28(+1)**
who	2136	466(+22)	490(+28)	1132(+158)	1147(+164)	463(+28)	71(+3)	1012(+155)	47(+6)	**1252(+200)**

* Listed and (+unlisted bugs) found by existing techniques and EcoFuzz.

In addition, EcoFuzz outperformed other AFL-type fuzzers on `who`, with finding 3× more bugs than AFL. Therefore, EcoFuzz is efficient in discovering bugs in LAVA-M. Since AFL-type fuzzers are deployed in our platform, where the configuration is slightly different from the cloud server, the comparison of EcoFuzz with Angora and VUzzer in Table 5 may not be strict enough. Therefore, we implement EcoFuzz on the same cloud server and do more analysis in Appendix 8.3.

5.6 Extended Application for EcoFuzz

The previous evaluation proved that EcoFuzz could find more paths than other AFL-type fuzzers in most cases with lower average-cost. There are also certain specific cases, such as when the test cases have slow execution speed and there is a low upper bound of paths (e.g., fuzzing the IoT devices or binary programs via QEMU), where EcoFuzz's advantages are prominent.

In such cases, EcoFuzz was applied on IoTHunter [37] to fuzz the `SNMP` component [8]. In RoutterOS'6.44.3 stable version, a vulnerability of `SNMP` component was observed. This issue was declared to be a failure of the processing input `SNMP` packet that may lead to a denial of service. The `SNMP` process will crash and restart when the packet in POC is received. Although `SNMP` does restart after a crash, repeated crashes might create an extended Denial of Service (DoS) condition, as shown in Table 8. Though we had submitted the crash, Mikrotik company released a new version of 6.45beta54 that has patched the bug.

6 Discussion

Compared to other techniques, EcoFuzz can effectively explore more paths in the same number of executions. The adaptive mechanism implemented by EcoFuzz enables EcoFuzz to flexibly revise subsequent energy allocations as per the current utilization ratio of energy.

It is noteworthy that EcoFuzz developed AFL's search strategy and power schedule, not including the mutation strategies, to be similar to that of AFLFast. That is, EcoFuzz does not change the transition probability p_{ij}, which is different from FairFuzz. Though FairFuzz improves the efficiency of random mutation, the result shows that EcoFuzz outperforms FairFuzz in terms of the ability to explore more paths while consuming less energy. Additionally, when testing the real-world software, sometimes the ability to maximize the coverage while saving energy is crucial for CGF. This has already been explained by implementing EcoFuzz for testing the IoT devices.

As EcoFuzz is built on AFL, EcoFuzz follows AFL's advantages. Compared to VUzzer [29] or other greybox fuzzing with taint analysis techniques, EcoFuzz's execution speed is higher. EcoFuzz also benefits from certain techniques used for enhancing AFL (e.g., CollAFL [13]), thus ensuring that EcoFuzz's performance can still be enhanced.

More importantly, regardless of which program analysis technique is used, whether the goal is to maximize coverage or explore rare branches, selecting an optimal seed to fuzz and assigning suitable energy are crucial for enhancing efficiency. The VAMAB model can still optimize the power schedule of other fuzzers, whether they are AFL-type fuzzers or other greybox fuzzers, by simply modifying the definition of goal and rewards as per the actual requirement.

7 Related Work

7.1 Scheduling Algorithms in Fuzzing

As a novel work that focuses on improving AFL's scheduling algorithm, AFLFast proposed a crucial concept transition probability for illustrating the transition between different paths, providing the direction of improving efficiency in power schedule and search strategy [6]. However, AFLFast did not conduct a deeper study of the transition probability. In contrast, we developed a VAMAB model for explaining the fuzzing process in terms of game theory and presented the reward probability of depicting each seed's ability to find new paths according to the transition probability. We also illustrated the probability attenuation of reward probability and stated the reward probability was not calculated accurately. Moreover, the fuzzing process was classified into three states, and the challenges of the different states were explained, followed by suggesting optimal strategies for each state. Compared to the Markov chain, our model reveals the challenges in scheduling algorithms more profoundly.

Woo *et al.* [36] once stated searching over the parameter space of blackbox fuzzing as the MAB problem. However, the goal of Woo *et al.* was finding the highest number of unique bugs, which is not applicable to CGF. If more energy is assigned to the seeds finding crashes, it may only trigger the same crashes. This is one of the reasons for not selecting the number of crashes as the target of our VAMAB model. On

the other hand, aiming coverage helped in finding more seeds exercising rare paths, thus aiding in finding unique crashes in different functions. In addition, Patil *et al.* [26] modeled the problem of deciding the number of random fuzzing iterations as Contextual Bandits (CB) problem between the full reinforcement learning problem and MAB problem [18]. Patil *et al.* considered the seeds as arms and proposed multipliers of the test case's energy, treating them as the arms in the contextual bandit setting [26]. The aim of Patil *et al.* was to determine the energy value from the test case contents by using reinforcement learning techniques. However, their work did not utilize the model for explaining the details of the fuzzing process and only presented an algorithm to decide a test case's energy multiplier, given fixed length contents of the test case [26]. In contrast, we considered the trade-off between exploration and exploitation of power schedules in CGF in detail. Therefore, our VAMAB model is better suited for modeling the scheduling algorithm of CGF than MAB or CB. To the best of our knowledge, we are the first to model the scheduling problem as VAMAB.

7.2 Smart Seeds Generation or Selection

Certain directions for enhancing CGF can be understood based on the VAMAB model. The first research direction is to improve the quality of the initial seeds, and this includes selecting the seed inputs from a wealth of inputs [30] or generating well-distributed seed inputs for fuzzing programs that process highly-structured inputs [33]. The core of these works is providing the high reward probability seeds to the initial state. As stated in Section 5.4, EcoFuzz can also benefit from a smart mechanism of seed generation. Besides, there are researchers who aim to establish the mechanism for estimating each seed's quality, which can help fuzzers accurately select the seeds with high reward probability. Further, Zhao *et al.* [44] designed a Monte Carlo-based probabilistic path prioritization model for quantifying each path's difficulty and prioritizing them for concolic execution as well as implementing a prototype system DigFuzz. Moreover, Böhme *et al.* [5] proposed the Directed Greybox Fuzzing by using the distance between the seeds and the target to measure the seeds' quality. Based on the VAMAB model, these researches provide certain methods for accurately estimating the reward probability of their problem. EcoFuzz also uses the SPEM algorithm to measure the quality of seeds. Moreover, the experiments in our evaluation showed that the frequency of effective searching in SPEM is approximately twice that of FidgetyAFL on `nm`, which is regarded as a precise method for estimating the quality of seeds. Besides, compared to AFLGo [5] and DigFuzz [44], EcoFuzz does not require additional program analysis techniques to achieve the same goals.

7.3 Greybox Fuzzing with Optimizing Mutation Strategies

Several approaches focus on the second direction that enhances the mutation efficiency by using program analysis techniques. Some approaches aim to find locations in seed inputs related to high-probability crash locations or to determine statements in the program [10,34], and other approaches try to learn input format and utilize it for assisting mutation. VUzzer [29] leveraged control- and data-flow features of targets and used this information in the feedback loop for generating new inputs. However, VUzzer realized this function based on `Pin` [20], which is slower than the techniques of instruments used by EcoFuzz.

FairFuzz is implemented on AFL and can identify the parts of the input that are crucial for satisfying the determined conditions. In test cases generation, it avoids mutating these crucial parts of the input and reduces the number of fuzz exercising high-frequency paths [17]. Nevertheless, FairFuzz achieves this function depending on the deterministic strategies being implemented, which is not as effective as the random mutation. In this paper, EcoFuzz was assessed against FairFuzz, and it had been proved that, with the same number of executions, EcoFuzz outperforms FairFuzz in exploring paths.

Some researchers aim to learn file formats and use them in mutation to improve efficiency. Learn&Fuzz [14] used sequence-based learning methods for the PDF's structures. Further, AFLSmart [28] kept the format attribute unchanged in the mutation by providing prior knowledge. However, such techniques require lots of initial files or prior knowledge, making it difficult to implement in testing real-world programs. In contrast, EcoFuzz can be started conveniently.

8 Conclusion

In this paper, we proposed a variant of the Adversarial Multi-Armed Bandit (VAMAB) model and used it for modeling the scheduling problem in CGF. We also introduced the reward probability for illustrating the ability of each seed to discover new paths and explained problems such as the probability attenuation. In addition, we classified the states of the seeds set into three categories and illustrated the challenges and opportunities in these states. Based on this, we proposed the SPEM for measuring the reward probability and developed an adaptive power schedule. We implemented these algorithms on an adaptive energy-saving greybox fuzzer called EcoFuzz. EcoFuzz explores more paths than six AFL-type fuzzers with fewer executions, significantly reducing the average-cost for discovering a new path. Besides, EcoFuzz's adaptive mechanism and energy-saving advantages can help improve other techniques. EcoFuzz was also compared with other works, and their optimization directions were explained by the VAMAB model, indicating that the applicability of our model is strong.

Since our VAMAB model is related to the reinforcement

learning and the schedule algorithms of EcoFuzz are slightly empirical, in the future, we may consider to optimize the schedule algorithms and improve our work by implementing some methods of reinforcement learning.

Acknowledgments

The authors would like to thank our shepherd Deian Stefan and anonymous reviewers for their valuable comments and helpful suggestions. The authors are supported in part by Tianhe Supercomputer Project 2018YFB0204301, National Science Foundation of Hunan Province in China (2019JJ50729), and National Science Foundation China under Grant 61902412 and 61902416.

References

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

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

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

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[7] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[8] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1990.

[9] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.

[10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[11] Brendan Dolan-Gavitt. Of bugs and baselines, 2018.

[12] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[13] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[14] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

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

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

[17] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485. ACM, 2018.

[18] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670. ACM, 2010.

[19] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637. ACM, 2017.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

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

[22] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[23] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[24] Charlie Miller. Fuzz by number. In *CanSecWest Conference*, 2008.

[25] James R Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[26] Ketan Patil and Aditya Kanade. Greybox fuzzing as a contextual bandits problem. *arXiv preprint arXiv:1806.03806*, 2018.

[27] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[28] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *arXiv preprint arXiv:1811.09447*, 2018.

[29] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

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

[31] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 309–318, 2012.

[32] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[33] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[34] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*, pages 497–512. IEEE, 2010.

[35] Peter Whittle. Multi-armed bandits and the gittins index. *Journal of the Royal Statistical Society: Series B (Methodological)*, 42(2):143–149, 1980.

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

[37] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. Poster: Fuzzing iot firmware via multi-stage message generation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2525–2527. ACM, 2019.

[38] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.

[39] Michal Zalewski. Afl vulnerability trophy case. Website, 2014. http://lcamtuf.coredump.cx/afl/#bugs.

[40] Michal Zalewski. American fuzzy lop.(2014). Website, 2014. http://lcamtuf.coredump.cx/afl.

[41] Michał Zalewski. Fidgetyafl. Website, 2016. https://groups.google.com/forum/#!msg/afl-users/fOPeb62FZUg/CES5lhznDgAJ.

[42] Michał Zalewski. American fuzzy lop technical details. Website, 2018. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[43] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

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

Appendix

8.1 More Analysis of Average-Cost Evaluation

In this subsection, we implement a more in-depth analysis of the evaluation results in Section 5.2.

Path Coverage. From Fig. 6, EcoFuzz outperforms the other six fuzzers on most programs except `cxxfilt`, `readelf`, `djpeg`, `xmllint` and `infotocap`. For these five programs, on `xmllint` and `infotocap`, EcoFuzz finds more paths than other tools in the same number of executions. The path coverage EcoFuzz achieves is only slightly lower than FidgetyAFL or AFLFast.new. The reason is that they generate more test cases than EcoFuzz. On `cxxfilt`, EcoFuzz performs better than AFLFast.new and FidgetyAFL when the number of paths is below 7,000. After that, AFLFast.new and FidgetyAFL generate more test cases than EcoFuzz so that the paths discovered by AFLFast.new and FidgetyAFL are more than EcoFuzz. On `readelf`, EcoFuzz performs similarly to AFLFast.new and FidgetyAFL in the early stage. In the later stage, the number of paths discovered by EcoFuzz is slightly less than that of AFLFast.new and FidgetyAFL. On `djpeg`, as can be seen from Fig. 6, there are two significant increases in the curve of AFLFast.new and FidgetyAFL in the latter stage, which makes the numbers of paths found by AFLFast.new and FidgetyAFL exceed that of EcoFuzz. We analyze the result of each run on `djpeg` and find that there are two runs of AFLFast.new and FidgetyAFL discovering over 4,500 paths on `djpeg`, respectively. In other cases, the number of paths they found is approximate to that of EcoFuzz. We regard this as the impact of experimental contingency.

In addition, in most cases, fuzzers without indeterministic strategies (EcoFuzz, FidgetyAFL, and AFLFast.new) were noted to perform better than FairFuzz, AFL, AFLFast, and MOPT-AFL. This proves that the indeterministic mutation strategies are efficient in general. Particularly, EcoFuzz finds significantly more paths than these four tools, and overall, EcoFuzz performs better than six other techniques in path exploration and energy saving.

Average-Cost and Total Executions. From Table 2, notice that, on most cases, under the same testing hours, the number of test cases produced by EcoFuzz is far fewer than other techniques, especially on the subjects `cxxfilt`, `xmllint` and `infotocap`. The reason is that when EcoFuzz assigns energy to a seed, EcoFuzz does not take the execution time or length of the seed into consideration. That leads EcoFuzz to allocate energy on a long execution time seed as same as other some fast seed, which costs EcoFuzz more time to fuzz it than some other seeds. Besides, EcoFuzz has fuzzed all seeds from the queue, with implementing more executions on the *trim* strategy than other techniques. Different from our scheduling algorithm, the power schedules of other fuzzers we compare against to EcoFuzz are mainly based on that of AFL and maintain most features. As introduced in Section 2.1, during the indeterministic strategies, AFL assigns energy to the seed according to its performance score, which is calculated based on the execution time, coverage, and discovery time. The longer its execution time is, the less energy is allocated. This mechanism guarantees that AFL will not spend a lot of time on fuzzing these long execution time seeds. However, it makes sense to allocate energy to these long execution time seeds, which also helps us to improve the coverage.

More Statistical Analysis. In Section 5.2, we have reported the results of statistical analysis and pointed out that EcoFuzz outperforms other tools in general. In this subsection, we analyze the statistical results of p value and extremum in detail.

From Table 6, on the path coverage, p_1 is smaller than 10^{-4} in all evaluations, indicating that the distribution of total paths found by EcoFuzz and AFL differs significantly. Further, p_3, p_5, and p_6 are also mostly tend to be smaller than 10^{-3}, which proves that EcoFuzz also outperforms AFLFast, FairFuzz, and MOPT-AFL notably in path exploration. In the majority of evaluation, p_4 is approximately the same as 10^{-1}, this indicating that the paths EcoFuzz and AFLFast.new find are not significantly different. However, on the average-cost, p_4 is smaller than 10^{-2} on 11 evaluations, thus proving that EcoFuzz's average-cost is significantly lower than that of AFLFast.new.

From Table 7, EcoFuzz and AFLFast.new outperform the other five tools on most programs, whether in the maximum or the minimum of discovered paths. EcoFuzz achieves the upper bound of the maximum of path coverage on six programs, minimum of path coverage on eight programs. Compared to AFLFast.new, though EcoFuzz achieves the path coverage approximate to AFLFast.new, the energy depletion of EcoFuzz is lower than AFLFast.new.

8.2 Analysis of Vulnerabilities Detected by EcoFuzz

In Section 5.4, we evaluated the validity of EcoFuzz on detecting vulnerabilities and reported some vulnerabilities found by EcoFuzz in general. We state some detailed analysis of these vulnerabilities in this subsection.

In addition to the bugs found in GNU Binutils, EcoFuzz also found 5 vulnerabilities on some programs tested in Section 5.2, with 2 heap-buffer-overflow in `gif2png`, and `tcpdump`, as well as 3 memory leak in `libpng` and `jhead`, which were only found by EcoFuzz, FidgetyAFL and AFLFast.new. In detail, there are 2 vulnerabilities found in `gif2png`, a heap-buffer-overflow in the *writefile* function in *gif2png.c* and a memory leek in the *xalloc* function in *memory.c*. In addition, since `gif2png` is built on `libpng`, EcoFuzz also found a memory leak in *png_malloc_warn* in *pngmem.c* of `libpng` when recurred a crash in `gif2png`. Moreover, EcoFuzz found a heap-buffer-overflow in `jhead`, which

Table 6: The p-value result in each evaluation

Subjects	Number of total paths						Average-cost					
	p_1	p_2	p_3	p_4	p_5	p_6	p_1	p_2	p_3	p_4	p_5	p_6
nm	$1.2*10^{-7}$	$1.2*10^{-2}$	$1.3*10^{-4}$	$6.4*10^{-1}$	$1.2*10^{-4}$	$3.9*10^{-7}$	$1.5*10^{-5}$	$8.5*10^{-3}$	$5.7*10^{-4}$	$2.9*10^{-4}$	$3.2*10^{-3}$	$1.4*10^{-5}$
objdump	$5.0*10^{-8}$	$1.4*10^{-1}$	$1.9*10^{-7}$	$2.6*10^{-1}$	$2.2*10^{-6}$	$3.6*10^{-8}$	$9.5*10^{-8}$	$1.9*10^{-3}$	$1.4*10^{-5}$	$5.6*10^{-3}$	$1.2*10^{-2}$	$4.0*10^{-8}$
readelf	$9.2*10^{-7}$	$5.2*10^{-1}$	$2.4*10^{-5}$	$4.6*10^{-2}$	$2.8*10^{-4}$	$1.4*10^{-6}$	$4.9*10^{-8}$	$6.2*10^{-1}$	$2.4*10^{-6}$	$7.9*10^{-1}$	$3.5*10^{-2}$	$8.8*10^{-9}$
size	$9.2*10^{-7}$	$2.8*10^{-5}$	$4.6*10^{-6}$	$9.5*10^{-3}$	$1.3*10^{-6}$	$4.8*10^{-6}$	$1.0*10^{-5}$	$7.2*10^{-6}$	$1.4*10^{-3}$	$1.3*10^{-4}$	$2.9*10^{-1}$	$4.8*10^{-4}$
cxxfilt	$5.8*10^{-6}$	$4.4*10^{-3}$	$3.3*10^{-5}$	$7.1*10^{-4}$	$6.1*10^{-8}$	$3.2*10^{-1}$	$4.4*10^{-7}$	$5.1*10^{-8}$	$1.4*10^{-7}$	$2.4*10^{-6}$	$4.4*10^{-7}$	$2.4*10^{-6}$
djpeg	$7.6*10^{-5}$	$2.3*10^{-1}$	$9.3*10^{-4}$	$1.4*10^{-1}$	$9.2*10^{-6}$	$4.4*10^{-2}$	$7.4*10^{-4}$	$4.8*10^{-2}$	$1.2*10^{-3}$	$3.7*10^{-1}$	$1.5*10^{-4}$	$2.4*10^{-5}$
xmllint	$9.3*10^{-9}$	$6.6*10^{-3}$	$1.7*10^{-7}$	$6.1*10^{-2}$	$1.9*10^{-3}$	$1.9*10^{-3}$	$2.0*10^{-5}$	$5.5*10^{-6}$	$1.4*10^{-7}$	$2.9*10^{-4}$	$8.6*10^{-5}$	$3.0*10^{-6}$
gif2png	$4.2*10^{-7}$	$6.3*10^{-4}$	$4.6*10^{-4}$	$1.5*10^{-1}$	$2.2*10^{-6}$	$1.8*10^{-4}$	$1.2*10^{-1}$	$9.2*10^{-4}$	$5.1*10^{-1}$	$1.0*10^{-3}$	$2.1*10^{-3}$	$7.2*10^{-3}$
readpng	$7.1*10^{-6}$	$4.3*10^{-2}$	$9.1*10^{-5}$	$3.2*10^{-1}$	$9.8*10^{-2}$	$4.8*10^{-2}$	$3.6*10^{-3}$	$2.0*10^{-4}$	$1.3*10^{-3}$	$2.1*10^{-4}$	$5.4*10^{-2}$	$4.3*10^{-4}$
tcpdump	$1.8*10^{-6}$	$2.3*10^{-3}$	$7.1*10^{-6}$	$9.1*10^{-1}$	$3.6*10^{-2}$	$2.6*10^{-5}$	$3.9*10^{-7}$	$1.4*10^{-2}$	$1.1*10^{-4}$	$1.4*10^{-3}$	$1.1*10^{-4}$	$3.4*10^{-7}$
infotocap	$4.5*10^{-6}$	$2.7*10^{-1}$	$7.0*10^{-5}$	$1.6*10^{-1}$	$8.7*10^{-5}$	$3.3*10^{-2}$	$6.4*10^{-6}$	$3.3*10^{-5}$	$7.1*10^{-8}$	$7.3*10^{-5}$	$1.4*10^{-6}$	$1.5*10^{-7}$
jhead	$5.7*10^{-6}$	$1.5*10^{-4}$	$6.9*10^{-5}$	$1.8*10^{-4}$	$7.9*10^{-6}$	$1.8*10^{-4}$	$8.4*10^{-7}$	$3.0*10^{-4}$	$6.0*10^{-6}$	$1.4*10^{-10}$	$1.4*10^{-8}$	$1.3*10^{-3}$
magick	$1.8*10^{-10}$	$3.8*10^{-2}$	$3.3*10^{-9}$	$4.4*10^{-1}$	$1.6*10^{-5}$	$7.1*10^{-7}$	$5.6*10^{-9}$	$2.3*10^{-2}$	$3.8*10^{-8}$	$1.9*10^{-5}$	$5.9*10^{-5}$	$3.3*10^{-5}$
bsdtar	$1.0*10^{-10}$	$6.7*10^{-3}$	$9.4*10^{-6}$	$7.8*10^{-1}$	$3.3*10^{-7}$	$6.1*10^{-7}$	$2.9*10^{-10}$	$2.6*10^{-3}$	$9.7*10^{-5}$	$9.4*10^{-2}$	$2.3*10^{-3}$	$1.1*10^{-5}$

Table 7: The maximum and minimum of discovered paths in each evaluation

Subjects	Maximum / Minimum of Discovered Paths						
	AFL	FidgetyAFL	AFLFast	AFLFast.new	FairFuzz	MOPT-AFL	EcoFuzz
nm	2,651 / 4,074	3,197 / 7,671	2,675 / 5,548	7,406 / **8,966**	2,683 / 5,613	2,547 / 4,069	**7,986** / 8,659
objdump	3,633 / 4,238	6,952 / 7,496	3,791 / 4,520	6,933 / 7,587	5,033 / 5,646	4,361 / 4,549	**7,063** / **7,810**
readelf	5,371 / 5,840	12,118 / 14,032	7,997 / 8,332	**13,110** / **14,813**	8,111 / 10,124	5,723 / 6,189	11,555 / 14,337
size	2,279 / 2,644	3,285 / 3,408	1,685 / 2,586	3,467 / 3,870	2,597 / 2,928	2,761 / 3,093	**3,727** / **4,097**
cxxfilt	3,329 / 4,786	7,424 / 8,020	3,883 / 5,237	**7,632** / **8,756**	4,906 / 5,278	6,125 / 7,352	6,847 / 7,393
djpeg	2,063 / 2,320	2,840 / 4,794	2,073 / 2,502	**2,940** / **4,895**	1,780 / 2,010	2,199 / 2,943	2,807 / 3,380
xmllint	3,385 / 3,591	6,114 / 6,435	3,886 / 4,347	**6,864** / **7,573**	4,732 / 6,268	5,742 / 6,259	6,304 / 7,062
gif2png	2,551 / 3,122	3,946 / 4,193	1,906 / 3,559	4,112 / 4,332	2,627 / 3,234	3,723 / 4,009	**4,204** / **4,347**
readpng	1,463 / 1,598	1,757 / 2,001	1,486 / 1,685	1,812 / 2,132	1,413 / **2,177**	1,608 / 1,981	**1,923** / 2,168
tcpdump	5,987 / 6,830	9,776 / 11,201	5,499 / 7,680	**12,456** / 13,321	10,678 / 12,635	7,393 / 8,612	12,417 / **15,191**
infotocap	2,849 / 3,914	5,428 / 6,433	4,089 / 4,794	**5,507** / **7,136**	3,388 / 4,668	4,932 / 5,831	5,443 / 6,240
jhead	482 / 513	527 / 551	511 / 545	524 / 552	496 / 521	528 / 556	**577** / **619**
magick	1000 / 1,164	4,379 / 5,623	1,891 / 2,230	5,132 / 5,567	2,853 / 3,948	3,116 / 3,739	**5,268** / **5,873**
bsdtar	2,691 / 2,823	6,367 / 6,906	2,139 / 4,148	6,490 / **7,648**	3,292 / 4,395	2,536 / 4,081	**7,006** / 7,581

is triggered in the *process_DQT* function in *jpgqguess.c* and has been requested as CVE-2020-6624 by others. This vulnerability was only found by EcoFuzz, FidgetyAFL and AFLFast.new, thus proving that EcoFuzz is more efficient than AFL and AFLFast in detecting vulnerabilities. In addition, we recompiled and tested `tcpdump` with the ASAN model of AFL. EcoFuzz found a memory leak in the *copy_argv* function in *tcpdump.c*. Finally, we submitted these 5 vulnerabilities and obtain CVE-2019-17371 as the memory leak in `libpng`. All vulnerabilities are listed in Table 8.

Table 8: The discovered vulnerabilities

Softwares	File/Function	Status
Binutils-2.32	cp-demangle.c/d_expression_1	CVE-2019-9070
Binutils-2.32	hash.c/bfd_hash_hash	Acknowledged
Binutils-2.32	bfd.c/_bfd_doprnt	CVE-2019-12972
Binutils-2.31	xmalloc.c/xmalloc	Patched
Binutils-2.31	cplus-dem.c/string_append	Patched
Binutils-2.31	cplus-dem.c/string_append_template_idx	Patched
Binutils-2.31	cplus-dem.c/demangle_class_name	Patched
gif2png-2.5.13	gif2png.c/writefile	Submitted
gif2png-2.5.13	memory.c/xalloc	Submitted
libpng-1.6.37	pngmem.c/png_malloc_warn	CVE-2019-17371
tcpdump-4.9.2	tcpdump.c/copy_argv	Acknowledged
jhead-3.03	jpgqguess.c/process_DQT	CVE-2020-6624
SNMP deamon	snmp/Context::createReply	Patched

8.3 More Analysis of Experiments on LAVA-M

In Section 5.5, we evaluate the performance of each technique on LAVA-M in general. We also point out the comparison between EcoFuzz with Angora and VUzzer is not strict enough. Now we do a more in-depth and detailed analysis.

We deployed EcoFuzz on the cloud server in Section 5.5. We also run EcoFuzz with the same setting as in Section 5.5. After validating the bugs detected by EcoFuzz during 5 times of 5-hours runs, EcoFuzz found all listed and unlisted bugs on `base64`, `md5sum`, and `uniq`, with 48(+4), 57(+4) and 28(+1) bugs. For `who`, EcoFuzz found 1,966 bugs in total, with 1,750 listed and 216 unlisted bugs, which are both more than that of Angora and VUzzer. In detail, EcoFuzz detected 1,139, 1,365, 1,377, 1,450 and 1,210 bugs on `who` in each run, respectively. Since different environments have an impact on the experimental results and there is non-negligible randomness in the experiment of fuzzing, it is not objective to deduce that EcoFuzz can always outperform Angora on LAVA-M from the results in our evaluation. In the origin paper, Angora can find 1,541 bugs on `who` in one 5-hours run [10], which states that Angora is still an efficient and state-of-the-art tool in detecting the bugs in LAVA-M.

From these results, on `base64`, `md5sum`, and `uniq`, EcoFuzz found all the listed and unlisted bugs, as same as FidgetyAFL and AFLFast.new. Angora also performs well on these three programs. Furthermore, these four tools all detected numerous bugs in `who`.

Moreover, AFL-type fuzzers all perform well on LAVA-M in the dictionary mode. In fact, the way to trigger the bugs injected in LAVA-M is extremely simple, just satisfying the

comparison of some four-byte magic bytes in some positions. However, AFL could not recognize magic bytes in the conditional statement. Therefore, a comparison of four-byte magic bytes will cost AFL too much energy to traverse. Some techniques using taint tracking or symbolic execution outperform than AFL without a dictionary on LAVA-M [10, 38]. In practice, the static analysis module of EcoFuzz has solved the problem by extracting the hard-code and magic bytes in LAVA-M. Therefore, it is an efficient way to combine the low-overhead program analysis techniques (e.g., static analysis) with the high-speed greybox fuzzing (e.g., AFL). Finally, all unlisted bugs found by EcoFuzz in different environments are listed in Table 9.

Table 9: The unlisted bugs found by EcoFuzz

Program	IDs of the unlisted bugs found by EcoFuzz
base64	274, 521, 526, 527
md5sum	281, 287, 314, 499
uniq	227
who	2, 4, 6, 8, 20, 61, 63, 73, 77, 81, 85, 89, 117, 125, 165, 169, 173, 177, 181, 185, 189, 193, 197, 210, 214, 218, 222, 226, 294, 298, 303, 307, 312, 316, 321, 325, 327, 334, 336, 338, 346, 350, 355, 359, 450, 454, 459, 463, 468, 472, 477, 481, 483, 488, 492, 497, 501, 504, 506, 512, 514, 522, 526, 531, 535, 974, 975, 994, 995, 996, 1007, 1026, 1034, 1038, 1049, 1054, 1071, 1072, 1329, 1334, 1339, 1345, 1350, 1355, 1361, 1377, 1382, 1388, 1393, 1397, 1403, 1408, 1415, 1420, 1429, 1436, 1445, 1450, 1456, 1461, 1718, 1727, 1728, 1735, 1736, 1737, 1738, 1747, 1748, 1755, 1756, 1891, 1892, 1893, 1894, 1903, 1904, 1911, 1912, 1921, 1925, 1935, 1936, 1943, 1944, 1949, 1953, 1993, 1995, 1996, 2000, 2004, 2008, 2012, 2014, 2019, 2023, 2027, 2031, 2034, 2035, 2039, 2043, 2047, 2051, 2055, 2061, 2065, 2069, 2073, 2077, 2079, 2081, 2083, 2181, 2189, 2194, 2219, 2221, 2223, 2225, 2229, 2231, 2235, 2236, 2240, 2244, 2246, 2247, 2249, 2253, 2255, 2258, 2262, 2266, 2268, 2269, 2271, 2275, 2282, 2286, 2291, 2295, 2302, 2304, 2462, 2500, 2507, 2521, 2681, 2703, 2790, 2804, 2806, 2810, 2814, 2823, 2827, 2834, 2838, 2847, 2854, 2919, 2920, 2922, 3082, 3083, 3099, 3185, 3187, 3188, 3213, 3218, 3222, 3232, 3235, 3237, 3238, 3239, 3242, 3245, 3247, 3249, 3256, 3257, 3260, 3264, 3265, 3267, 3269, 3389, 3464, 3465, 3468, 3469, 3471, 3487, 3488, 3495, 3496, 3509, 3510, 3517, 3523, 3527, 3545, 3551, 3561, 3939, 4024, 4025, 4026, 4222, 4223, 4224, 4225, 4287, 4295