Rainfuzz: Reinforcement-Learning Driven Heat-Maps for Boosting Coverage-Guided Fuzzing

Lorenzo Binosi[a], Luca Rullo, Mario Polino[b], Michele Carminati[c] and Stefano Zanero[d]
Politecnico di Milano, Milan, Italy

Keywords: Fuzzing, Heat-Maps, Reinforcement-Learning.

Abstract: Fuzzing is a dynamic analysis technique that repeatedly executes the target program with many different inputs to trigger abnormal behavior, such as a crash. One of the most successful techniques consists in generating inputs to increase code-coverage by using a mutational approach: this type of fuzzers maintains a population of inputs, they perform mutations on the inputs in the current population, and they add mutated inputs to the population if they discover new code-coverage in the target program. Researchers are continuously looking for techniques to increment the efficiency of fuzzers; one of these techniques consists in generating heat-maps for targeting specific bytes during the mutation of the input, as not all bytes might be useful for controlling the program's workflow. We propose the first approach in the literature that uses reinforcement learning for building heat-maps, by formalizing the problem of choosing the position to be mutated within the input as a reinforcement-learning problem. We model the policy by means of a neural network, and we train it by using Proximal Policy Optimization (PPO). We implement our approach in Rainfuzz, and we show the effectiveness of its heat-maps by comparing Rainfuzz against an equivalent fuzzer that performs mutations at random positions. We achieve the best performance by running AFL++ and Rainfuzz in parallel (in a collaborative fuzzing setting), outperforming a setting where we run two AFL++ instances in parallel.

1 INTRODUCTION

In a world where technology plays such a significant role, by influencing many aspects of our lives, it is extremely important to rely on secure software. Software vulnerabilities are what make software insecure, by making it possible for an attacker to violate CIA (Confidentiality, Integrity, Availability). Whenever new software is developed or existing software is changed, it is reasonable to consider that software vulnerabilities are introduced as well. For this reason, security best practices are usually inserted in the software development life cycle. One of the most popular and promising practices to address vulnerability detection is fuzzing. Fuzzing consists in repeatedly executing the Program Under Test (PUT) by providing it with many different inputs, with the intent of finding an abnormal behavior (for instance, by causing a crash). There are many different types of fuzzers (Zalewski, 2016), (Fioraldi et al., 2020), (Google, 2016), (LLVM, 2017). They mainly differ from each other due to the way they generate new inputs to be tested. A very popular category is the one of gray-box mutational fuzzers: this class of fuzzers employs a genetic algorithm to generate increasingly interesting inputs: they maintain a population of inputs, at each step, they apply a mutation to one of these inputs, and they use code-coverage as a fitness function in order to decide whether to keep the mutated input in the population or not. AFL (Zalewski, 2016), which is one of the most popular fuzzers, falls under this category. In recent years researchers have dedicated a lot of effort to improving fuzzers' performance by using machine learning techniques (Wang et al., 2019b). In this work, we focus on machine learning techniques that learn which bytes within the input are more convenient to mutate; this process is often referred to in the literature as creating heat-maps associated to an input that guide the fuzzer when deciding which positions within the input to choose for mutation. Two noteworthy approaches that try to achieve the same results are reported in (Rajpal et al., 2017) and (She et al., 2019). Both these approaches use supervised-learning techniques, and they both need to alternate

[a] https://orcid.org/0000-0001-7476-0166
[b] https://orcid.org/0000-0002-0925-2306
[c] https://orcid.org/0000-0001-8284-6074
[d] https://orcid.org/0000-0003-4710-5283

Binosi, L., Rullo, L., Polino, M., Carminati, M. and Zanero, S.
Rainfuzz: Reinforcement-Learning Driven Heat-Maps for Boosting Coverage-Guided Fuzzing.
DOI: 10.5220/0011625300003411
In *Proceedings of the 12th International Conference on Pattern Recognition Applications and Methods (ICPRAM 2023)*, pages 39-50
ISBN: 978-989-758-626-2; ISSN: 2184-4313

phases of training with phases of fuzzing, introducing significant overhead.

The approach explored in this work is the first attempt in the literature to use reinforcement learning to build heat-maps. We model the problem of choosing the next byte to mutate as a reinforcement learning problem, where states are inputs to be mutated, and actions consist in choosing a position to perform a series of mutations. The RL agent receives an higher reward the higher the effectiveness of the mutations performed at the position it chooses.

We implement our approach by means of Rainfuzz: a fuzzer built on top of AFL++ (Fioraldi et al., 2020) guided by a reinforcement learning module for its mutation strategy.

Overall, the main contributions of this work are:

- The first fuzzing approach guided by reinforcement learning heat-maps.
- We overcome the issue of alternating fuzzing and training phases, which are present in state-of-the-art approaches for building heat-maps.
- We provide evidence that the reinforcement learning policy outperforms the random policy; this is a great theoretical result, and it sets the stage for future research in building heat-maps using the same reinforcement learning formalization.
- We also show that running Rainfuzz and AFL++ in parallel (in a collaborative fuzzing setting) achieves better results than running two AFL++ instances in parallel; this result has direct practical uses.

2 FUZZING

Fuzzing is a commonly used technique for testing the reliability and security of software (Manès et al., 2021). The goal of fuzzing is to uncover software bugs, such as crashes, by providing a program with a wide range of different inputs. These bugs often have the potential to become vulnerabilities in the software. Over time researchers have developed more and more advanced methods, giving rise to various fuzzing techniques.

2.1 Classification

Below a brief summary of how fuzzers can be classified based on the techniques they use (Chen et al., 2018).

Mutation-Based vs Generation-Based Fuzzing. The core feature of fuzzers is to create new inputs to be fed into the program. In mutation-based fuzzers, new inputs are generated by taking old inputs and applying some mutations to them (mutations can be, for instance, `MIN_INT`, `MAX_INT`, `MIN_BYTE`, `MAX_BYTE`, bit-flipping, etc.). In generation-based fuzzers, a formal specification of the input format must be provided, and inputs are generated following the specification (for instance, if the specification is provided in the form of a formal grammar, inputs can be generated by randomly applying grammar rules). Generation-based fuzzers are particularly useful when the PUT parses the input and checks whether it is compliant with a specific grammar (e.g., program languages and data formats). In such a case, an input that is not compliant with the grammar would be rejected in the early stages of the program without exercising a large part of the program's code. When using mutation-based fuzzers in these scenarios, chances are that most of the inputs created are invalid, while using generation-based fuzzers, the input is guaranteed to be valid. The main drawback of generation-based fuzzers is that a formal specification of the input is not always provided, and creating it might be really challenging based on how well the specifications are described in the program's documentation.

Black-Box vs White-Box vs Gray-Box Fuzzing. In black-box fuzzing, the internal logic of the program is not observed, and mutations are applied blindly without any kind of feedback. On the opposite, in white-box fuzzing, the fuzzer observes the internal logic of the program and uses it to enhance the efficiency of the fuzzing process (for instance, a white-box fuzzer might use symbolic execution to generate an input so that a particular branch is taken). Gray-box fuzzing is a trade-off between the two, which observes just some aspects of the program execution (for instance, code-coverage information obtained by using lightweight code instrumentation).

2.2 AFL

AFL (American Fuzzy Lop) (Zalewski, 2016) is a gray-box mutation-based fuzzer. AFL uses a genetic algorithm that keeps a population of inputs (input corpus), performs various kinds of mutations starting from inputs in the population, and uses edge-coverage information generated by the execution of the program as a fitness function, to decide whether to add the mutated input to the population (for future mutation) or not.

To get code-coverage information, it uses a lightweight instrumentation of the program (which is what makes AFL gray-box). This simple idea comes

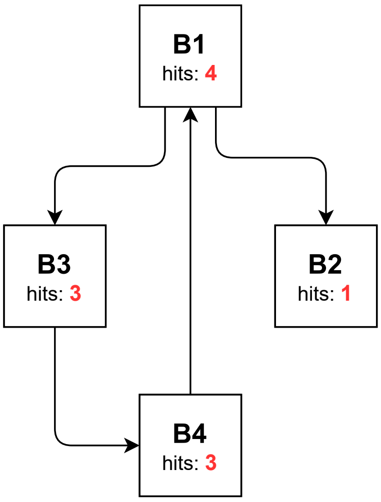

Figure 1: Control flow graph (CFG) of a small program, visualizing the block-coverage of an example execution.

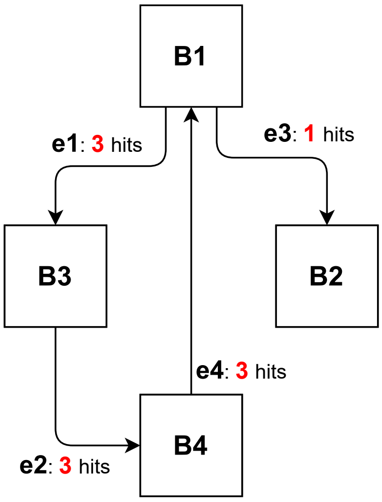

Figure 2: Control flow graph (CFG) of a small program, visualizing the edge-coverage of an example execution.

from the intuition that an input corpus that covers a more significant portion of the program's code is also more likely to uncover "buggy" portions of the code. AFL represents a milestone in the history of fuzzers. It discovered many bugs in various programs and paved the way for other successful fuzzers that work around the same idea. AFL is not maintained anymore; AFL++ (Fioraldi et al., 2020) is a community-driven fork of AFL that incorporates state-of-the-art fuzzing research.

2.3 Coverage Metrics

Many metrics can be used to measure the coverage of the program code; the key idea behind these metrics is to recognize different program behaviors. This plays a key role in coverage-guided fuzzing, as it allows to recognize which inputs exercise a different behavior in the program (w.r.t. the inputs currently in the corpus).

Ideally, it is possible to trace the whole execution path: when the program gets executed, keep track of the history of which basic-block are visited and in which order (for instance, $h = [B_1, B_3, B_2, B_1, \ldots]$). This way, two program executions with different execution paths are considered to exercise the program in a different way. This *path-coverage* metric allows to distinguish different program behaviors with very high sensitivity, but cannot be used in practice: using this coverage metric in a real-world fuzzer would make the input corpus grow very large and very fast (a lot of inputs will be judged interesting and maintained because they generate a different execution path than the ones already seen). On the other extreme, we have *block-coverage* (Figure 1), a coverage metric that, for each execution of the program, counts how many times a certain basic-block is visited (hit): the output generated by this metric is a map $m = [B_1 \rightarrow c_1, B_2 \rightarrow c_2, \ldots, B_n \rightarrow c_n]$ where B_i is the i-th block, and c_i is the number of times the i-th block was visited during this program execution.

A trade-off between execution path-coverage and block-coverage is the *edge-coverage* (Figure 2); for a given execution of the program, it keeps track of the number of hits of a certain edge (edges are the transitions between basic blocks – in the control flow graph of the program they are represented as arrows). The output generated by this metric is a map $m = [e_1 \rightarrow c_1, e_2 \rightarrow c_2, \ldots, e_n \rightarrow c_n]$ where e_i is the i-th edge, and c_i is the number of times the i-th edge was hit during this program execution. Edge-coverage is strictly more sensitive than block-coverage, meaning that: if two program executions have the same edge-coverage, that implies that they also have the same block-coverage, but the opposite is not true.

There are many more types of code-coverage (Wang et al., 2019a), but edge-coverage and block-coverage are the most popular among state-of-the-art fuzzers. AFL (and AFL++) use edge-coverage; we also use edge-coverage within Rainfuzz, both for deciding whether to keep a mutated input in the input-corpus and to evaluate the effectiveness of a mutation.

3 PROBLEM FORMALIZATION

During fuzzing, the fuzzing engine makes several decisions on the input to be fed to the PUT. For instance, which input seed should be mutated, and where and how to perform the mutation. State-of-the-art fuzzers' engines mostly employ random choices: they pick a random seed from the pool of interesting seeds, and they perform fixed and random mutations in a random portion of the input seed. In this work, we focus on the choice of where to apply the mutation, and we formalize the problem as a reinforcement learning problem. From the point of view of the RL agent, inputs to be mutated correspond to states; an action corresponds to choosing a position within the input and performing a set of mutations using that position as an offset; the reward corresponds to a numerical evaluation of the effectiveness of the mutations.

To better understand the rest of this work, we summarize the main concepts of reinforcement learning (Section 3.1), with a particular focus on the Proximal Policy Optimization (PPO) technique (Section 3.2) used by Rainfuzz.

3.1 Reinforcement Learning

Reinforcement learning is a branch of machine learning. It studies the problem of an agent interacting with an environment whose objective is to take actions to maximize their reward over time.

The environment state S_t^e is the environment's internal representation that it uses to produce the next reward and observation. The reinforcement learning problem is usually modeled as a Markov Decision Process (MDP), which requires that the state of the environment is fully observable by the agent (observation = S_t^e), and that S_t^e is all that the environment needs to know in order to define what are the next state and reward when the agent takes an action (independently from the history of previous states, actions, rewards). More formally:

$$MDP = \langle S, A, P, R, \gamma, \mu \rangle$$

- S is a set of states.
- A is a set of actions.
- P is a state transition probability matrix ($|S| * |A| \times |S|$), where each element $p^a_{s,s'}$ is the probability to go from state s to s' when taking action a.
- R is a reward function: $R(s,a) =$"average reward when taking action a in state s".
- γ is the discount factor ($\gamma \in [0,1]$); it is used to compute the cumulative discounted reward, defined as : $V = \sum_{t=0}^{\infty} \gamma^{t-1} r_t$.
- μ is a vector of probabilities, where each element μ_s is the probability that the initial state is s.

At each time-step t the environment will be in state $s_t \in S$, the agent will pick an action $a_t \in A$ available in state s_t, the environment will return the reward $r_t = R(s_t, a_t)$ and the environment will perform a state transition from s_t to s_{t+1} according to the state transition probability matrix P. The episode ends when the state is terminal (a state without available actions); episodes are not required to end: there might be infinite episodes. The agent chooses their actions according to a policy π; More formally, $\pi(a|s)$ is the probability of taking action a when we are in state s.

3.1.1 Solving the RL Problem

The goal of the agent is to act following a policy that maximizes its reward. When we have full knowledge about the MDP, we can compute a deterministic optimal policy (which is guaranteed to exist for all MDPs). When the characteristics of the MDP are unknown (or the size of the model makes it computationally unfeasible), the agent must learn the policy by interacting with the environment directly (model-free methods). Some model-free methods are Monte Carlo control, SARSA, Q-Learning and policy gradients.

3.1.2 Policy Gradient Methods

Policy gradients are a class of model-free methods that allows using a policy approximation as a function in the state's features and improving it as the agent gathers more information by interacting with the environment (Mnih et al., 2016), (Schulman et al., 2015), (Lillicrap et al., 2016), (Barth-Maron et al., 2018). These methods are based on the policy gradient theorem, which allows to differentiate the expected cumulative reward w.r.t. the policy parameters; this allows, for example, to compute the gradient and use it for stochastic gradient ascent.

3.2 Proximal Policy Optimization

The policy gradient theorem can be used directly to estimate the gradient of the expected reward and perform stochastic gradient ascent. Over time, more advanced objective functions have been developed to improve the learning process's performance.

PPO (Schulman et al., 2017) is a state-of-the-art family of policy gradient methods. It uses different objective functions designed to overcome some drawbacks of previous approaches. The objective function we use in this work is the clipped surrogate objective, proposed in the original PPO paper, with the addition of an entropy term; following its definition:

$$L^{CLIP+S}(\theta) =$$
$$\mathbb{E}_t[R_t * min(r_t(\theta), clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)) + T * S[\pi_\theta]]$$

where:

- R_t is the reward of episode t.
- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio.
- $S[\pi_\theta]$ is the entropy of the modelled policy.
- ε is the clip_param (an hyper-parameter).
- T is the temperature (an hyper-parameter).

Let's give an intuition of the role of each term: keeping in mind that our goal is to make our policy more likely to take high-reward functions, what we want to maximize is $\mathbb{E}[R * \pi_\theta(a|s)]$ (where the expectation is over the possible states). In order to estimate this expected value, we sample experience by following the policy itself (on-policy learning). We want to take into consideration the probability of what action is taken in order to add stability to the learning process; to reach this goal, we use a technique called importance sampling, and the expectation becomes:

$$\mathbb{E}_t[R_t * \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}] = \mathbb{E}_t[R_t * r_t(\theta)].$$

PPO uses $min(r_t(\theta), clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon))$ instead of using $r_t(\theta)$ directly: This inhibits the effect of those update steps that would make the new policy too far away from the old policy, with the goal to avoid performing destructive updates that might force the policy to sub-optimal behaviors: since the policy being learned is the same used for sampling experience, once the policy becomes too bad, it is impossible to recover from it. This is why it is important to proceed with caution and avoid greedily performing very large update steps. The clip_param (ε) is the hyper-parameter that allows defining how far the new policy is allowed to be with respect to the old policy. $T * S[\pi_\theta]$ is an entropy term used to encourage more stochastic policies. This allows us to handle the exploration-exploitation dilemma: the final goal is to obtain greater cumulative rewards when using the policy we are learning (exploitation), but since the policy used for sampling is the same used for training, it is also reasonable to leave some possibility for new behaviors to take place (exploration), to learn mechanisms that might lead to even greater rewards. The temperature (T) is the hyper-parameter defining how much low-entropy policies are discouraged.

4 RAINFUZZ

State-of-the-art gray-box fuzzers (like AFL) implement a genetic algorithm that randomly mutates inputs and keeps them in the input corpus (for future mutation) if they discover new edge-coverage in the program. Our goal is to use this coverage information not only to decide whether to keep the mutated input in the input corpus or not but also to evaluate the effectiveness of the mutation performed. This feedback about the mutation should allow learning which mutations are effective and which are not, and should allow taking more effective mutations in the future. Reinforcement learning provides a framework that allows an agent to learn to perform better by interacting with the environment, and it is a suitable choice to formalize our problem. We list the steps performed within the mutational stage of Rainfuzz, as schematized in Figure 3: ① We pick an input from the queue. ② We feed the input into the neural-network policy model. ③ The output policy constitutes a heat-map that gives a probability distribution over the positions within the current input (higher probability corresponds to a higher chance that a mutation in that position is effective). ④ We sample from the probability distribution given by the heat-map, to retrieve a specific position within the input. ⑤ We feed the input into the mutator, together with the position we sampled. ⑥ The mutator performs a predefined set of mutations at the position we just sampled, and we obtain a number of mutated inputs. ⑦ We feed the mutated inputs, one by one, into the executor; the executor runs the PUT with each one of the mutated inputs while collecting coverage information. ⑧ If one or more of the mutated inputs generate new unseen edge-coverage, we add that mutated input to the queue. ⑨ We collect the coverage information generated by each execution of the PUT, and we compare it against the coverage generated by the original un-mutated input; the result of this comparison is a numerical evaluation of the performance of the mutations at this position: the reward of the action performed. ⑩ We send the reward back to the policy model, which uses this sampled experience to train the neural network.

4.1 Learning the Policy: Proximal Policy Optimization (PPO)

A policy is a mathematical function that, given a state, returns a probability distribution over the actions available in that state. Our goal is to improve the policy over time so that it starts privileging high-reward actions (which, in our case, corresponds to performing better mutations). To carry out this learning task, we decide to use a Policy Gradient method: Proximal Policy Optimization (PPO).

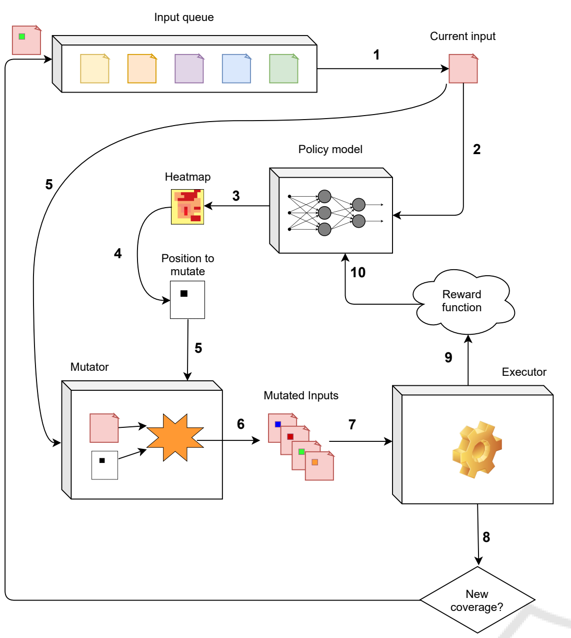

Figure 3: High-level overview of the steps involved in the fuzzing process of Rainfuzz.

4.1.1 Model of the Policy

PPO provides a method to perform learning steps over a policy model. The model we choose for the policy is a Feed Forward Neural Network (FFNN); the input of the NN is the current state of the RL agent (an array of bytes constituting the input to be mutated); the output of the NN is a probability distribution over the actions available in the current state (a heat-map indicating which byte offsets are more interesting for applying mutations). To make sure that the sum of the output probabilities is 1 we use softmax as the activation function of the output layer. Moreover, we do not allow the input to be larger than a fixed number of bytes, and we force all the mutations to be within the allowed size. This can happen, for instance, if the position of the mutation is the last byte and we want to perform a multibyte mutation (e.g., `MAX_INT`). In such a case, we simply perform the mutation and discard the overflowing bytes.

4.1.2 Learning Algorithm

To improve the performance of the policy model, we perform gradient ascent as defined in Section 3.2. We perform a single update step by using a mini-batch of sampled experience:

1. We put each sampled experience $< s_t, a_t, r_t >$ into a memory buffer;
2. If the memory is full (number of sampled experiences = M, mini-batch size), then we perform a learning step: we estimate L^{CLIP+S} by using the available experience, we differentiate it with respect to the model parameters, and then we apply the gradients to the parameters;
3. Each time we perform a learning step, we clear the memory.

4.2 Mutations

As anticipated (in Section 3), an action corresponds to performing a set of mutations at a given position within the input. We perform the following actions at position *pos*:

- assign_random_byte
- add_to_{byte, dbyte, qbyte}_{le, be}
- sub_from_{byte, dbyte, qbyte}_{le, be}
- interesting_{byte, dbyte, qbyte}_{le, be}
- clone_piece_{overwrite, insert}
- piece_insert
- delete_block

4.3 Reward Functions

The last step of our learning model is to quantify the effectiveness of the last action. To do so, we consider the difference between the coverage of the original input (c_{orig}, an array where the $i-th$ element contains the hit-count for the $i-th$ edge) and the coverage of the mutated inputs (C_{new}, an array containing the edge-coverage of the various mutated inputs generated by the last action). The general principle we follow is that when the coverage of one or more of the mutated inputs hits new edges (or has more hits on already discovered edges), then the reward should be positive. We experiment with three different ways of quantifying the effectiveness of the mutations. We report the reward function **R1** in Algorithm 1, **R2** in Algorithm 2 and **R3** in Algorithm 3.

5 EXPERIMENT EVALUATION

The goal of our experimental evaluation is to evaluate several aspects of Rainfuzz and its approach, answering the following research questions:

- **RQ1:** Does Rainfuzz's policy outperform the random policy?
- **RQ2:** Which amount of randomness in the action taken is ideal for finding the most edges over time?

```
input: c_orig, C_new
output: reward

diffs = []
for c_i in C_new:
    diff = 0
    for k in [1,2,...,num_edges]:
        if c_i[k] > c_orig[k]:
            diff += c_i[k] - c_orig[k]
    diffs.append(diff)
return average(diffs)
```

Algorithm 1: Pseudocode of reward function R1.

```
input: c_orig, C_new
output: reward

tots = []
for c_i in C_new:
    tot = 0
    for k in [1,2,...,num_edges]:
        if c_i[k] > c_orig[k]:
            tot += 1
    tots.append(tot)
return average(tots)
```

Algorithm 2: Pseudocode of reward function R2.

```
input: c_orig, C_new
output: reward

tots = []
for c_i in C_new:
    tot = 0
    for k in [1,2,...,num_edges]:
        if c_orig[k] == 0 and c_i[k] > 0 :
            tot += 1
    tots.append(tot)
return max(tots)
```

Algorithm 3: Pseudocode of reward function R3.

- **RQ3:** Which Reward Function among the ones we designed performs best?
- **RQ4:** What is the overhead introduced for generating mutations following Rainfuzz's reinforcement learning policy?
- **RQ5:** How does Rainfuzz perform with respect to AFL++?
- **RQ6:** Can the union of AFL++ and Rainfuzz (running in a collaborative fuzzing setting) outperform two AFL++ instances running in parallel?
- **RQ7:** Is the configuration of Rainfuzz we tuned against `libjpeg-turbo` still effective if the PUT changes?

Throughout our experiments, we use `libjpeg-turbo` as a PUT, a binary taken from *FuzzBench*, a fuzzing benchmarking framework developed by Google to unify fuzzing evaluation. We use a different binary in **RQ7** to confirm the results obtained. We tune the policy model by going through a hyper-parameter tuning phase; in this phase, we run experiments where the reinforcement learning policy and the random policy co-live; we use the difference between the average reward generated by the reinforcement learning policy and the one generated by the random policy as a metric to decide which configuration is best. We report the best-performing configuration among the ones we tested.

- Activation function for the intermediate layers of the NN: **tanh**
- Number of intermediate layers for the NN: **1**
- Number of neurons for each intermediate layer of the NN: **128**
- Learning rate for the stochastic gradient ascent update: **0.0001**
- Mini-batch size: **50**
- Clip hyper-parameter of the clipped surrogate loss function: **0.5**
- Temperature hyper-parameter of the entropy term in the loss function: **3.0**

We also decide to introduce an amount of actions to be taken randomly; in **RQ2**, we discover that 75% is the percentage that works best while, in **RQ3**, we find that the best-performing reward function is **R1**. The resulting configuration is what we call a tuned version of Rainfuzz.

RQ1: Does Rainfuzz's Policy Outperform the Random Policy? First, we run three 24H long experiments for each reward function we designed. We discover that for all three reward functions, the reinforcement learning policy always outperforms the random policy in terms of average reward. We report the plot of the average-reward signals for **R1** as a sample in Figure 4. We are also interested in assessing the effectiveness of Rainfuzz in terms of edge-coverage over time (this is the ultimate metric we use to determine the effectiveness of two fuzzing approaches). We run three 24H experiments using the tuned Rainfuzz, and three 24H experiments using an equivalent fuzzer that uses the random policy; we plot the resulting average edge-coverage in Figure 5.

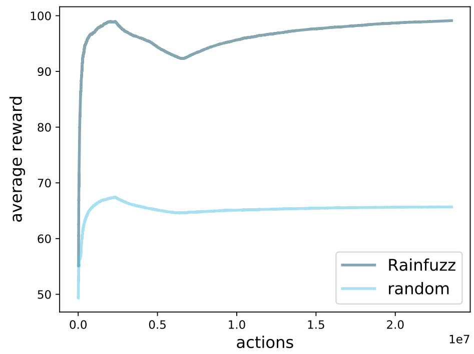

Figure 4: Average rewards generated by R1.

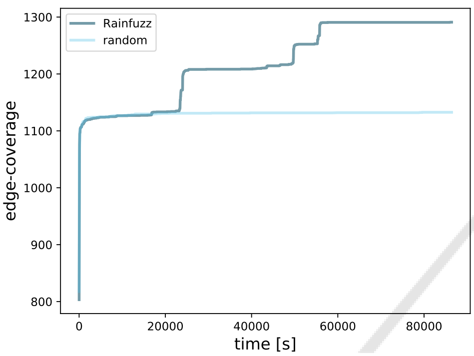

Figure 5: Average edge-coverage generated by Rainfuzz and by the random policy.

As we can see, the reinforcement learning policy outperforms the random policy by generating an average edge-coverage of 1277 against 1133.

RQ2: Which Amount of Randomness in the Action Taken Is Ideal for Finding the Most Edges over Time? We run three 24H long experiments for each amount of randomness, using **R2** as a reward function. We plot the average edge-coverage in Figure 6. As we observe, 75% randomness outperforms the other configurations by reaching an average edge-

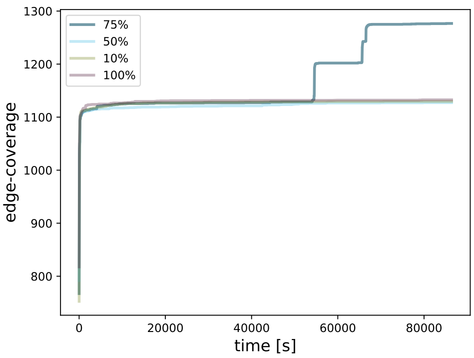

Figure 6: Average edge-coverage generated by each amount of randomness.

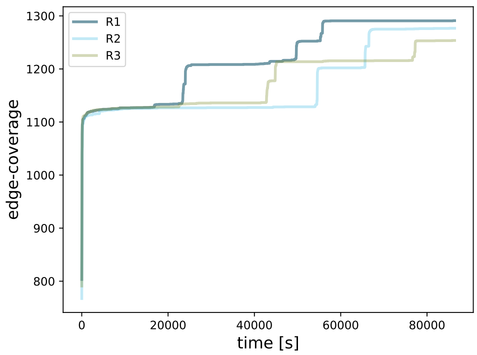

Figure 7: Average edge-coverage generated by each reward function.

coverage of 1277 (against 1130 for 10%, 1120 for 25% and 1133 for 100%).

RQ3: Which Reward Function Among the Ones We Designed Performs Best? We run three 24H long experiments for each reward function we designed and we plot the average edge-coverage in Figure 7. As we observe, **R1** outperforms the other configurations by reaching an average edge-coverage of 1291 (against 1277 for R2 and 1254 for R3).

RQ4: What Is the Overhead Introduced for Generating Mutations Following Rainfuzz's Reinforcement Learning Policy? We measure the number of times the fuzzer executes the PUT per unit of time. For the random policy we observe an execution speed of 10624 $execs/sec$, for Rainfuzz (75% randomness) we observe 5541 $exec/sec$, while for a completely reinforcement learning policy (0% randomness) we observe 2255 $exec/sec$. As we can see, when following the reinforcement learning policy, we execute the PUT at a speed $4,71$ lower w.r.t. a completely random policy: this is the cost introduced by the need of querying the policy model and training it. The tuned version of Rainfuzz has an execution speed that is $1,92$ times lower than the random policy, but our experimental evaluation (**RQ1**) shows that the quality of the actions picked compensates the overhead of choosing them.

RQ5: How Does Rainfuzz Perform with Respect to AFL++? We want to compare Rainfuzz against a real-world fuzzer (AFL++). Rainfuzz needs to restrict the size of the inputs generated when performing mutations because they need to fit into the neural network maximum size. In order to understand the impact of this restriction, we build aflpp_mod, a version of AFL++ that restricts the size of inputs just like Rainfuzz. We run three 24H long experiments

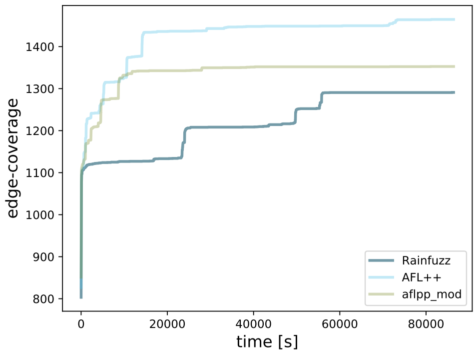

Figure 8: Average edge-coverage generated by Rainfuzz, aflpp_mod, AFL++.

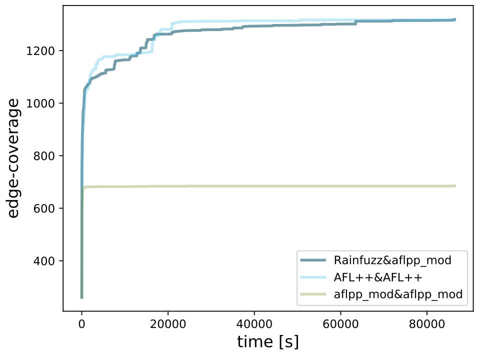

Figure 10: Average edge-coverage generated by Rainfuzz, aflpp_mod, AFL++.

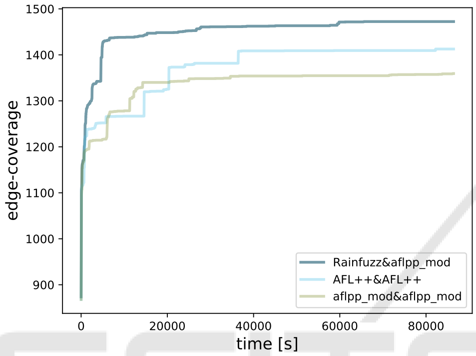

Figure 9: Average edge-coverage generated by Rainfuzz&aflpp_mod, aflpp_mod&aflpp_mod, AFL++&AFL++

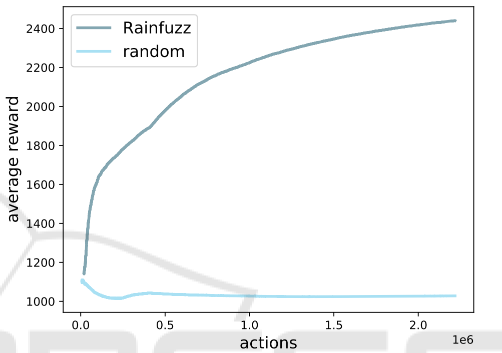

Figure 11: Average reward generated by Rainfuzz, using `file` as PUT.

for Rainfuzz, aflpp_mod and AFL++; we plot the average edge-coverage in Figure 8. AFL++ generates an average edge-coverage of 1465, aflpp_mod 1352 and Rainfuzz 1291. The fact that AFL++ outperforms aflpp_mod shows the negative impact of restricting input size. Rainfuzz is outperformed by both AFL++ and aflpp_mod, but we find a more interesting result in **RQ6**.

RQ6: Can the Union of AFL++ and Rainfuzz (Running in a Collaborative Fuzzing Setting) Outperform Two AFL++ Instances Running in Parallel?. Collaborative fuzzing is a technique that consists in running instances of different fuzzers in parallel; this approach is often capable of exploiting the strengths of different fuzzing approaches (Güler et al., 2020). We refer to a setting where an instance of a fuzzer F1 is ruan in parallel with F2 with F1&F2. We run three 24H long experiments for Rainfuzz&aflpp_mod, aflpp_mod&aflpp_mod, AFL++&AFL++; we plot the average edge-coverage in Figure 9. Rainfuzz&aflpp_mod generates an average edge-coverage of 1473, AFL++&AFL++ 1414 and aflpp_mod&aflpp_mod 1359.

RQ7: Is the Configuration of Rainfuzz We Tuned Against `Libjpeg-Turbo` Still Effective if the PUT Changes?. We are interested in finding out if the tuning phase we did is robust, and can still be effective if the PUT changes; we repeat the same set of experiments we ran for **RQ6** but using `file` as PUT, Figure 10 shows the results. Rainfuzz&aflpp_mod generates an average edge-coverage of 1319, AFL++&AFL++ 1317, aflpp_mod&aflpp_mod 685. As we can observe, Rainfuzz&aflpp_mod still outperforms the other two configurations, but just by a very small amount.

We are also interested in visualizing the effectiveness of the reinforcement learning policy over the random policy. We take the rewards generated by the Rainfuzz instance of Rainfuzz&aflpp_mod. We plot them in Figure 11.

5.1 Threats to Validity

Edge-coverage over time is a metric that is subject to a discreet amount of variance, and results may vary a lot due to random chance. For this reason, we repeated our experiment three times, and we analyzed the average edge-coverage observed to draw our conclusions. To definitely confirm the results of our work, it's probably necessary to repeat experiments more than three times. Moreover, we analyzed the robustness of the tuned version of Rainfuzz in **RQ7**, by observing how Rainfuzz performs against a different PUT then `jibjpeg-turbo`: `file`. To confirm the robustness of Rainfuzz it is probably necessary to experiment against a much larger variety of PUTs.

6 RELATED WORKS

Previous attempts tried to use different machine learning techniques to build heat-maps. In (Rajpal et al., 2017) the authors experiment with neural network models capable of predicting heat-maps given an input. Data on the effectiveness of mutations is collected by running a standard gray-box fuzzer (AFL), and then the neural network is trained using that data, in a supervised learning setting. The model is then used to predict what bytes are useful to mutate, and mutations that don't stress those bytes are vetoed. In (She et al., 2019) a neural network model is used to predict the resulting edge coverage given the input. An adversarial machine learning technique is then used, to detect the input byte with the highest gradient associated with it. This is equivalent to detect the byte that, if mutated, has the highest probability to cause a change in the output coverage predicted by the model; if the model is accurate enough, this change should also be reflected in the coverage of the actual program. This byte is then used as an offset to perform a number of mutations. Both these approaches have the drawback that they must be preceded by a phase where data is collected and used for training a model. As the fuzzing process goes on, new program behaviours are discovered, and the model gets quickly outdated; since this approaches are based on the effectiveness of the model, a new training phase needs to be taken in order to update the model. This alternation between training and fuzzing phases introduces significant overhead.

Also reinforcement learning has been applied to fuzzing. In (Böttinger et al., 2018) the authors explore the possibility of making mutations more efficient by modelling the fuzzing process as a full reinforcement learning problem:

- **inputs** are the states of the MDP.
- an **action** corresponds to randomly select a substring within the input, and to perform a single mutation on such a sub-string. The mutation is chosen accross several available mutations (e.g., `Delete`, `Shuffle`, `Random bit-flips`, etc.).
- They experiment with two types of **reward**: *Discovered Blocks* and *Execution time*.

The technique used to solve the reinforcement learning problem is deep Q-learning: the deep Q-network observes a portion of the input i (the sub-string s'), and estimates the value function $Q(s',a)$, for each action a; they use an $\varepsilon - greedy$ policy to choose the next action to take. Finally, the experimental evaluation compares the approach against a baseline created using the random policy. The metric they use is the cumulative reward generated by the two policies, proving that the reinforcement learning policy chooses higher reward functions. However, this evaluation has a limitation. The metric used during experimental evaluation shows an interesting theoretical result, but does not provide evidence of the effectiveness of the approach in a real-world scenario: overheads introduced by the reinforcement learning approach might defeat the purpose; *edge-coverage over time* is the right metric to use if we want to test the effectiveness of a fuzzer in a real-world scenario. For completeness we cite (Zhang et al., 2020), another approach that uses reinforcement learning in fuzzing. The formalization of the problem is very similar to the one used in (Böttinger et al., 2018). The main difference is related to the algorithm they use for solving the reinforcement learning problem, an actor-critic technique: *Deep Deterministic Policy Gradient*. Their experimental evaluation explores many hyper-parameters combinations, but ultimately uses a metric similar to the one used in (Böttinger et al., 2018), proving a result that is theoretically interesting, but with no direct impact on real-world fuzzing.

7 FUTURE WORKS

A key role in Rainfuzz is played by the reward function, which evaluates the effectiveness of the mutations taken in a given position within the input. In our implementation, we experiment with three reward functions, but there is space for more approaches. We propose the creation of a reward function that weights edges differently based on their rarity: a mutation that allows increasing the hit count on edges that are not seen very often should be rewarded more than a mutation that allows increasing the hit count on edges that are already stressed very frequently by the current input-corpus. This idea of taking into account the rarity of edges was already explored in the context of seed scheduling (Böhme et al., 2016) with great results; we believe that shifting this concept in the context of rewarding mutations is very promising.

The NN architecture we use in Rainfuzz has a fixed input size, forcing us to restrict the size of mutated inputs. There is space for experimentation to overcome this issue by using different NN architectures. Probably recurrent NNs are a suitable choice, but it faces the challenge of modeling a variable-length policy.

An important component of Rainfuzz is the set of position-specific mutations (Section 4.2) corresponding to a single action. The mutations we use are inspired by random-position mutations that are used within AFL++; it might be interesting to experiment with different sets of position-specific mutations and study how they influence the performance of fuzzing based on the input format of the PUT.

8 CONCLUSIONS

In this paper, we propose an innovative fuzzing approach that builds heat-maps using reinforcement learning, aiding the mutation strategy and overcoming the issue of alternating training phases to fuzzing phases. We implemented our approach by means of Rainfuzz, and we tuned it by trying different configurations (**RQ2**, **RQ3**). We tested the validity of our approach (**RQ1**) by comparing Rainfuzz against an equivalent fuzzer that uses a fully random policy, showing that Rainfuzz performs better both in terms of average reward per action and in terms of edge-coverage. We tested Rainfuzz against a state-of-the-art fuzzer (AFL++), with poor results (**RQ5**); but we showed that Rainfuzz and AFL++ running in a collaborative fuzzing setting obtain the best performance (**RQ6**). We confirmed the robustness of Rainfuzz by showing that the previous results still apply if the PUT changes (**RQ7**). Finally, we concluded by providing some ideas to extend and improve the approach we proposed.

ACKNOWLEDGEMENTS

Lorenzo Binosi acknowledges support from TIM S.p.A. through the PhD scholarship.

REFERENCES

Barth-Maron, G., Hoffman, M. W., Budden, D., Dabney, W., Horgan, D., TB, D., Muldal, A., Heess, N., and Lillicrap, T. P. (2018). Distributed distributional deterministic policy gradients. *CoRR*, abs/1804.08617.

Böhme, M., Pham, V., and Roychoudhury, A. (2016). Coverage-based greybox fuzzing as markov chain. In Weippl, E. R., Katzenbeisser, S., Kruegel, C., Myers, A. C., and Halevi, S., editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1032–1043. ACM.

Böttinger, K., Godefroid, P., and Singh, R. (2018). Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*, pages 116–122. IEEE Computer Society.

Chen, C., Cui, B., Ma, J., Wu, R., Guo, J., and Liu, W. (2018). A systematic review of fuzzing techniques. *Comput. Secur.*, 75:118–137.

Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. (2020). AFL++ : Combining incremental steps of fuzzing research. In Yarom, Y. and Zennou, S., editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association.

Google (2016). HonggFuzz. https://honggfuzz.dev/.

Güler, E., Görz, P., Geretto, E., Jemmett, A., Österlund, S., Bos, H., Giuffrida, C., and Holz, T. (2020). Cupid : Automatic fuzzer selection for collaborative fuzzing. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, pages 360–372. ACM.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. In Bengio, Y. and LeCun, Y., editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

LLVM (2017). libFuzzer. http://llvm.org/docs/LibFuzzer.html.

Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. (2021). The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Software Eng.*, 47(11):2312–2331.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783.

Rajpal, M., Blum, W., and Singh, R. (2017). Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR*, abs/1711.04596.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015). Trust region policy optimization. *CoRR*, abs/1502.05477.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

She, D., Pei, K., Epstein, D., Yang, J., Ray, B., and Jana, S. (2019). NEUZZ: efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 803–817. IEEE.

Wang, J., Duan, Y., Song, W., Yin, H., and Song, C. (2019a). Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pages 1–15. USENIX Association.

Wang, Y., Jia, P., Liu, L., and Liu, J. (2019b). A systematic review of fuzzing based on machine learning techniques. *CoRR*, abs/1908.01262.

Zalewski, M. (2016). AFL: American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt.

Zhang, Z., Cui, B., and Chen, C. (2020). Reinforcement learning-based fuzzing technology. In Barolli, L., Poniszewska-Maranda, A., and Park, H., editors, *Innovative Mobile and Internet Services in Ubiquitous Computing - Proceedings of the 14th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2020), Lodz, Poland, 1-3 July, 2020*, volume 1195 of *Advances in Intelligent Systems and Computing*, pages 244–253. Springer.