RLGFuzz: Reinforcement Learning Guided Fuzzing with State-Coverage Mapping Environment

Yanlong Shen, Yu Liu, Ying Zhou*
School of Electronics and Communication Engineering, Sun Yat-Sen University
Shenzhen, China
shenylong@mail2.sysu.edu.cn, liuy2533@mail2.sysu.edu.cn, zhouying5@mail.sysu.edu.cn

***Abstract*—To ensure the security and reliability of protocol implementations, protocol fuzzing serves as a crucial technique of security assessment and is widely adopted. However, existing fuzzers exhibit low efficiency after a certain duration of testing, primarily due to the lack of more flexible and adaptive guidance in seeds and states selection. In response to this issue, we propose a reinforcement learning guided fuzzer named RLGFuzz. Neural networks are employed to learn potential mappings between seeds and states to path coverage, which are then utilized in an interactive environment for reinforcement learning. This process allows the fuzzer to preferentially test seeds and states. Experimental results on a diverse set of real-world protocols demonstrate significant improvements in path coverage and triggering crashes efficiency achieved by our RLGFuzz. Compared to AFLNet, AFLNwe, and SMGFuzz, RLGFuzz outperforms all three, achieving a 6.68% enhancement in path coverage and a 5.52% enhancement in efficiency of triggering crashes.**

***Index Terms*—Protocol Fuzzing, Greybox Fuzzing, Reinforcement Learning**

I. Introduction

With the rapid development of the Internet and the communication industry, the demands for data rate, communication range, and energy consumption are increasingly higher [1], various protocols play crucial roles. However, protocol implementations represent one of the most exposed components in modern Internet systems. Due to the complexity of the protocol description and the ever-changing network environment, there are various security vulnerabilities and errors in the servers in which the protocol is implemented. In order to ensure the security and reliability of protocols, protocol fuzzing, as an important security assessment technique, is widely applied.

Protocol fuzzing involves sending protocol messages with random variations to test the protocol implementation of the target system, simulating attackers sending abnormal or unexpected protocol messages to trigger potential vulnerabilities or errors within the target system. By observing the responses of the target system, abnormal behaviors such as crashes, denial of service, memory leaks, can be detected. Traditional protocol fuzzing methods, such as state coverage-guided AFLNet [2], AFLlegion [3], etc., have been successful in finding vulnerabilities by using randomly generated and mutated test cases as inputs. In recent years, research on protocol fuzzing based on self-learning models and large language models has gradually emerged. Techniques such as GANFuzz [4], ChatAFL [5], etc., generate more effective test cases by learning the syntax and semantic features of protocols, aiming to improve code coverage and state space coverage [6]. These advancements facilitate better discovery of potential vulnerabilities and errors within protocols.

979-8-3503-5171-2/24/$31.00 ©2024 IEEE

However, whether traditional or emerging, fuzzing methods lack flexibility and adaptability in the later stages of fuzzing. In fact, it has been demonstrated that over prolonged periods of fuzzing [7], this deficiency is further magnified: extended mutation leads to excessively long message sequences, which are ineffective in triggering new code basic block path coverage. Instead, much time and computational resources are wasted in achieving selected states. The objective of this paper is to identify states with the potential to trigger new path coverage, thereby enhancing the efficiency of network protocol testing.

It is hypothesized that there exists a high-dimensional, implicit mapping between seed files and selected states and path coverage. The seed file of interest, the selected state and the actual triggered path coverage serve as training data, which is then utilized to train a deep neural network in order to learn a mapping between seeds, states and path coverage. The learned mapping is employed in the interactive environment of D3QN to supplant the conventional heuristic function, whereby it presents a multitude of opportunities for states with the potential to trigger path coverage through reinforcement learning. This approach will facilitate the generation of more efficient seed files and states in the middle and late stages of fuzzy testing, thereby enhancing the path coverage of the protocol implementation program and prompting the occurrence of more crashes and potential bugs. The main contributions of our work are summarized as follows:

- We design a neural network model for learning potential mappings of seeds and states to path coverages.
- We construct a virtual environment that simulates different seed and selected state triggering code coverage, and use reinforcement learning to accomplish seeds and states preferences.
- We implemented a double-threaded reinforcement learning guided fuzzing pipeline and deployed experiments on four Systems Under Test (SUTs), comparing to AFLNet [2], AFLNwe [8], and SMGFuzz [7], with an improvement of 6.68% in terms of path coverage, and an improvement of 5.52% in terms of efficiency in triggering crashes over the best of them.

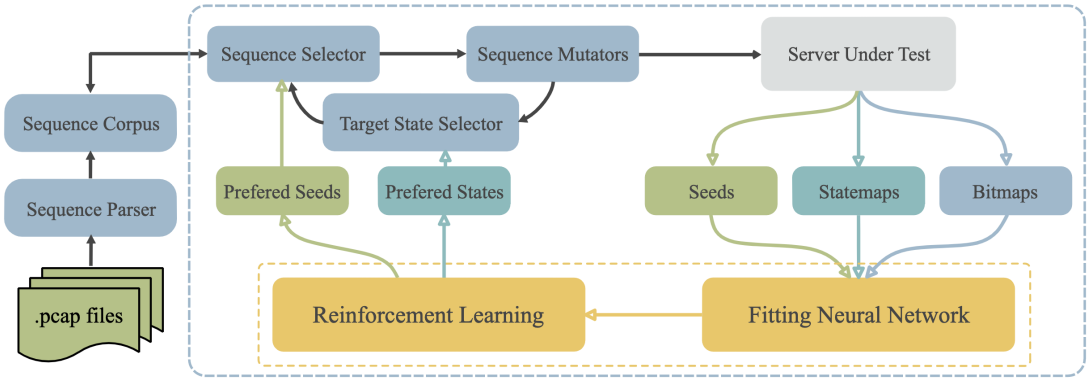


Fig. 1. System Overview.

II. Related Work

Our work is influenced by stateful coverage-based greybox fuzzing (SCGF) and reinforcement learning (RL).

A. Stateful Coverage-based Greybox Fuzzing

Stateful coverage-based greybox fuzzing, e.g., AFLNet [2] guides fuzzing test through state feedback, and in the process of fuzzing test, an implemented protocol state machine (IPSM) is continuously improved based on the feedback from the protocol, and accordingly guides the fuzzing test framework for seed mutation and state selection. AFLNET utilizes a heuristic function, as illustrated in the following equation, to allocate energy to states.

$$\texttt{Energy}(s) \propto \frac{\texttt{Path}(s)}{\texttt{Tested}(s)} \tag{1}$$

The energy is proportional to the number of paths $\texttt{Path}(s)$ that have been triggered by that state, and inversely proportional to the number of times $\texttt{Tested}(s)$ that state has been tested. Although the equation was generalized by experienced predecessors, it results in an unreasonable allocation of minimal energy to numerous states that have the potential to trigger new path coverage when testing network protocols with a massive amount of states.

As a further optimization of AFLNET, AFLNetLegion [3] proposes a Monte Carlo tree search based algorithm for inspiring state selection. However, the more complex heuristic algorithm did not result in a significant improvement in the efficiency of the state selection algorithm. The authors suggest that this may be caused by insufficiently fine-grained model state identification and a throughput that is too low to unlock the full performance of the state selection algorithm.

Our work builds upon the SCGF framework by introducing improvements. The traditional sequence selector and state selector of SCGF are replaced with reinforcement learning, which enables the preference of seed files and states.

B. Reinforcement Learning

Q-learning [9] is a basic reinforcement learning algorithm based on the Q-value function, i.e., action-utility function, which guides the decision-making of an intelligent body by means of a randomized greedy algorithm (ε-greedy), and after iterating for many rounds, it obtains a complete Q-Table. Deep Q-Network (DQN) [10] improves the learning capability of complex environments in massive state space by means of a neural networks to approximate the Q-value function to improve the learning ability for complex environments with massive state spaces, and introduces a replay buffer to avoid the correlation problem between experimental data. Double DQN [11] uses two neural networks, one for selecting the optimal action and the other for evaluating the value of that action, which mitigates the overestimation bias of the Q-value function in DQN. Dueling DQN [12] improves the ability to evaluate different actions by decomposing the Q-value function into a value function and a dominance function to better learn the value information of the state. Dueling Double DQN (D3QN) [13] combines the improvements of the previous both, with faster convergence and more stability.

In addition, previous projects combining reinforcement learning and fuzzing have made excellent progress. RL-FUZZ [14] models the process of traditional fuzzing as a Markov decision process and utilizes reinforcement learning algorithms to guide the direction of each step in the mutation process, using path coverage as a reward and DDPG [15] algorithms for optimization in order to improve the quality of samples and the efficiency of fuzzing. AFL-HIER [16] improves coverage by introducing a new concept of multilevel coverage metric and a corresponding hierarchical scheduler based on reinforcement learning.

The D3QN reinforcement learning algorithm, which exhibits a robust capacity for generalization, is employed to prioritize test states, thereby facilitating the identification of a greater number of potential states for evaluation. Concurrently, the elevated stability and rapid convergence capacity of D3QN ensure the high caliber of the selected seeds and states. Therefore, in this paper, we implement the definition of key elements of reinforcement learning in combination with the characteristics of protocols testing, and utilize D3QN to achieve the preferential selection of seeds and states.

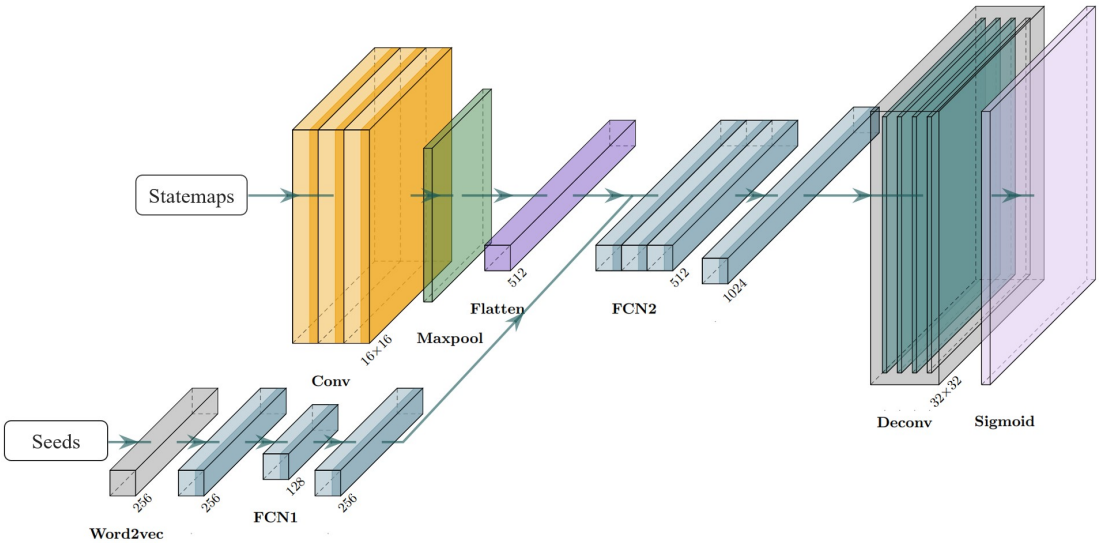


Fig. 2. Architecture of the Coverage Prediction Network.

III. Architecture Design

The system block diagram, depicted in Fig. 1, provides a visual representation of our work. During the execution of SCGF, a dataset is generated that records the seed files that can effectively trigger path coverage, the selected states (`statemaps`) and the actual triggered path coverage (`bitmaps`) over a period of time. A deep neural network is utilized to learn them, and the mappings obtained from this process are employed in the interactive environment for reinforcement learning. The seed files and the states to be selected are selected by reinforcement learning and are used to guide the state selection of the SCGF.

A. Data Acquisition

During the fuzzing process of the SCGF, seeds that trigger new path coverage will be added to the seed queue for subsequent utilization. Seed files datasets originate from the seeds queues. Upon the addition of seed to the queue, the fuzzer parses the message sequence and response sequence, and incorporates the new states into the IPSM of the SUT.

Let $G_s = (V_s, E_s)$ be the graph describing the state transition relations in the IPSM of SUT, where the vertex set $V_s = \{v_{s1}, v_{s2}, \ldots, v_{sn}\}$ represents a total of n states abstracted by the IPSM, and the edge set $E_s = \{e_{s1}, e_{s2}, \ldots, e_{s\varepsilon}\}$ represents the transitions between states, and s_{ij} represents whether a state transition has occurred between state v_{si} and state v_{sj}, and takes the value of 0 or 1. `Statemap` is the adjacency matrix that records the IPSM state transfers.

$$\texttt{statemap} = (s_{ij})_{n\times n} \tag{2}$$

Edges between basic blocks of a program can also be modeled by a directed graph, let $G_b = (V_b, E_b)$ be the graph describing the coverage of the basic blocks of the code after the program has been run, where the set of vertices $V_b = \{v_{b1}, v_{b2}, \ldots, v_{bp}\}$ represents the program with a total of p basic blocks, the set of edges $E_b = \{e_{b1}, e_{b2}, \ldots, e_{b\beta}\}$ represents the jumps between basic blocks, and b_{ij} represents whether a jump occurs between the basic block v_{bi} and basic block v_{bj}, and takes the value of 0 or 1. `Bitmap` is the adjacency matrix that records the edge coverage of the basic blocks of the program.

$$\texttt{bitmap} = (b_{ij})_{p\times p} \tag{3}$$

Both `bitmaps` and `statemaps` are sparse, and their sizes vary for different SUTs. Therefore, we introduce hash functions to remap them, standardizing their sizes for subsequent processing. Although this remapping results in the loss of their original meanings, the hash functions are pseudorandom and possess unidirectional repeatability. In accordance with the data structure design in C code and our experience, the size of the processed `statemaps` is 16×16, and the size of the `bitmaps` is 256×256.

$$\texttt{statemap}_{16\times16} = \text{hash}(\texttt{statemap}_{n\times n}) \tag{4}$$

$$\texttt{bitmap}_{256\times256} = \text{hash}(\texttt{bitmap}_{p\times p}) \tag{5}$$

B. Coverage Prediction Based on Deep Learning

The neural network architecture, as illustrated in Fig. 2, is used to learn the mapping between seed files, selected states, and the actual triggered path coverage. Considering the data types to be processed include matrices and vectors, we adopted a hybrid architecture consisting of Convolutional Neural Networks (CNNs) [17] and fully connected neural networks (FCNs) [18]. This serves as a predictor, taking different seed files and selected states as input, and predicting the possible triggered path coverage.

$$\texttt{bitmaps} = \mathcal{F}(\texttt{seeds}, \texttt{statemaps}) \tag{6}$$

First, the seed files in the candidate set $\mathcal{C}$ of mutated seeds are preprocessed using the `word2vec` model [19] to encode

them into 256-dimensional word vectors $\boldsymbol{v}_{\mathcal{C}}^{256}$. Simultaneously, the statemaps in the collection $\mathcal{M}$ recording the changes in SUT states are flattened after through CNNs to obtain 256-dimensional feature vectors $\boldsymbol{v}_{\mathcal{M}}{}^{256}$. CNNs are employed for rapid training and considering the local correlations in `statemaps`. Subsequently, the word vectors and feature vectors are merged to yield 512-dimensional feature vectors $\boldsymbol{v}_r^{512}$, which are then passed through FCNs, denoted as the nonlinear mapping layer $\mathcal{F}_{\boldsymbol{\Theta}}(\cdot)$. Finally, the output of the nonlinear mapping

$$\boldsymbol{v}_r'{}^{1024} = \mathcal{F}_{\boldsymbol{\Theta}}\left(\boldsymbol{v}_r^{512}\right) \tag{7}$$

is restructured as a 32×32 matrix $\mathbf{B}^{32}$, and expanded to the same size as `bitmaps` through a deconvolutional layer. The network predicts the values of the `bitmap` as $\mathbf{B}^{256}$. Our primary concern is the probability of elements in $\mathbf{B}^{256}$ being "1", which implies that new edges are covered.

Reference [20] mentions that in binary classification problems, it is more appropriate to choose the `Sigmoid` function for the activation function of the output layer, whereas the `Softmax` function gives rise to structural redundancy. Furthermore, the activation functions for the remaining layers are set to the `ReLU` function.

Binary cross entropy(BCE) is selected as the loss function,

$$\begin{aligned} L(\mathbf{B}, \mathbf{B}^*) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{256} \sum_{k=1}^{256} \Big[{B_{j,k}^i}^* \log\left(B_{j,k}^i\right) \\ + \left(1 - {B_{j,k}^i}^*\right) \log\left(1 - B_{j,k}^i\right) \Big] \end{aligned} \tag{8}$$

where $\mathbf{B}$ denotes the predicted value of the bitmap, $\mathbf{B}^*$ denotes the true value of the bitmap, N denotes the number of samples, i denotes the sample ordinal number, and j, k denote the subscripts of the matrix element positions.

C. Preference for Seeds and States

Considering that the state space S of the designed system consists of whether each element in the `bitmap` is "1", with up to 2^{65536} possible configurations, the state space is massive, necessitating the use of reinforcement learning methods. Algorithm 1 is implemented based on the fundamental framework of the D3QN [13] reinforcement learning algorithm. Taking into account the characteristics of protocol fuzzing, several key elements of reinforcement learning are defined as follows [14], [16]:

- **Environment**: Using the mapping $\mathcal{F}_{\boldsymbol{\Theta}}(\cdot)$ as a virtual environment for reinforcement learning;
- **State Space**: All possibilities for elements at each position of the bitmap,

$$S = [0, 1]^{65536}; \tag{9}$$

- **Action Space**: The Cartesian product of the power sets of the set of seed documents $\mathcal{C}$ and the set of optional states,

$$A = 2^{\mathcal{M}} \times \mathcal{C}, \tag{10}$$

$$\text{card}(A) = 2^{\text{card}(\mathcal{M})} \times \text{card}(\mathcal{C}); \tag{11}$$

- **Current State**: A predicted possible bitmap, $s \in S$;
- **Current Action**: Select a seed file from the set of alternative seed files and generate a possible statemap, $a \in A$;
- **Reward**: Calculated based on the new path coverage added globally in the current iteration round,

$$r(t) = f_{\text{offset}}(t) \times \sum_{i=1}^{256} \sum_{j=1}^{256} B_{i,j}^t \cap {B_{\text{total}}}_{i,j}^{C} \tag{12}$$

where t denotes the current iteration round, i, j denote the indices of matrix elements, B_{total} denotes the total paths covered globally, and its complement represents the paths globally uncovered. $f_{\text{offset}}(t)$ denotes the offset function used to compensate for the phenomenon of decreased sensitivity to coverage changes of newly added paths with increasing rounds. It is simulated using the Monte Carlo algorithm, and the coverage of newly added paths decreases exponentially with the increase of rounds:

$$\Delta B_{\text{total}} = ae^{-bt} \tag{13}$$

To counteract this decay and ensure fairness in rewards for each round, an offset function is designed,

$$f_{\text{offset}}(t) = \frac{1}{a} e^{bt} \tag{14}$$

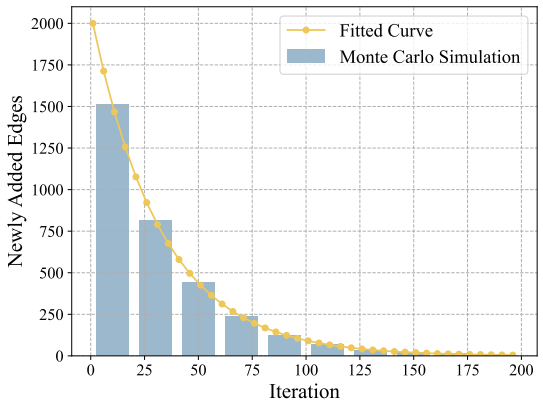


Fig. 3. Monte Carlo Simulation Results. As a consequence of the memoryless, repetitive, random addition of edges, the newly added edges and the number of iterations are distributed according to the Poisson distribution. The coefficients are obtained by Monte Carlo simulation.

Algorithm 1 specifies the input as the candidate seed file set $\mathcal{C}$, the collection of SUT state changes $\mathcal{M}$, the trained mapping $\mathcal{F}_{\boldsymbol{\Theta}}(\cdot)$, the number of iterations T, the decay rate γ, the exploration rate ε, and the update frequency p. The output is defined as the seeds triggering new path coverage and the corresponding selected state set $\mathcal{O}_t$. Lines 1-10 perform a series of initialization operations, where $done$ indicates whether this iteration has ended, $count$ records the number of invalid actions, and $flag$ indicates whether an action is worth saving. Lines 12-19 utilize the ε-greedy algorithm to select actions. With a certain probability, the greedy algorithm is used

Algorithm 1 Preference for Seeds and States.

Input: seeds set $\mathcal{C}$, statemaps set $\mathcal{M}$, $\mathcal{F}(\cdot)$, decay rate γ, exploration rate ε, update frequency p, number of iterations T

Output: The tuple list of seeds that trigger new coverage and their corresponding selected states $\mathcal{O}_t$

1: Initialize the networks' parameter θ and θ'
2: Replay Buffer $\mathcal{D} \leftarrow 0$
3: Global Coverage Map $B_{\text{total}} \leftarrow 0$
4: $\mathcal{O}_t \leftarrow \phi$
5: **for** i from 1 to T **do**
6: Reward $r \leftarrow 0$
7: $done, flag \leftarrow$ False
8: $count \leftarrow 0$
9: **while** True **do**
10: **if** Random()$< \varepsilon$ **then**
11: $a \leftarrow \text{argmax}_a Q(s, a; \theta)$
12: **else**
13: $a_1 \leftarrow$ Randint (card($\mathcal{M}$))
14: $a_2 \leftarrow$ ChooseState($\mathcal{C}, a_1$)
15: $a \leftarrow < a_1, a_2 >$
16: Current State $s \leftarrow \mathcal{F}(a)$
17: **end if**
18: $seed \leftarrow$ChooseSeed($\mathcal{C}, a_1$)
19: $statemap \leftarrow$ChooseStatemap($\mathcal{M}, a_2$)
20: $s' \leftarrow \mathcal{F}_{\mathbf{\Theta}}$(seed, statemap)
21: $r \leftarrow$ CalculateReward(s', B_{total})
22: $B_{\text{total}} \leftarrow B_{\text{total}} \cup s'$
23: **if** $r \leq 0$ **then**
24: $count \leftarrow count$+1
25: **if** $count > Threshold$ **then**
26: $done \leftarrow$True
27: **end if**
28: **else**
29: $flag \leftarrow$True
30: **end if**
31: Update Replay Buffer $\mathcal{D} \leftarrow (s, s', a, r, \text{done}) \cup \mathcal{D}$
32: **if** $done$==True **then**
33: break
34: **end if**
35: **end while**
36: Sample m tuples $(s, s', r, a, \text{done}) \sim \text{Unif}(D)$
37: $a^{\max}(s'; \theta) \leftarrow \arg\max_{a'} Q(s', a'; \theta)$
38: $y_j \leftarrow r + \gamma\left[V(s'; \theta') + A\left(s', a^{\max}(s'; \theta); \theta'\right)\right]$
39: $loss \leftarrow \frac{1}{m}\sum_{j=1}^{m}\left\|y_j - Q(s_j, a_j; \theta)\right\|^2$
40: Updating parameters $\theta \leftarrow$ optimize($\theta, loss$)
41: **if** mod(i, p)==0 **then**
42: $\theta' \leftarrow \theta$
43: **end if**
44: **if** $flag$==True **then**
45: $\mathcal{O}_t \leftarrow \mathcal{O}_t \cup < seed, statemap >$
46: **end if**
47: **end for**

to select actions, while with a certain probability, actions are randomly chosen. Lines 20-32 illustrate the interaction process with the environment, explaining how actions affect states. Line 22 calculates the reward r using the new state and the global coverage variable. Lines 25-32 specify the conditions for ending the current iteration and the conditions that need to be recorded. Lines 33-44 adopt the core part of the D3QN algorithm, which integrates the optimizations of Double DQN [11] and Dueling DQN [12].

D. Double-threaded Pipeline

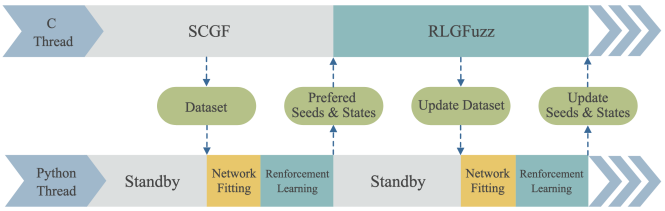


Fig. 4. Double-threaded Reinforcement Learning Guided Fuzzing Pipeline.

The deployment of our algorithm entailed the establishment of a double-threaded asynchronous pipeline, as depicted in Fig. 4. Given the significant impact of execution efficiency on the effectiveness of fuzzing [3], we ensure its operational efficiency by making the fuzzing run continuously within a dedicated C thread. The python thread are utilized to provide optional seed files and states for testing, replacing the heuristic functions in determining the priority of seed files and test states in the original SCGF [2], [3].

In the C thread, SCGF performs for 2 hours to generate the dataset. The python thread starts after the dataset reaches a certain size to learn the mapping of seed files and statemaps to bitmap, and then preferred seed files and states through reinforcement learning, meanwhile, the C thread still performs fuzzing according to the SCGF during this period until the output of the python program is finished. The C thread's fuzz program adds the preferred seed files and states to the queue every half hour and continues the fuzz test. The python thread also adds the new seed files and statemaps to the dataset every half hour, relearns the mapping and performs the preferred seed files and states. It is worth to notice that we do not update the dataset and train every time we get a interest seed, the time cost of training is substantial, we just use the model for inference. Our strategy is to update the dataset every half an hour, guaranteeing a trade-off between execution efficiency and virtual environment mapping accuracy.

IV. Experiment

A. Research Questions

The utility of our reinforcement learning-guided fuzzing pipeline is validated by answering four research questions:

- **RQ.1 Mapping Fitting.** Does the neural network genuinely learn the mapping between seed files, states, and path coverage?

- **RQ.2 Path Coverage.** In comparison to the baseline, how much more path coverage does RLGFuzz achieve?
- **RQ.3 Execution Speed.** In comparison to the baseline, how much does RLGFuzz improve its execution speed?
- **RQ.4 Efficiency in Triggering Crashes.** In comparison to the baseline, how much does RLGFuzz improve its efficiency in triggering crashes?

B. Benchmark and Baselines

The subject programs used in our experiment are demonstrated as follows. These subject programs originate from Profuzzbench [21] and are utilized extensively in the field of IoT communication. Moreover, they serve as the most illustrative representation of the numerous implementations of their protocols.

- Real-Time Streaming Protocol (RTSP) [22] is a network protocol for real-time streaming media delivery that allows clients to connect to a server to request the playback or recording of video and audio streams. Live555 is a streaming media open-source library that provides implementations of a variety of streaming media protocols including RTSP.
- Domain Name System (DNS) [23] is a distributed naming system used to map domain names to IP addresses, enabling the addressing and accessing of resources on the Internet by resolving domain names to their corresponding IP addresses. DNSmasq is a lightweight DNS forwarder and DHCP server software that also provides caching of DNS queries and DHCP server functions.
- Session Initiation Protocol (SIP) [24] is a communication protocol for initiating and managing interactive sessions involving multimedia elements such as voice, video, and messaging over IP networks. Kamailio is an open-source SIP server platform designed for scalability and flexibility in handling real-time communication services.
- Datagram Transport Layer Security (DTLS1.2) [25] is a cryptographic protocol that provides secure communication over unreliable datagram protocols such as UDP. TinyDTLS is a lightweight implementation of DTLS designed for resource-constrained devices, enabling secure communication in IoT and embedded systems [26].

As baseline tools, AFLNet [2], AFLNwe [8] and SMGFuzz [7] are selected. AFLNet [2] is one of the most popular, open source, mutation-based state-guided protocol fuzzers. AFLNwe [8] is a network-based version of AFL [27] that uses TCP/IP sockets instead of files as inputs. SMGFuzz [7] is a reverse state selection method for Stateful Cover-Guided Fuzzing Based on StateMap. The method prioritizes the coverage information of fuzzing seeds and uses Statemap to optimize the representation of the protocol state machine. We calculate the average of coverage and unique crashes achieved across 4 repetitions of 24 hours.

C. Datasets

The datasets employed in RQ2-4 were generated online for each experiment, and the evaluation metrics were computed and recorded in real time. The dataset employed for the evaluation of RQ1 comprises seeds, `statemaps`, and `bitmaps` obtained from all repeated experiments. These were prepared offline and divided into training and test sets in a 7:3 ratio.

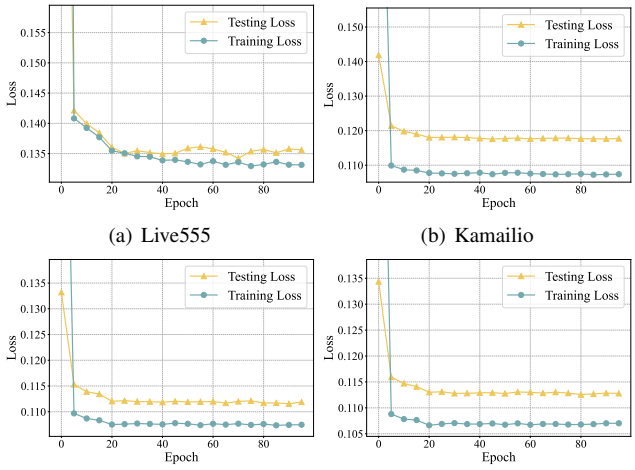
D. Experimental Infrastructure

All experiments were conducted on a server equipped with 56 Intel(R) Xeon(R) E5-2690 v4 CPU cores at 2.60GHz, and 8 Nvidia RTXForce 3090 graphics cards. Its operating system is Linux Ubuntu 22.04.2 LTS.

V. Evaluation

RQ.1 Mapping Fitting

The first aspect to verify is the accuracy of mapping learning of seed files and states to path coverage by neural network, which is crucial for ensuring the reliability of virtual environments in reinforcement learning.



(a) Live555 (b) Kamailio

(c) DNSmasq (d) Tinydtls

Fig. 5. Average Loss Curves on Training and Testing Datasets.

We calculate the average loss on both sets across multiple repeated experiments for different SUTs. As illustrated in Fig. 5, the training loss initially exceeded or was comparable to the testing loss. After approximately 20 training epochs, convergence was essentially achieved, with the final training set loss lower than the testing loss and no occurrence of overfitting.

TABLE I
AVERAGE OF VARIOUS CLASSIFICATION EVALUATION METRICS.

Subject	Accuracy(%)		F1-score(%)		AP(%)	
	Train	Test	Train	Test	Train	Test
Live555	95.93	95.92	57.21	53.52	66.92	62.43
Kamailio	95.81	95.74	57.80	56.19	68.09	56.92
DNSmasq	95.80	95.76	58.06	55.31	72.43	62.81
Tinydtls	95.84	95.78	56.68	52.63	68.17	54.33

As previously stated in Section III-B, the research problem can be considered both a prediction and a classification

TABLE II
AVERAGE OF PATH COVERED BY RLGFUZZ AND THE BASELINES IN 4 REPETITIONS OF 24 HOURS.

Subject	RLGFuzz	Comparison with AFLNet			Comparison with SMGFuzz			Comparison with AFLNwe		
		AFLNet	Improve	$\hat{A}_{12}$	SMGFuzz	Improve	$\hat{A}_{12}$	AFLNwe	Improve	$\hat{A}_{12}$
Live555	2942.75	2507.00	17.38%	1.00	2668.00	10.30%	1.00	2713.00	8.47%	0.92
Kamailio	3513.50	3188.25	10.20%	0.83	3367.00	4.35%	0.75	1746.50	101.17%	1.00
DNSmasq	936.00	892.00	4.93%	1.00	915.50	2.24%	0.75	431.00	117.17%	1.00
Tinydtls	404.00	260.50	55.09%	1.00	357.25	13.09%	0.70	238.00	69.75%	1.00
AVG.	1949.06	1711.94	13.85%	-	1826.94	6.68%	-	1174.38	65.97%	-

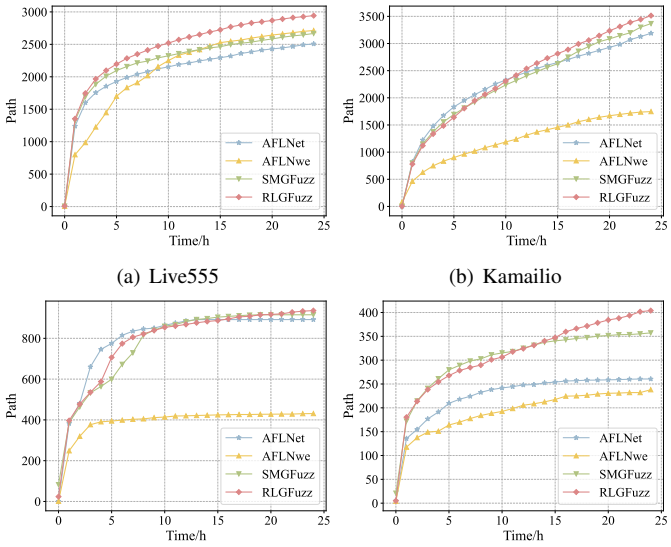
problem. The efficacy of deep neural network learning can be assessed in terms of the metrics employed to evaluate the classification problem. Accuracy, F1-score and Average Precision(AP) are selected as evaluation metrics.

Table I shows the average values of various classification evaluation metrics on the training and test sets in multiple repetition experiments for different SUTs. Accuracy represents the overall effect of classification, and our neural network achieves high scores on all four SUTs. The number of label "0" in `bitmaps` is significantly greater than the number of label "1", resulting in an imbalance in the categories of our dataset. Consequently, the F1-score and AP are inevitably low. Nevertheless, the scores of our neural network are deemed an acceptable result [28], [29]. Fig. 5 and Table I indicate that the neural network has indeed learned the mapping of seed files and states to path coverage, ensuring that the interactive environment used for reinforcement learning closely resembles the real fuzzing environment.

RQ.2 Path Coverage

The measurement of path coverage between code basic blocks is the most popular metric for evaluating the effectiveness of fuzzing. Under the same execution time, a higher coverage achieved by a fuzzer indicates its greater efficiency in discovering potential bugs and vulnerabilities in the subjects. Table II and Fig. 6 present the average coverage obtained from four repetitions of 24-hour fuzzing for each of the four SUTs.

The experimental figures indicate that for all SUTs, our RLGFuzz covers more paths than all baseline methods. The average path coverage of RLGFuzz is improved by 13.85% over AFLNet, 6.68% over SMGFuzz, and 65.97% over AFLNwe. In the case of the specific SUT, RLGFuzz demonstrated superior performance to the other three methods when tested on Live555. Upon testing on Kamailio, DNSmasq, and Tinydtls, RLGFuzz initially exhibited a decline in performance relative to AFLNet or SMGFuzz. Nevertheless, RLGFuzz completed the overtake after 10, 21, and 14 hours, respectively, and ultimately yielded higher results than the other three methods. This is the result of the offset function designed to calculate the rewards in the reinforcement learning algorithm. When the fuzzing test is in the middle and late stages, the state selection of RLGFuzz is still instructive and flexible, which is not the case with AFLNet, which allocates the energy to states by a fixed formula, and SMGFuzz, which selects states by polling.



(a) Live555 (b) Kamailio

(c) DNSmasq (d) Tinydtls

Fig. 6. Average of Path Covered by RLGFuzz and the Baselines.

We also performed a non-parametric Mann-Whitney U-test [30] with a significance level of 0.05. The $\hat{A}_{12}$ is given by the following equation.

$$\hat{A}_{12} = \frac{R_1}{mn} - \frac{m+1}{2n} \tag{15}$$

where R_1 represents the sum of the rank of the first set of data in ascending order in all the data, and m, n represents the number of the first and second sets of data, respectively. Compared to AFLNet and AFLNwe, the boost from our RLGFuzz is very robust, and compared to SMGFuzz, the $\hat{A}_{12}$ of RLGFuzz is also above the Vargha-Delaney effect size 0.70 [5], [31], which can be recognized that RLGFuzz is superior to SMGFuzz.

RQ.3 Execution Speed

Our original intention of designing RLGFuzz is to reduce the waste of time and computational resources caused by overly verbose variant message sequences, and we also need to verify that the design of the asynchronous double-threaded pipeline does not degrade the efficiency of the fuzzing test, so the execution speed is a necessary evaluation metric.

The number of messages executed by SUTs in each fuzzing test was divided by 86400 seconds to calculate the average

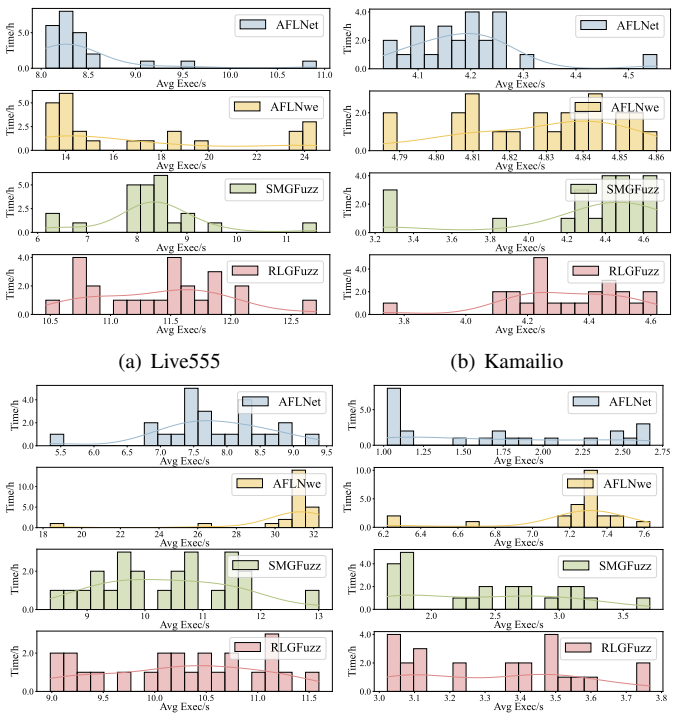
TABLE III
AVERAGE OF CRASHES TRIGGERED BY RLGFUZZ AND THE BASELINES IN 4 REPETITIONS OF 24 HOURS.

Subject	RLGFuzz	Comparison with AFLNet			Comparison with SMGFuzz			Comparison with AFLNwe		
		AFLNet	Improve	$\hat{A}_{12}$	SMGFuzz	Improve	$\hat{A}_{12}$	AFLNwe	Improve	$\hat{A}_{12}$
Live555	109.25	167.00	-34.58%	0.00	94.50	15.61%	0.75	161.75	-32.46%	0.06
Kamailio	6.00	0.00	/	1.00	1.00	500.00%	1.00	0.00	/	1.00
DNSmasq	123.50	44.50	177.53%	1.00	131.50	-6.08%	0.06	75.00	64.67%	1.00
Tinydtls	86.00	78.00	10.26%	0.75	66.00	30.30%	1.00	71.00	21.13%	0.88
Total	324.75	289.50	12.18%	/	293.00	10.84%	/	307.75	5.52%	/

TABLE IV
AVERAGE EXECUTION SPEED OF RLGFUZZ AND THE BASELINES IN 4 REPETITIONS OF 24 HOURS.

Subject	RLGFuzz	AFLNet	SMGFuzz	AFLNwe
Live555	11.4	8.5(35.0%)	8.3(37.1%)	16.9(-32.4%)
Kamailio	4.3	4.2(3.1%)	4.3(0.7%)	4.8(-10.6%)
DNSmasq	3.3	1.7(94.7%)	2.4(37.6%)	7.2(-53.8%)
Tinydtls	10.4	7.8(33.1%)	10.3(1.3%)	30.6(-66.0%)
AVG.	7.4	5.5(32.9%)	6.3(16.5%)	14.9(-50.5%)

execution speed for each experiment. Additionally, the number of messages executed by SUTs in each hour was divided by 3600 seconds to calculate the average execution speed in each hour. Table IV and the statistical histograms shown in Fig. 7 were obtained.



(a) Live555 (b) Kamailio

(c) DNSmasq (d) Tinydtls

Fig. 7. Statistical Histogram of the Average Execution Speed of RLGFuzz and the Baselines. The horizontal axis represents the average execution speed, and the vertical axis represents the number of hours we counted executed in each speed interval.

Since AFLNwe is a stateless fuzzer that does not need to guide the next selection by reconstructing the IPSM, its execution speed is quite fast and much larger than the other three. Among AFLNet, SMGFuzz and RLGFuzz, which are also SCGF, the average execution speed of RLGFuzz is 32.9% higher than that of AFLNet 16.5% higher than that of SMGFuzz.

RQ.4 Efficiency in Triggering Crashes

The amount of crashes triggered can also be used as a practical indicator for evaluating the efficiency of fuzzers. We enumerated the average number of unique crashes triggered in four repetitions of 24-hour experiments and obtained the data shown in Table III.

Overall, our RLGFuzz triggered more crashes compared to the baselines, with an increase of 12.18% over AFLNet, 10.84% over SMGFuzz, and 5.52% over AFLNwe.For specific fuzzers and protocol-implemented servers, AFLNwe triggered the most crashes in Live555, SMGFuzz triggered the most crashes in DNSmasq, and RLGFuzz triggered the most crashes in Tinydtls. All four fuzzers triggered few or no crashes in Kamailio, which may be caused by the large size of code in Kamailio, up to 939k lines [5], [24], and the low actual percentage of path coverage and code base block coverage achieved by the fuzzers.

VI. CONCLUSION

This article identifies one of the reasons for the decreased efficiency of existing protocol fuzzers after a certain duration of testing, which is the lack of flexibility and adaptability in seed and state selection. To address this issue, we propose RLGFuzz, which utilizes neural networks to learn the underlying mapping of seeds and state to path coverage, and employs it in the environment for reinforcement learning to preferentially select seeds and states for the fuzzer. Thus implementing an asynchronous double-thread reinforcement learning-guided fuzzing pipeline. Experimental results demonstrate significant improvements in code coverage and crash-triggering efficiency with RLGFuzz. Compared to AFLNet, AFLNwe, and SMGFuzz, RLGFuzz exhibits a 6.68% enhancement in path coverage and a 5.52% enhancement in crash-triggering efficiency, outperforming all three. RLGFuzz also demonstrates commendable execution speed, achieving a 16.5% improvement compared to AFLNet and SMGFuzz, both of which are also SCGF.

REFERENCES

[1] Z. Chi, Y. Li, Y. Yao, and T. Zhu, "Pmc: Parallel multi-protocol communication to heterogeneous iot radios within a single wifi channel," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pp. 1–10, 2017.

[2] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 460–465, 2020.

[3] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 720–730, 2022.

[4] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "Ganfuzz: A gan-based industrial network protocol fuzzing framework," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pp. 138–145, 2018.

[5] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[6] L. Huang, P. Zhao, H. Chen, and L. Ma, "Large language models based fuzzing techniques: A survey," *arXiv preprint arXiv:2402.00350*, 2024.

[7] L. Yu, S. Yanlong, and Z. Ying, "Stateful protocol fuzzing with statemap-based reverse state selection," *arXiv preprint arXiv:2408.06844*, 2024.

[8] V.-T. Pham, "Github - thuanpv/aflnwe," 2020. [Online]. Available: https://github.com/thuanpv/aflnwe.

[9] J. Clifton and E. Laber, "Q-learning: Theory and applications," *Annual Review of Statistics and Its Application*, vol. 7, no. 1, pp. 279–301, 2020.

[10] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[11] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[12] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*, pp. 1995–2003, PMLR, 2016.

[13] S. E. Li, *Reinforcement learning for sequential decision and optimal control*. Springer, 2023.

[14] Z. Zhang, B. Cui, and C. Chen, "Reinforcement learning-based fuzzing technology," in *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)*, pp. 244–253, Springer, 2021.

[15] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in neural information processing systems*, vol. 12, 1999.

[16] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing," 2021.

[17] L. O. Chua, *CNN: A paradigm for complexity*, vol. 31. World Scientific, 1998.

[18] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[19] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[20] A. Apicella, F. Donnarumma, F. Isgrò, and R. Prevete, "A survey on modern trainable activation functions," *Neural Networks*, vol. 138, pp. 14–32, 2021.

[21] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, pp. 662–665, 2021.

[22] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol (rtsp)," tech. rep., 1998.

[23] S. Powers, "The open-source classroom: Dnsmasq, the pint-sized super dæmon!," *Linux Journal*, vol. 2014, no. 245, p. 8, 2014.

[24] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "Sip: session initiation protocol," tech. rep., 2002.

[25] A. Haroon, S. Akram, M. A. Shah, and A. Wahid, "E-lithe: A lightweight secure dtls for iot," in *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)*, pp. 1–5, IEEE, 2017.

[26] A. Fragkiadakis, "Dtls connection identifiers for secure session resumption in constrained iot devices," in *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pp. 427–432, IEEE, 2021.

[27] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting american fuzzy lop: a fuzzbench evaluation," *ACM transactions on software engineering and methodology*, vol. 32, no. 2, pp. 1–26, 2023.

[28] L. Wang, M. Han, X. Li, N. Zhang, and H. Cheng, "Review of classification methods on unbalanced data sets," *Ieee Access*, vol. 9, pp. 64606–64628, 2021.

[29] A. Kumar, S. Goel, N. Sinha, and A. Bhardwaj, "A review on unbalanced data classification," in *Proceedings of International Joint Conference on Advances in Computational Intelligence: IJCACI 2021*, pp. 197–208, Springer, 2022.

[30] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[31] G. Neumann, M. Harman, and S. Poulding, "Transformed vargha-delaney effect size," in *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings 7*, pp. 318–324, Springer, 2015.