Received 9 June 2024, accepted 24 June 2024, date of publication 2 July 2024, date of current version 20 September 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3421989

RESEARCH ARTICLE

A Coverage-Guided Fuzzing Method for Automatic Software Vulnerability Detection Using Reinforcement Learning-Enabled Multi-Level Input Mutation

VAN-HAU PHAM, DO THI THU HIEN, NGUYEN PHUC CHUONG, PHAM THANH THAI, AND PHAN THE DUY
Information Security Laboratory, University of Information Technology, Ho Chi Minh City 700000, Vietnam
Vietnam National University Ho Chi Minh City, Ho Chi Minh City 700000, Vietnam

Corresponding author: Van-Hau Pham (haupv@uit.edu.vn)

This work was supported by the VNUHCM-University of Information Technology's Scientific Research Support Fund.

ABSTRACT Fuzzing is a popular and effective software testing technique that automatically generates or modifies inputs to test the stability and vulnerabilities of a software system, which has been widely applied and improved by security researchers and experts. The goal of fuzzing is to uncover potential weaknesses in software by providing unexpected and invalid inputs to the target program to monitor its behavior and identify errors or unintended outcomes. Recently, researchers have also integrated promising machine learning algorithms, such as reinforcement learning, to enhance the fuzzing process. Reinforcement learning (RL) has been proven to be able to improve the effectiveness of fuzzing by selecting and prioritizing transformation actions with higher coverage, which reduces the required effort to uncover vulnerabilities. However, RL-based fuzzing models also encounter certain limitations, including an imbalance between exploitation and exploration. In this study, we propose a coverage-guided RL-based fuzzing model that enhances grey-box fuzzing, in which we leverage deep Q-learning to predict and select input variations to maximize code coverage and use code coverage as a reward signal. This model is complemented by simple input selection and scheduling algorithms that promote a more balanced approach to exploiting and exploring software. Furthermore, we introduce a multi-level input mutation model combined with RL to create a sequence of actions for comprehensive input variation. The proposed model is compared to other fuzzing tools in testing various real-world programs, where the results indicate a notable enhancement in terms of code coverage, discovered paths, and execution speed of our solution.

INDEX TERMS Reinforcement learning, fuzzing, vulnerability detection, coverage fuzzing.

I. INTRODUCTION

Recently, software testing and security enhancement have become increasingly important and attract significant research and development attention. Among the techniques for detecting software vulnerabilities, fuzzing is one of the most popular, convenient, and effective methods [1], [2]. The goal of fuzzing is to identify potential flaws in software by providing unexpected and invalid inputs to the target program, thereby monitoring behavior and identifying errors, failures, or undesired outcomes [3], [4]. Fuzzing techniques can be broadly categorized into three main types: black-box fuzzing, white-box fuzzing, and grey-box fuzzing. Black-box fuzzing, also known as conventional fuzzing, operates

The associate editor coordinating the review of this manuscript and approving it for publication was Jiachen Yang.

without any knowledge of the internals of the program and randomly generates inputs to uncover basic flaws. In contrast, white-box fuzzing requires a comprehensive understanding of the program's source code, employing static and dynamic analysis to methodically explore and test the software's execution paths. Grey-box fuzzing strikes a balance between the two, leveraging limited knowledge of the application's internals along with feedback from testing to intelligently refine the fuzzing process [5]. Each approach varies in complexity, resource requirements, and efficiency, with black-box fuzzing being the simplest and quickest, white-box offering the most thorough examination at the cost of greater complexity, and grey-box providing an effective middle ground that combines ease of implementation with the potential for deep vulnerability detection.

More specifically, grey-box fuzzing, particularly when augmented with code coverage metrics, represents a prominent methodology within the fuzzing landscape [6]. This approach necessitates access to the executable files of the target application, which are then subjected to fuzzing procedures within a specialized environment. Its widespread adoption can be attributed to its applicability to closed-source software, eliminating the need for source code access. By harnessing insights from code coverage data garnered during the execution of the program, grey-box fuzzing enhances its efficacy [7]. Fuzzing frameworks employing this technique initiate the execution of the program using various test inputs while deploying sophisticated mechanisms to capture detailed code coverage information. This crucial data serves as the basis for evaluating and refining test input criteria, with the overarching goal of augmenting the fuzzing process's effectiveness. Through this iterative optimization based on dynamic execution feedback, grey-box fuzzing with code coverage insight emerges as a powerful tool in identifying software vulnerabilities [8].

The fusion of machine learning (ML) with fuzzing methodologies signifies a groundbreaking advancement in the detection of software vulnerabilities, offering a sophisticated, intelligent approach to security assessments. This innovative integration empowers fuzzing tools with the ability to learn from previous iterations, refining and targeting their search for flaws more effectively. Machine learning algorithms analyze patterns from past fuzzing activities to enhance the generation of test inputs, focusing on areas more likely to reveal critical vulnerabilities. This not only increases the efficiency of the fuzzing process by prioritizing high-risk code paths but also enables the adaptation of fuzzing strategies in real-time, optimizing the exploration of complex software environments. The result is a more nuanced, context-aware approach to vulnerability detection that significantly reduces the time and computational resources required, marking a substantial leap forward in the field of software security.

Reinforcement Learning (RL), a subset of ML characterized by an agent learning to make decisions through trial and error to achieve a specific goal, has emerged as a potent tool for enhancing fuzzing techniques in the generation of new test cases. By applying RL principles, fuzzing frameworks can dynamically adjust their strategies based on the outcomes of previously executed test cases, effectively learning which types of inputs are more likely to induce anomalies or reveal vulnerabilities in the software under test. This integration leverages the outcomes of previously executed test cases to inform the creation of new inputs, focusing on those more likely to uncover vulnerabilities by navigating unexplored or less-tested paths in the application's codebase. The application of RL not only enhances the efficiency of the fuzzing process by learning from past interactions but also expertly balances the concepts of exploitation and exploration—key elements in effective fuzzing. Exploitation delves deep into the program to reach critical code segments, while exploration aims for broad branch coverage to ensure no potential fault is overlooked. Achieving a harmonious balance between these two approaches is essential for optimizing the fuzzing model's effectiveness, thereby leading to quicker and more comprehensive identification of software vulnerabilities [9]. This evolution in fuzzing marks a significant milestone in automated software testing, offering a smarter, more adaptive, and outcome-focused method for detecting and addressing security flaws.

Current approaches like [7] and [10] merging RL with fuzzing are intensely focused on developing the algorithms, states, rewards, and parameters specific to RL while overlooking critical factors like the balance between exploitation and exploration. This emphasis leads to fuzzing models based on RL that concentrate excessively on a single code branch, thereby missing potential errors in other branches. These models often give priority to selecting mutation actions without incorporating effective mechanisms for the selection and scheduling of inputs, setting them apart from conventional fuzzing tools. Moreover, there's a significant gap in research offering a comparative analysis of RL models in fuzzing against the efficiency, strengths, and weaknesses of modern fuzzing tools.

To a certain point, the fuzzing techniques can be classified into exploitation and exploration techniques [11]. Therein, exploitation and exploration stand as pivotal concepts in the fuzzing process, with exploitation denoting the capacity of test cases to penetrate deeply and access code segments buried within the program [12], [13]. Conversely, exploration pertains to test cases achieving extensive branch coverage. A fuzzing model with an excessive focus on exploration may not generate test cases that effectively pinpoint faulty code segments within the program. Similarly, a model overly concentrated on exploitation might only aim to reach the deepest branch within a program, potentially overlooking faults in other branches. Therefore, finding an optimal balance between exploitation and exploration is essential for the development of an efficient fuzzing model. This balance ensures that the model can effectively uncover faults across different branches of the code, maximizing the potential for identifying vulnerabilities within the software [7], [9], [11].

Hence, in our research, we introduce an innovative fuzzing model guided by coverage metrics, leveraging RL (RL) to enhance input selection and scheduling. This approach aims to meticulously balance the exploration of new paths and the exploitation of deeper, potentially vulnerable segments within a program. Furthermore, our model incorporates a novel multi-level input mutation mechanism, designed to synergize with RL. This mechanism facilitates the generation of mutations via a series of deliberate actions, enabling a more granular and targeted exploration of the software's attack surface. To validate the effectiveness and efficiency of our proposed model, we undertake a comprehensive comparative analysis, juxtaposing it with two notable fuzzing tools: rlfuzz, which also employs RL strategies, and AFLplusplus, a contemporary and widely used fuzzer. Through this comparison, we aim to underscore the advantages and potential of our RL-based fuzzing approach in enhancing the identification and mitigation of software vulnerabilities, making a compelling case for its adoption in the field of cybersecurity.

Our contributions are summarized as follows:

- Propose CTFuzz, a coverage-guided fuzzing model that utilizes reinforcement learning (RL) to optimize input mutation for enhanced code coverage.
- Integrate an RL model with an input selection and scheduling algorithm for ranking the inputs in a queue based on their current coverage, then prioritizing inputs with higher coverage. Besides, it allocates the number of trials for each input to ensure a balanced distribution.
- Design and implement the multi-level input mutation that can enhance the capabilities of the proposed model by enabling the generation of complex mutations using a sequence of actions.
- Finally, we provide a perspective on how well our model performs compared to other fuzzing tools, with a particular focus on state-of-the-art fuzzing tools like AFLplusplus and rlfuzz.

The remaining sections of this article are constructed as follows. **Section II** introduced related works in fuzzing using machine learning. Next, the proposed framework and methodology are discussed in **Section III**. **Section IV** describes the experimental settings and evaluation result of CTFuzz compared with another RL-based fuzzing tool - rlfuzz and a modern fuzzing tool - AFLplusplus. **Section V** discussion about the experiments. Following the discussion, future developments are outlined in **Section VI**. Finally, we conclude the paper in **Section VII**.

II. RELATED WORK

A. FUZZING

With continuous development and research, modern fuzzing techniques can be categorized into three main types: black-box fuzzing, white-box fuzzing, and grey-box fuzzing. Among these three types, grey-box fuzzing is the most common and flexible. Therefore, our team decided to focus on investigating grey-box fuzzing for our model. The coverage-guided fuzzing process of an application consists of several fundamental steps, as illustrated in **Fig. 1**. The functions of each step are outlined below:

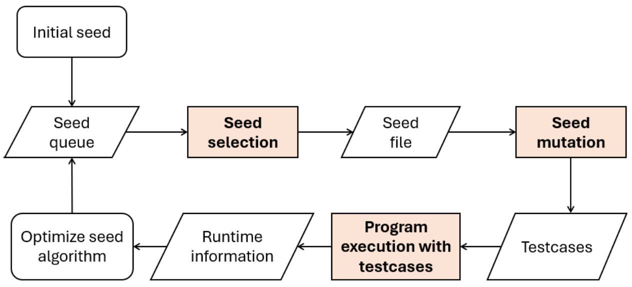


FIGURE 1. The coverage-guided fuzzing process.

1) First, we need to provide an input value (seed), usually a valid input of the program, which the fuzzing tool places into a queue of input values.
2) From this queue, the fuzzing tool selects an input value for mutation and each fuzzing tool employs different algorithms for this selection. The effectiveness of these seed selection algorithms significantly influences the fuzzing process's efficiency.
3) Next, using the chosen input value, the fuzzing tool generates various test cases by applying specific actions to the input (e.g., flipping bits, adding bits, shifting bits, etc.). The choice of actions and the effort devoted to this process also impact the fuzzing process's performance.
4) Subsequently, the fuzzing tools test the generated inputs with the target program, collecting runtime information (program crashes, execution time, code coverage, etc.).
5) Using the collected information, the fuzzing tool decides whether to continue, remove, replace, or adjust the inputs in the input queue. If a crash occurs, the fuzzing tool saves the test case and the crash information for further analysis by the fuzzing operator. The process then returns to step 2 and the cycle continues.

For fuzzing techniques based on the current code coverage, the main challenges that researchers are focusing on improving are as follows:

1) How to increase code coverage?
2) How to generate better test inputs?
3) How to perform input mutation more effectively, reducing ineffective mutations?
4) How to overcome program structure checks?
5) How to reach more branches in the program?
6) How to reduce false positives for detected vulnerabilities?

For each question, various ways and alternatives have been researched and proposed. However, we can divide the improvement approaches into the following steps:

- **Generating test cases**: In this step, researchers often apply static and dynamic analysis techniques to extract information from the executable file and create an input that can achieve deeper program exploration. This is different from using a random input or a pre-defined input pattern. In cases where the input involves special data structures (file formats, etc.), some studies employ artificial intelligence to learn the data structure and generate test cases that can bypass initial structural checks of the program.
- **Input selection and mutation**: When multiple good test cases are evaluated and saved for subsequent steps, the selection of the next input for transformation, the choice of transformation operations, and the amount of effort invested in testing that input significantly influence the model's performance. Researchers focus on minimizing redundant transformation efforts, quickly selecting accurate transformations to increase code coverage, thereby enhancing the fuzzing process's efficiency.
- **Post-fuzzing analysis**: In certain programs, the number of false positives found during fuzzing can be high, or duplicate vulnerabilities might be discovered. This can be cumbersome for manual analysts. Therefore, researchers also address this by implementing algorithms to detect duplicates, evaluating the return of detected vulnerabilities, or employing machine learning algorithms to score the exploitability of vulnerabilities.

B. TRADITIONAL IMPROVEMENT TECHNIQUES IN FUZZING

These are traditional improvement methods for fuzzing, each with its advantages and drawbacks. Ji et al. introduced AFLPro in [14], enhancing input selection and scheduling by combining static analysis with a basic block synthesis model. The goal is to prioritize inputs that reach code segments less explored previously, aiming to increase the coverage of deeper program areas. Yue et al. proposed EcoFuzz [9], modeling input scheduling as a Multi-Armed Bandit problem and presenting a variant of the Adversarial Multi-Armed Bandit model to improve it. The common idea of both techniques is to enhance input selection and scheduling, prioritizing inputs predicted to have a higher likelihood of containing errors and leading to better code coverage. However, the issue with using static analysis techniques always lies in the lack of complete runtime data and often produces low-accurate or false-positive results. Additionally, it might not work effectively on applications utilizing code obfuscation or packing mechanisms. Chen et al. created Matryoshka [15], using taint analysis to solve conditional statements and penetrate deeper into the program. However, this technique demands significant resources and slows down the fuzzing process, while taint analysis also faces challenges with under-tainting and over-tainting. Zhang et al. proposed a lightweight and convenient mechanism to surpass input checks by combining static analysis with mutating key bytes in InsFuzz [16]. They identify bytes influencing conditional statement results and then mutate them. However, since it also employs static analysis, it is still subject to limitations like the techniques mentioned above and requires modifying the executable files, causing instability in applications with integrity checks.

C. THE COMBINATION OF ARTIFICIAL INTELLIGENCE INTO FUZZING

Recently, with the explosive development of artificial intelligence, machine learning techniques have also been applied by researchers to enhance the fuzzing process. Surveys [12], [17] indicate that the application of machine learning techniques to fuzzing is diverse and creative, yielding promising results. The steps typically addressed by artificial intelligence include (1) input selection, (2) input scheduling, (3) input generation, (4) mutation action selection, etc. RapidFuzz [18] and CGFuzzer [19] employ Generative Adversarial Networks (GANs) to learn the structure of complex inputs, aiming to generate higher similarity patterns for fuzzing protocols or specific file formats. This approach helps save time by avoiding mutating invalid samples and increasing the likelihood of passing structure checks. NeuFuzz by Wang et al. [20] models the bug-finding process akin to natural language processing, utilizing deep-learning Long Short-Term Memory (LSTM) to learn the structure of error-containing paths, predicting which paths are more likely to have errors and prioritizing them for input scheduling.

The RL approaches have also been applied to fuzzing for the first time in 2018 by Böttinger et al. [21]. They transformed the fuzzing problem into an RL problem, where the selection of the next mutation action is analogous to choosing the next move in a chess game. Although an optimal strategy may exist, searching for optimal actions is performed using the deep Q-learning algorithm. However, their proposed model was specifically designed for PDF files, lacking objective results when compared to modern fuzzing tools and not addressing the balance between exploitation and exploration. Kuznetsov et al. also employed deep Q-learning to select mutation actions for application testing [22]. They demonstrated that combining RL can reduce the time needed to create expected test cases by up to 30%, yet their evaluation method does not suit real-world applications. Reddy et al. improved mutation action selection using the Monte Carlo Control algorithm, creating more valid samples for applications with complex input structures [23]. The results enhanced the rate of passing structure checks for samples. However, their model skewed towards exploitation rather than exploration, focusing on generating diverse inputs with similar features instead of exploring new behaviors. Liu et al. introduced Reinforcement Compiler Fuzzing [24], also utilizing deep Q-learning for mutation action selection at the compilation level. Nevertheless, their implementation requires source code to work effectively. Drozd and Wagner combined Deep Double Q-learning to select mutation actions and accelerate libFuzzer [25]. However, they acknowledged that it is not sufficient and that further enhancements are needed in terms of input selection and filtering.

Zhang et al. proposed rlfuzz [10], a method to balance exploitation and exploration in a deep-q-learning fuzzing model by randomly selecting trial inputs for subsequent transformations when the model does not experience an increase in code coverage. However, this selection method is not yet optimal, as inputs with low code coverage in the queue still have an equal chance of being transformed as those with higher potential. In a different approach, Wang et al. [7] utilized RL for input scheduling rather than selecting mutation actions like other studies. They proposed a multi-level code coverage model to enhance fuzzing detail and introduced a scheduling mechanism to support this multi-level code coverage model using RL. The results showed a balance between exploitation and exploration in the generated test cases, but there was no significant improvement in selecting more effective mutation actions.

Current combined RL and fuzzing solutions tend to focus heavily on designing RL algorithms, states, rewards, and parameters without considering factors like the balance between exploitation and exploration. This leads to RL fuzzing models delving deep into one code branch, missing opportunities to find vulnerabilities in other branches. Moreover, the focus often lies solely on mutation action selection, without mechanisms for effective input selection and scheduling that are crucial for real-world fuzzing tools. Furthermore, no RL fuzzing study provides a comparative perspective on performance, strengths, and weaknesses compared to modern fuzzing tools.

Recognizing the weaknesses in the exploitation-exploration imbalance of combined RL fuzzing models, this issue could be mitigated by combining effective input selection and scheduling algorithms, a topic that has been extensively studied in traditional fuzzing. Our work investigates and proposes an RL-based guided coverage-aware fuzzing model to address these weaknesses by integrating it with an effective input selection and scheduling algorithm. Additionally, we propose a multi-level input transformation algorithm that can be applied to RL-based fuzzing models, coupled with a waste reduction mechanism to improve model efficiency.

III. METHODOLOGY

A. THE ARCHITECTURE OF CTFUZZ

Inherited from the fuzzing model using RL named rlfuzz [10], our CTFuzz is designed with four main components as in **Fig. 2**. The main improvements of our CTFuzz compared to its predecessor include replacing the random selection approach with a balanced input selection and scheduling algorithm, which ensures prioritization of inputs with higher code coverage. Additionally, we have implemented a multi-level input transformation model that can be combined with RL to enhance the long-term performance of the fuzzing model, especially when multiple consecutive actions are needed to transform inputs effectively.

Each component in CTFuzz is responsible for different tasks to enhance the effectiveness of the fuzzing process.

1) **Seed selection and schedule algorithm:** From a queue of inputs, the algorithm ranks them based on the current coverage and prioritizes inputs with higher coverage, while allocating the number of trials to the model.
2) **RL model:** This model receives inputs and selects the appropriate action to mutate them, aiming to predict the best coverage improvement.
3) **Multi level input mutation:** This component receives the input from (1) and the action from (2) to perform a mutation on that input, generating a list of test cases, which are then fed into (4) to obtain results. Multi-level input mutation and early stopping mechanism are applied at this stage.
4) **Coverage observer:** takes the responsibility of executing inputted test cases on the target program to obtain results.

B. INPUT MUTATION WITH RL

When applying RL to chess, the state of the RL model can be considered as the positions of the pieces on the chessboard, the actions as selecting the next move, and the reward as the advantage gained after making the move (chess computer programs have algorithms to evaluate advantages based on the chessboard position, which we will not elaborate on here). Researchers can then apply RL techniques to train the model to find the optimal moves by playing numerous chess games and accumulating experience (rewards). One notable example is AlphaZero, developed by DeepMind, which used a combination of RL and deep learning to play chess automatically. It improved its chess abilities by playing thousands of games against itself, accumulating experience, and has become one of the strongest chess-playing tools in the world.

Similarly, we can apply a similar concept to fuzzing, where the test inputs become like the positions of chess pieces and become the input for the RL model. Selecting transformation actions will be similar to choosing the next move on the chessboard. Meanwhile, code coverage will be analogous to the advantage on the chessboard and becomes the reward for the RL model. Consequently, we can train the RL model to select the best transformation actions for each input based on the accumulated experience. The reward mechanism does not necessarily depend solely on code coverage but can incorporate other factors based on what we want to improve in the model, such as execution time, length of test cases, or their combinations.

Based on a similar concept, we model the fuzzing process as a Markov decision problem, where the number of newly

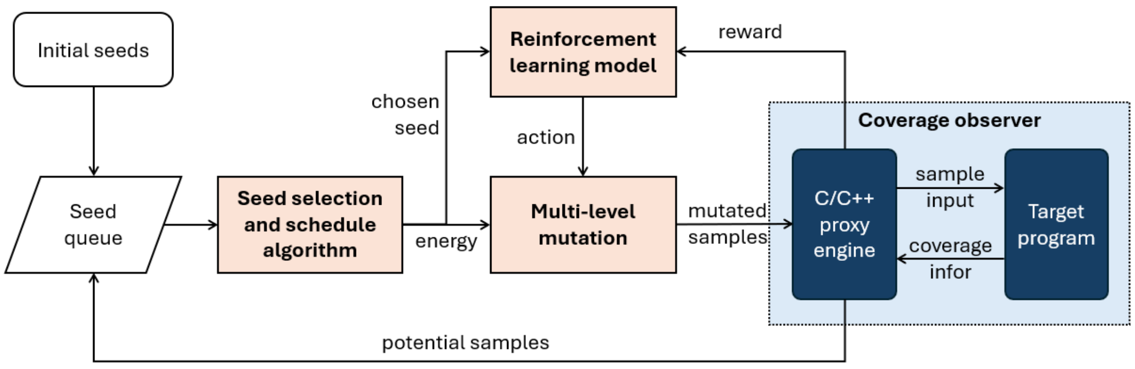
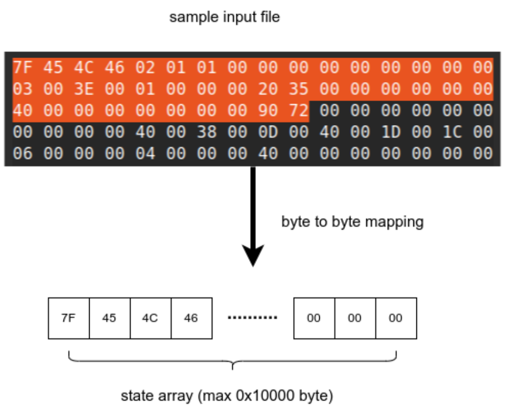


FIGURE 2. The architecture of CTFuzz.



```
7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00
03 00 3E 00 01 00 00 00 20 35 00 00 00 00 00 00
40 00 00 00 00 00 00 00 90 72 00 00 00 00 00 00
00 00 00 00 40 00 38 00 0D 00 40 00 1D 00 1C 00
06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00
```

FIGURE 3. Converting fuzzing input to state in RL model.

TABLE 1. Actions for mutation.

Action	Description
Mutate_EraseBytes	Erase byte
Mutate_InsertByte	Insert byte
Mutate_InsertRepeatedBytes	Insert a sequence of bytes
Mutate_ChangeByte	Change byte
Mutate_ChangeBit	Change a bit of a byte
Mutate_ShuffleBytes	Shuffle the order of bytes within a range
Mutate_ChangeASCIIInteger	Change integer
Mutate_ChangeBinaryInteger	Change binary integer
Mutate_CopyPart	Perform byte copy and insertion

discovered code coverage is considered as the reward, aiming for the model to seek new code portions within the program. The detailed definitions of *states*, *actions*, and *rewards* in our proposed RL model will be further elaborated in the subsequent sections.

1) STATE

For the state space, we represent all test inputs as byte arrays with a maximum length of 65,536 (0×10000) bytes, as in **Fig. 3**. Bytes in each input are converted into values in the range of [0, 255]. Such a state representation limits the maximum length of the test input to a constraint of 65,536 bytes due to the consideration that a too-large state would slow down the processing of the RL model. With the initial byte length, based on the chosen mutate action, the input values are modified accordingly by adding bytes, truncating bytes, or permuting bytes (new states). Hence, our aim is for the fuzzing model to select actions based solely on the mutated input and choose the best mutation action based on experience.

2) ACTION

In RL, actions are taken to interact with an environment and achieve specific goals. In the context of fuzzing, the actions are the mutation applied to the test inputs to feed into the target programs. Our RL model includes 9 mutate actions on the bytes of the test input, inspired by libFuzzer [26], due to its independence and ease of implementation compared to other fuzzing engines. The detailed definition of these actions is provided in **Table 1**.

3) REWARD

One of the most crucial steps in designing an RL model is devising the reward mechanism, as it significantly impacts the performance and outcomes of the RL model. Thus, the reward mechanism needs to be well-designed and aligned with the desired goals of the model. In the context of our model during fuzzing operations, our aspiration is for the model to maximize code coverage, aiming to explore code regions that have not been touched before. Consequently, *new* or *increased* code coverage becomes the criterion within our reward mechanism. Particularly, considering the target program as a graph with various code blocks as nodes and their relationships as edges, we aim to discover as many edges as possible. The number of newly explored edges plays the role of an indication for increased coverage and should result in a higher reward. Hence, given that *total_new_coverage* is indeed the number of newly discovered edges and *energy* is the number of attempts used, the *reward* in our RL model is defined as in (1).

$$reward = \frac{total\_new\_coverage}{energy} \tag{1}$$

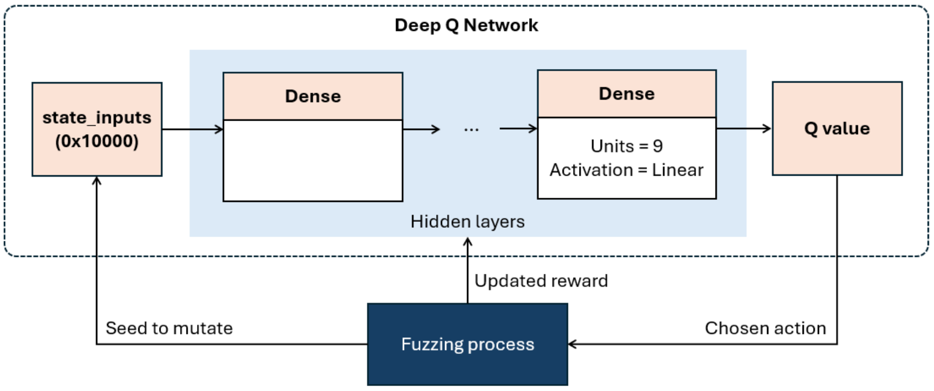


FIGURE 4. Architecture of Deep Q Learning.

This reward mechanism is designed to help optimize the discovery of new code segments for the model, as only those segments truly make an impact, but not the high code coverage on previously discovered parts. It also ensures the goal of our model is to prioritize the ability to find inputs that contribute to new code coverage, rather than achieving the highest coverage on each input. In addition, using the number of attempts in the reward is for compatibility with the input scheduling mechanism described in **Section III-C**, where the number of trials is distributed differently from action to action and the averaging approach will ensure fairness for inputs with few trials.

4) DEEP Q NETWORK MODEL

Our RL model employs Deep Q-Learning with the architecture and general operation depicted in **Fig. 4**. The initial hidden layer takes an input size of 65,536 (0×10000), which is the state size of the model. The final hidden layer has 9 units, corresponding to the number of available actions. Additionally, at the end of the network, a policy is incorporated to select an action based on the predicted scores of the 9 actions.

The chosen action from this process is then used to create a list of test cases used for fuzzing by transforming the input. The outcome of the fuzzing process with test cases created by transforming input with the given action is used as feedback to update the layers of the network. Subsequently, the next input in the queue becomes the next input state of the RL model.

C. SEED SELECTION AND SCHEDULE ALGORITHM

Utilizing a random input selection mechanism for transformations, like rlfuzz [10], is ineffective, as it can lead to an improper allocation of test cases due to an equal probability of selecting inputs with both low and high code coverage. Based on the common steps in the fuzzing process with code coverage, it can be noticeable that inputs achieving higher code coverage typically surface in later stages, after multiple transformations from ones with lower code coverage. Therefore, transforming found inputs with higher code coverage has the potential to enable the exploration of deeper and previously unexplored branches, thus increasing code coverage. Moreover, after fuzzing for a certain period, mutating inputs with lower code coverage is more likely to revisit previously discovered code regions, resulting in duplication and time wastage in the fuzzing process.

Hence, our paper introduces an input selection and scheduling algorithm with the following criteria.

- Prioritize fuzzing for inputs with higher code coverage.
- Allocate more trials to inputs with higher code coverage.
- Distribute some trials and ensure the reasonable testing frequency of inputs with lower code coverage, balancing the exploitation and exploration of the model.

To ensure that all inputs are used in the fuzzing process, we iterate through input transformations in a loop. At the beginning of each loop, inputs are organized in the queue based on their code coverage in descending order and are allocated a corresponding number of trials. Upon a loop finishing testing all inputs, only the ones that result in new code coverage are added to the queue for being used as the input for the next loop of further transformations.

For more details, each input in the ordered queue is assigned a specific number of trials according to (2), where its ranking based on code coverage does matter.

$$energy_i = \max(\frac{total\_energy * (n-i)}{SAP(n)}, min\_energy) \quad (2)$$

where:

- i: The order of the input after being rearranged according to decreasing code coverage, starting from 0.
- $energy_i$: Number of trials corresponding to the i^{th} input
- $total\_energy$: Total number of trials for one loop.
- min_energy: Minimum number of trials.
- n: Total number of available inputs in the queue.

Algorithm 1 Seed Selection and Schedule Algorithm

```
Require: Seed queue $Q$
Ensure: List of seed and mutation energy in order
1: function SEED_SELECTION_AND_SCHEDULE($Q$)
2: $Q' = Q$
3: $sort(Q')$ // Order seeds in $Q'$ by coverage
4: $n = len(Q')$
5: for $i \leftarrow 0$ to $n-1$ do
6:     $Q'[i].energy = \max(total\_energy * (n - i)/SAP(n), min\_energy)$
7: end for
8: return $Q'$
```

- $SAP(n)$: Sum of arithmetic progression with n elements $(1 \ldots n)$, which is defined as in (3).

$$SAP(n) = \sum_{i=1}^{n} i \tag{3}$$

With this formula, inputs with higher code coverage, or higher rank, will be tested on a more regular basis with more trials and prioritized for testing. Meanwhile, lower-code-coverage inputs will still be tested later with smaller numbers of trials, but not less than the minimum. Moreover, we limit the total trial count of one loop to avoid excessively long loops in which newly added inputs are not tested adequately.

In general, **Algorithm 1** provides a pseudocode representation of our seed selection and scheduling algorithm. Additionally, to complement and prevent wastage for inputs allocated a larger trial count, we integrate a waste-reduction mechanism as detailed in **Section III-D2**.

D. MULTI-LEVEL INPUT MUTATION AND EARLY STOPPING MECHANISM

Fuzzing models using RL currently lack practicality due to the absence of certain techniques or straightforward optimization algorithms. Notable examples include the *Multi-level Input Mutation* mechanism and the *Early stopping Mechanism*. These are algorithms already developed and utilized by modern fuzzing tools like AFLplusplus [27]. In the scope of our work, we have made slight simplifications and modifications to incorporate these techniques seamlessly into our fuzzing model.

1) MULTI-LEVEL INPUT MUTATION

Many modern fuzzing tools like AFLplusplus [27] have implemented the idea of multi-level mutation on input, where the fuzzing model needs to execute more than one consecutive action to discover new "noteworthy" samples. Considering this as an essential mechanism to enhance the effectiveness of the fuzzing process, we designed a simplified multi-level mutation algorithm that is compatible with the RL-based fuzzing model.

Algorithm 2 Multi-Level Input Mutation Algorithm

```
Require: Input $seed$, mutation depth $depth$
Ensure: List of test cases
1: function MULTI_LEVEL_MUTATE($seed$, $depth$)
2: $input = seed.input$
3: $energy = seed.energy$
4: for $step \leftarrow 1$ to $depth - 1$ do
5:     $action = get\_action(input)$ // Get action from RL model
6:     $input = mutate(input, action)$
7: end for
8: $action = get\_action(input)$
9: // Get action from the RL model
10: if $action\_is\_maximun\_try(seed, action, C)$ then
11:     // Random pick another valid action
12:     $action = pick\_another\_action(seed, C)$
13: end if
14: $testcase\_list = array[energy]$
15: for $i \leftarrow 1$ to $seed.energy$ do
16:     $testcase = mutate(input, action)$
17:     $testcase\_list$.append($testcase$)
18: end for
19: $update\_mutate\_count(seed, action)$
20: if $all\_mutate\_reach\_maximum\_try(seed, C)$ then
21:     $remove\_from\_queue(Q_{depth}, seed)$
22:     $add\_to\_queue(Q_{depth+1}, seed)$
23: end if
24: return $testcase\_list$
```

Algorithm 2 provides a pseudocode representation of the proposed multi-level mutation algorithm, with the key ideas as follows.

- The initial transformation depth is set to 1.
- Each input is associated with a counter table that tracks the number of times each transformation action is executed.
- Each action is **selected** no more than C times.
- If an action has been selected more than C times, a different action that has not reached its threshold is randomly chosen for transformation.
- Once all actions for an input have been selected C times, the input is removed from the queue Q_n and placed into the queue Q_{n+1}.
- If the queue Q_n is emptied, the model proceeds to a new depth of $n+1$ and switches to queue Q_{n+1}.

2) EARLY STOPPING MECHANISM

Another effective algorithm that contributes to the efficiency of the fuzzing process of other tools is the "early stopping" mechanism, or so-called early abort. Take AFLplusplus as an example, where this allows the fuzzing system to early stop testing and switch to other inputs upon numerous trials on a specific input without achieving appropriate results, ignoring the remaining number of available trials.

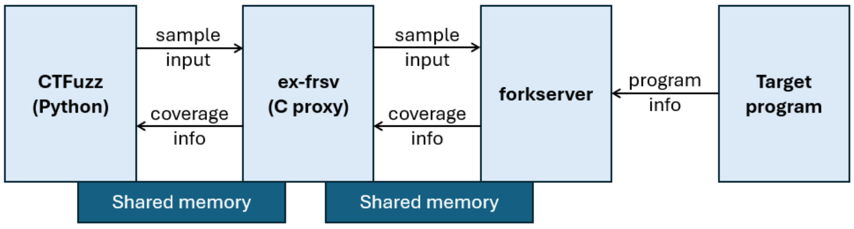


FIGURE 5. Coverage observation method.

Algorithm 3 Early Stopping Mechanism

```
1:  $Q' = SEED\_SELECTION\_AND\_SCHEDULE(Q)$
2:  for $seed$ in $Q'$ do
3:      $last\_find = 0$
4:      for $i \leftarrow 1$ to $seed.energy$ do
5:          $input = mutate(seed.input, action)$
6:          $reward = run\_target(input)$
7:          if $has\_new\_cov\_or\_new\_unique\_path(reward)$ then
8:              $last\_find = i$
9:          end if
10:         if $i - last\_find > M$ then
11:             break // go to the next seed in the queue
12:         end if
13:     end for
14: end for
```

Inheriting this idea, we simplified and implemented this mechanism for our model by introducing a constant M representing the maximum consecutive unsuccessful trials. When transforming an input fails to increase code coverage or find new paths for M times continuously, the model will move on to a different input. This approach can save a significant number of futile trial attempts.

However, it is noticeable that setting an appropriate value for M is also crucial. While too large a value leads to an ineffective waste-prevention mechanism, a value that is too small can cause the model to ignore significant opportunities for new code coverage in subsequent trials. Thus, a balanced value for M is essential, striking a trade-off for the model's performance. Moreover, the efficacy of this choice also depends on the values of the two constants $total\_energy$ and min_energy used in the trial allocation algorithm discussed in **Section III-C**. Particularly, though adjusting M based on the allocated number of trials might yield better results, in our model, we opt for simplicity by using a fixed value for M throughout the process. The mechanism can be outlined in pseudocode form, as shown in **Algorithm 3**.

E. COVERAGE OBSERVER

In coverage-guided fuzzing models, a crucial step is the extraction of code coverage information during program execution, significantly impacting the model's effectiveness. In our work, this process needs to be rapid, precise, and stable, especially when dealing with numerous test cases and independent of access to the source code of the target program.

To meet these requirements, we take the idea of the client-server model using a fork server introduced in AFLplusplus [27]. While the server is responsible for initializing the target program and employing Frida to inject code as well as recording the initial state, the client interacts with it via shared memory to transfer samples and receive code coverage information. To implement our coverage observation mechanism, we consider the CTFuzz model as an ALFplusplus client. Then, we designed an application called $ex-frsv$, which is responsible for not only initiating a *forkserver*-like server but also playing the role of a proxy, enabling the connection between client and server.

The interaction of the proposed CTFuzz and those components to obtain code coverage is depicted in **Fig. 5**. For more details, $ex-frsv$ receives test cases from the CTFuzz model and sends them to *forkserver*, which then returns the code coverage result. Some techniques such as shared memory and semaphores are also employed to facilitate communication between the model and $ex-frsv$ to enhance both speed and stability. Moreover, being a coverage-guided fuzzing tool, the coverage observer in our model returns obtained coverage via a *bitmap*, in which each bit corresponds to a basic code block and is enabled when that block is hit.

F. WORKFLOW OF CTFUZZ

The workflow of our proposed model, which involves the cooperation of the components mentioned above, is illustrated in **Fig. 6**. As a coverage-oriented grey-box fuzzing model using RL, equipped with other supporting algorithms to enhance effectiveness and performance, CTFuzz works with the main steps as follows.

1) Initially, the model pushes the initial seed to the queue.
2) The model proceeds to mutate the seed in a loop. At the start of a new iteration, all seeds in the queue are sorted in descending order of code coverage to prioritize seeds with higher code coverage.
3) Next, the scheduling algorithm is invoked to allocate the number of attempts for each seed. This is designed to distribute mutate energy reasonably, ensuring that a seed with higher potential is given more priority while

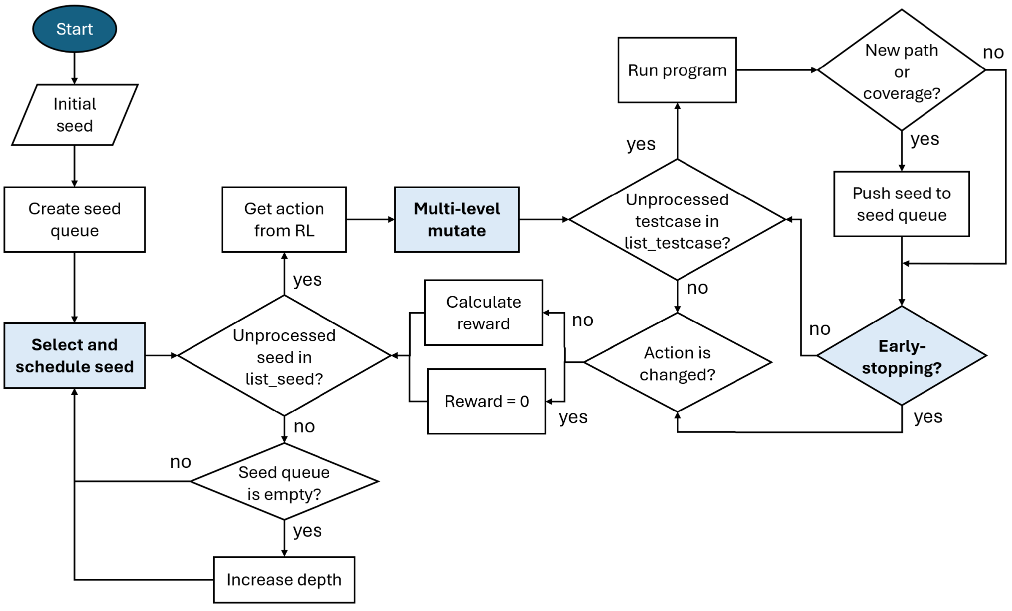


FIGURE 6. **The workflow of CTFUzz.**

still trying to mutate a seed with lower potential. This balances the exploitation and exploration of our model.

4) The model runs a loop, taking each seed and its allocated number of attempts as input for the RL model.
5) The RL model predicts and chooses an action that maximizes the new code coverage for the selected seed.
6) The model mutates the seed using the action chosen by the RL model and the number of attempts given by the scheduling algorithm, generating a series of test cases.
7) The target application is executed with the generated test cases, while the code coverage information is collected. Inputs that result in new code coverage are added to the queue to be used in the next iteration. During this phase, multi-level mutation and early stopping mechanisms are also implemented to optimize the fuzzing process.
8) After finishing fuzzing on a seed, the reward points are calculated and returned, and the next seed in the queue becomes the next state for the RL model.
9) Continue to step (4) until the iteration is completed, then return to step (2) to create a new loop to start a new iteration.

With the design of the proposed CTFuzz, we expect to enable a better fuzzing process. First, the RL algorithm can enable the fuzzing model to select better actions based on the criterion of increasing discovered code coverage. Second, the input selection and scheduling algorithm may assist in distributing test attempts according to the nature of inputs, prioritizing inputs with higher code coverage, and ensuring testing for lower-coverage inputs with the minimum number of attempts to balance the exploitation and exploration of the model. The multi-level mutation mechanism makes the model flexible and more practically effective for long-term and challenging fuzzing scenarios. Meanwhile, the waste prevention or early stopping mechanism helps avoid wasting time on inefficient inputs with excessive test attempts.

IV. IMPLEMENTATION AND EXPERIMENTS

A. RESEARCH QUESTION

Based on the improvements expected in the model presented above, in this experimental section, we will focus on answering the following questions.

- **Question 1:** How effective is the model compared to the previous RL-based fuzzing model?
- **Question 2:** What is the efficiency of the model compared to a modern fuzzing tool?
- **Question 3:** What is the contribution of the RL model used within the entire framework?
- **Question 4:** If the speed disparity between programming languages is improved, what will be the effectiveness of the model?

B. ENVIRONMENTAL SETTINGS

1) IMPLEMENTATION SETUP

We deploy the proposed model as well as perform experimentation on a VPS machine equipped with an Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz, boasting 4 cores, 64Â GBÂ RAM, and running Ubuntu 20.04.1 LTS 64-bit. On this system, we installed AFLplusplus version 4.05c, *Binutils* 2.34, *Poppler* 0.86.1, along with requisite libraries such as *tensorflow* 2.3.3, *gym* 0.10.3, *posix-ipc* 1.1.1, *keras-rl2* 1.0.5, *xxhash* 3.2.0, etc.

Moreover, the RL model is implemented with a DQN-based agent and an environment deployed using gym library [28]. More details of hyperparameters of the RL model

TABLE 2. Hyperparameters for training DQN-based fuzzing model.

Hyperparameter	Value
Size of input layer	Size of state space: 65536 (0x10000)
Size of output layer	Size of action space: 9
Optimizer	Adam
Discount factor γ	0.9
ϵ for EpsGreedyPolicy	0.7
Learning rate	0.001

TABLE 3. Settings of the mutation strategy for fuzzing.

Parameter	Value
total_energy	1,000,000
min_energy	50
C - Maximum selection times of an action	10
M - Maximum consecutive unsuccessful trial	5,000

are mentioned in **Table 2**, in which they are inherited from related work as well as defined based on the best results in multiple experiments. Meanwhile, the settings for mutation strategy are described in **Table 3**.

2) TARGET PROGRAMS FOR EVALUATION

In the scope of this paper, our investigation involved statistical analysis and fuzz testing of two toolsets, specifically Binutils-2.34 and Poppler-0.86.1, as outlined in **Table 4**. These toolsets are commonly used for testing in practical fuzzing research due to their open-source and easy-to-use nature, allowing for testing methods requiring source code access. The target applications used in our experiment are categorized into 2 main types, including PDF and ELF. Besides, they are not too excessive or complicated to require a dedicated fuzzer with specific customization for the fuzzing process. However, chosen target programs still possess a high potential for vulnerabilities because of their complex functionalities and diverse formats.

In addition, the initial criteria for the model targeted grey-box fuzzing of toolkits and software, therefore network services were not included in the objectives. Network services receive inputs differently and require a different approach when developing fuzzing tools. Contemporary fuzzing tools also have specific features for fuzzing network services. In terms of image processing tools, fuzzing via parsing software is quite similar to PDF file parsing. However, our tools do not yet support more complex operations like interactive image editing in terms of fuzzing based on events during UI application interactions.

3) EVALUATION METRICS

Our approach is evaluated via the four following metrics.

- **Code coverage** is calculated by the total number of edges found by each model with the edge-based coverage computation method. Each traversed edge is only counted once, ignoring later duplications during the fuzzing process. As mentioned above, a bitmap is returned to indicate the observed coverage to make up a global bitmap to record the edges found after each trial. A higher code coverage represents a better performance.
- **Unique paths** are measured for each test case, based on the set of edges it traverses (without taking the frequency of each edge traversed into account). This is an essential metric to evaluate the path-discovery capability of fuzzing models, reflecting the diversity of executed paths. Once again, more unique paths found indicate better effectiveness in our model.
- **Execution rate** is a parameter evaluating the speed of the models, which is identified by time-related information when trying inputs on the target programs. For example, it can be the total number of trials each model performed in a specific amount of time or the time it takes to finish a particular number of trials. In fact, given the inherently random nature of fuzzing, more trials conducted can increase the chances of discovering vulnerabilities or new paths as well as performance. In general, this metric can be calculated by the total number of executed trials per second. Hence, the higher speed of models is reflected via the higher number of trials performed or the less time-consuming.
- **Enhancement rate (ER)**: is used to determine the increase or decrease in performance of CTFuzz compared to other models, calculated by (4). According to this formula, a positive ER indicates that CTFuzz is more effective than its counterpart and vice versa, which can be used to compare all the above metrics.

TABLE 4. Target fuzzing programs.

ID	Target Program	Parameters	Version	Type
1	readelf	-a @@	Binutils-2.34	ELF
2	strings	-a @@	Binutils-2.34	ELF
3	size	-A -x -t @@	Binutils-2.34	ELF
4	objdump	-a -f -x -d @@	Binutils-2.34	ELF
5	nm	-C @@	Binutils-2.34	ELF
6	pdfinfo	-box @@	Poppler-0.86.1	PDF
7	pdfimages	-list -j @@	Poppler-0.86.1	PDF
8	pdfdetach	-list @@	Poppler-0.86.1	PDF
9	pdftotext	-htmlmeta @@	Poppler-0.86.1	PDF
10	pdftohtml	-stdout @@	Poppler-0.86.1	PDF
11	pdftoppm	-mono @@	Poppler-0.86.1	PDF

$$ER = \frac{Value\ of\ \ CTFuzz - Value\ of\ \ model\ X}{Value\ of\ \ model\ X} \times 100 \tag{4}$$

where:

- **Value of CTFuzz:** The value, which can be one of the above 3 metrics, of the CTFuzz model to be compared.
- **Value of model X:** The value, which can be one of the above 3 metrics, of the other models.

4) EXPERIMENTAL SCENARIOS

To address these four questions, we conducted experiments to evaluate the effectiveness of our model. In our experiments,

TABLE 5. Total edge-based coverage (number of found edges) after 6 hours.

ID	Program	CTFuzz	AFLplusplus		Random		rlfuzz	
		# Edges	# Edges	ER (%)	# Edges	ER (%)	# Edges	ER (%)
1	readelf	2,950	11,047	**-73.3**	2,815	4.8	2,278	29.5
2	strings	71.6	72	-0.6	71.6	0	70.4	**1.7**
3	size	65	65	0	61	6.6	56	16.1
4	objdump	335.6	240.6	**39.5**	332.4	-1	174,6	92.2
5	nm	100.4	119.6	-16.1	65.2	**54**	47.6	110.9
6	pdfinfo	149.2	149.4	-0.1	150.6	-0.9	41.8	256.9
7	pdfimages	55	55	0	55	0	52.4	5
8	pdfdetach	53	52.6	0.8	53	0	46.6	13.7
9	pdftotext	195.4	195.4	0	195.2	0.1	47	**315.7**
10	pdftohtml	326.2	319.2	2.2	322	**1.3**	207.8	57
11	pdftoppm	105.8	102.2	3.5	105.4	0.4	101.8	3.9
Average ER(%)			**-4.0**		**6.1**		**82.1**	

TABLE 6. Total number of unique paths found after 6 hours.

ID	Program	CTFuzz	AFLplusplus		Random		rlfuzz	
		# Paths	# Paths	ER (%)	# Paths	ER (%)	# Paths	ER (%)
1	readelf	28,753.6	689,824.4	**-95.8**	11,920.4	141.2	11,056.2	160.1
2	strings	11.4	68.6	-83.4	8.6	32.6	1.6	612.5
3	size	8.8	11.6	-24.1	4	120	2.8	214.3
4	objdump	362.4	477	-24	310.8	16.6	101.4	257.4
5	nm	13	93.2	-86.1	3.2	**306.3**	2.2	490.9
6	pdfinfo	30	43.4	-30.9	36	**-16.7**	2.6	**1,053.8**
7	pdfimages	3	4	-25	3.4	-11.8	2.6	**15.4**
8	pdfdetach	6	6.8	-11.8	5.8	3.4	3.6	66.7
9	pdftotext	25	60.2	-58.5	25.6	-2.3	3	733.3
10	pdftohtml	309.6	434	-28.7	242	27.9	111.4	177.9
11	pdftoppm	9.8	6.2	**58.1**	10.2	-3.9	6.2	58.1
Average ER(%)			**-37.3**		**55.8**		**349.1**	

TABLE 7. Execution speed (number of performed trials per second) after 6 hours.

ID	Program	CTFuzz	AFLplusplus		Random		rlfuzz	
		# Trials/s	# Trials/s	ER (%)	# Trials/s	ER (%)	# Trials/s	ER (%)
1	readelf	22.5	208.4	-89.2	23.3	-3.1	12.6	**78.7**
2	strings	48.5	286.3	-83.1	48.4	0.2	15	224.3
3	size	51.9	598	-91.3	53.1	-2.2	16.2	220
4	objdump	39.9	542.9	**-92.7**	39.2	1.8	13.9	187.1
5	nm	55.9	731.9	-92.4	57.5	-2.8	16.5	**237.8**
6	pdfinfo	37.8	200.8	**-81.2**	39.6	**-4.6**	14.7	157.2
7	pdfimages	37.6	237.3	-84.1	38.5	-2.2	14.7	155.9
8	pdfdetach	39.7	216.5	-81.7	41.2	-3.6	15.2	161.5
9	pdftotext	36.1	196.8	-81.6	35.4	**2.2**	14.7	145.3
10	pdftohtml	31.3	202.2	-84.5	32.2	-2.8	14.3	119.5
11	pdftoppm	28.6	210.1	-86.4	28.9	-1	13.4	112.6
Average ER (%)			**-86.2**		**-1.7**		**163.6**	

our proposed CTFuzz is put side by side with 3 other fuzzing tools, each of which is leveraged to answer one of the above questions. While the first question can be resolved by comparing CTFuzz with *rlfuzz* [10], another tool called *AFLplusplus* is used for the second one. Moreover, to address the third question, we compared CTFuzz with a randomly generated mechanism designed to separate the RL model from the overall model and evaluated the performance difference between selecting transformation actions based on RL and selecting them randomly. In the case of the final question, which is related to the performance of a programming language, we observe some metrics after 200,000 initial trials.

To sum up, 2 different scenarios are conducted as follows.

- **Scenario 1:** We compared the effectiveness of our model against various fuzzing models, including *rlfuzz* [10], *AFLplusplus* [27], and a random mechanism within the same execution time of 6 hours via all metrics and their ERs.
- **Scenario 2:** This scenario aims to answer the last question, where all models are compared using code coverage and unique path after completing 200,000 trials.

Note that, in each scenario, each model is tested 5 times to obtain averaged results.

C. EXPERIMENTAL RESULTS AND ANALYSIS

1) SCENARIO 1: PERFORMANCE AFTER 6 HOURS

Table 5 summarizes the number of edges found by each model after a 6-hour experimentation. In general, CTFuzz achieves lower code coverage compared to AFLplusplus when fuzzing most of the tested applications, while it beats *rlfuzz* regardless of target programs. For more details, the most significant decreased performance compared to AFLplusplus is observed in the *readelf* application, with 73.3%, while the best increase in code coverage can be observed in an application called *objdump* with the ER of 39.5%. Especially, in comparison with *rlfuzz*, our CTFuzz can even achieve a climb of 315.7%, which is 4 times more effective than its counterpart in terms of seeking new edges. On average of 11 applications, CTFuzz achieves lower coverage of 4.0% compared to AFLplusplus after 6 hours. In comparison with the random transformation model, CTFuzz experiences a slight increase in code coverage by 6.1% overall. However, the code coverage of the two models is nearly equivalent across all tested applications, except for the *nm* application, where the difference reaches up to 54% in favor of CTFuzz. Overall, compared with rlfuzz, CTFuzz exhibits a higher code coverage with an average increase of 82.1%.

When it comes to finding unique paths, **Table 6** contains information about the number of discovered paths for each model after 6 hours. Once again, CTFuzz trails behind AFLplusplus almost entirely, with the largest gap in the *readelf* application at -95.8%. However, there is an exception in the *pdftoppm* application, where CTFuzz increases by 58.1%. Despite the relatively high ratio, the actual difference is not substantial (9.8 compared to 6.2). On average, CTFuzz lags AFLplusplus by 37.3%. In comparison to the random transformation model, CTFuzz experiences a slight increase. The most significant boost is observed in the *nm* application with 306.3%, while the lowest decrease is -16.7% in *pdfinfo*. On average, CTFuzz surpasses the random model by 55.8% in terms of discovered path count across 11 applications. Concerning the *rlfuzz* model, CTFuzz once again outperforms all test applications. The highest superiority is in *pdfinfo* with 1,053.8% and the lowest is 15.8% in *pdfimages*. The average across the 11 applications is 349.1%.

In terms of execution speed, let's delve into the comparison in terms of execution speed, as illustrated in **Table 7**, based on the number of trial attempts per second. CTFuzz experiences a significant speed disadvantage, with an average number of trial attempts per second being 86.2% lower than that of *AFLplusplus*. As for the random transformation model, the difference is marginal at -1.7%. However, CTFuzz wins over *rlfuzz* model with 163.6% faster.

2) SCENARIO 2: PERFORMANCE AFTER 200,000 TRIALS

Given a fixed 200,000 initial trials, **Table 8** summarizes the performance of models in the fuzzing process in terms of discovered edges. In this context, CTFuzz outperforms AFLplusplus, exhibiting an improvement of over 6.5% on average in terms of the number of edges discovered. The largest disparity is observed in the *objdump* application, where CTFuzz achieves a lead of more than 47.3%. For the *rlfuzz* model, CTFuzz continues to win over when fuzzing most applications, showing an average improvement of 82.5%. The highest figure is seen in the *pdftotext* application, with an increase of 314%, without any lower performance observed. With the random transformation mechanism, the overall difference remains relatively small, and the coverage achieved in various applications is nearly comparable. Notably, in the case of the nm application, CTFuzz exhibits a significant increase of 70.8% in edge coverage compared to the random mechanism. On average, across 11 applications, CTFuzz achieves more than 9.4% higher edge coverage than the random mechanism after 200,000 executions.

In terms of the number of discovered execution paths within the first 200,000 execs as shown in **Table 9**, CTFuzz continues to demonstrate a slight advantage over AFLplusplus, exhibiting an increase of 8.2%. The disparity in improvement varies across applications. The largest boost was seen in *objdump* at 97.6%, and the most marginal in *pdftotext* at -34.6%. Compared to the random transformation mechanism, CTFuzz surpasses the average by 33.4%, achieving the highest increase in the *readelf* application at 169.8%, and the lowest in *pdfdetach* at -15.4%. For the *rlfuzz* model, the average increase across 11 applications is 350.4%. Notably, in the *pdftotext* application, there is a substantial difference of 1,560% due to rlfuzz discovering only one path compared to 16.6 paths found by CTFuzz.

Moreover, we proceed to compare the execution speeds of the four models within the first 200,000 trials in the form of the number of executed trials per second, as illustrated in **Table 10**. The increased ratios of CTFuzz in comparison to AFLplusplus, the random transformation mechanism, and rlfuzz are -86.3%, -3.5%, and 154.6%, respectively.

V. DISCUSSION

According to the nature of designed experiments, it is essential to provide a few general observations about the experimental results, as follows.

- The 6-hour timeframe is relatively short to have a comprehensive comparison of the effectiveness of the fuzzing models in a real-world scenario, where practical fuzzing projects can span over weeks, utilizing multiple machines with significantly higher speeds. However, this duration is sufficient to highlight differences between the models, meanwhile, running each application three times helps mitigate the impact of luck.
- Many applications achieved similar values across all four models and did not show significant improvement regardless after 200,000 trials or 6 hours. This could be due to the less effective initial inputs used or the relatively short timeframe.
- Some comparison values exhibit minor differences in effectiveness in terms of improvements. However, since the base values being compared against were low, the resulting improvement ratios appear substantial, impacting the final average numbers.

When it comes to addressing the three pre-defined research questions, the above experiment results lead us to the following answers.

TABLE 8. Total code coverage (number of found edges) after 200,000 trials.

ID	Program	CTFuzz	AFLplusplus		Random		rlfuzz	
		# Edges	**# Edges**	**ER (%)**	**# Edges**	**ER (%)**	**# Edges**	**ER (%)**
1	readelf	2,739.2	3,444.6	**-20.5**	2,512.6	9	2,194.6	24.8
2	strings	70.8	71.6	-1.1	70.4	0.6	70.4	**0.6**
3	size	63.4	65	-2.5	61	3.9	56	13.2
4	objdump	291.6	198	**47.3**	246.8	18.2	160.2	82
5	nm	88.8	64.6	37.5	52	**70.8**	47.6	86.6
6	pdfinfo	148.6	148.4	0.1	147.4	0.8	41	262.4
7	pdfimages	55	55	0	55	0	52.4	5
8	pdfdetach	51	51	0	52.6	**-3**	46.6	9.4
9	pdftotext	194.6	194.8	-0.1	193.6	0.5	47	**314**
10	pdftohtml	311.2	295	5.5	305.4	1.9	151.2	105.8
11	pdftoppm	105.4	99.8	5.6	104.6	0.8	101.4	3.9
Average ER (%)			**6.5**		**9.4**		**82.5**	

TABLE 9. Total number of unique paths found after 200,000 trials.

ID	Program	CTFuzz	AFLplusplus		Random		rlfuzz	
		# Paths	**# Paths**	**ER (%)**	**# Paths**	**ER (%)**	**# Paths**	**ER (%)**
1	readelf	13,111.4	18,269.6	-28.2	4,858.8	**169.8**	9,033.8	45.1
2	strings	4.6	5	-8	2.8	64.3	1.6	187.5
3	size	7	7.8	-10.3	4	75	2.8	150
4	objdump	149	75.4	**97.6**	111.4	33.8	79.8	86.7
5	nm	3	4	-25	3	0	2.2	36.4
6	pdfinfo	22.4	23.6	-5.1	24.4	-8.2	1.4	1,500
7	pdfimages	3	3	0	3	0	2.6	**15.4**
8	pdfdetach	4.4	3.8	15.8	5.2	**-15.4**	2.8	57.1
9	pdftotext	16.6	25.4	**-34.6**	16.4	1.2	1	**1,560**
10	pdftohtml	148.2	153.2	-3.3	111	33.5	54.6	171.4
11	pdftoppm	8.4	4.4	90.9	7.4	13.5	5.8	44.8
Average ER (%)			**8.2**		**33.4**		**350.4**	

TABLE 10. Execution speed (number of performed trials per second) after 200,000 trials.

ID	Program	CTFuzz	AFLplusplus		Random		rlfuzz	
		# Trails/s	**# Trails/s**	**ER (%)**	**# Trails/s**	**ER (%)**	**# Trails/s**	**ER (%)**
1	readelf	21.9	431.1	**-94.9**	22.7	-3.6	12.6	**73.6**
2	strings	46.4	212.6	**-78.2**	47.3	-1.9	15.1	207.5
3	size	47.4	523.8	-91	53.1	**-10.7**	16.2	191.9
4	objdump	38.4	454.8	-91.6	37.9	**1.3**	13.9	177.1
5	nm	55	666.7	-91.7	56.6	-2.8	16.5	**234.6**
6	pdfinfo	37	198	-81.3	36.7	0.9	14.7	151.4
7	pdfimages	36	232.8	-84.5	38.3	-5.8	14.7	145.2
8	pdfdetach	38.1	220.9	-82.8	38.7	-1.7	15.1	152.3
9	pdftotext	34.5	198.5	-82.6	35.5	-2.9	14.7	135.2
10	pdftohtml	31.3	208.8	-85	31.6	-0.8	14.3	119.1
11	pdftoppm	28.6	204.1	-86	31.9	-10.4	13.4	113.1
Average ER (%)			**-86.3**		**-3.5**		**154.6**	

Question 1: How effective is the model compared to the previous RL-based fuzzing model?

This question can be answered using the result of the comparison of CTFuzz and rlfuzz, another RL-based model. Clearly, in both 6-hour and 200,000-trial experiments, it is evident that the CTFuzz model consistently outperforms the rlfuzz model in all terms of code coverage, unique path count, and execution speed. CTFuzz's code coverage improves by approximately 80% compared to rlfuzz, and even surpasses rlfuzz in the number of discovered paths by 411.8% in the initial 200,000 trials and by 658.6% within the 6-hour timeframe. In terms of execution speed, CTFuzz exhibits approximately 160% faster performance in both contexts. These results demonstrate the higher effectiveness of CTFuzz than that of rlfuzz in both the exploitation and exploration aspects of the model.

Question 2: How does the model compare to a state-of-the-art fuzzing tool?

Take AFLplusplus as an example of a modern fuzzing tool, the effectiveness of CTFuzz slightly falls behind in all evaluation metrics. However, when examining the effectiveness on a per-run basis, CTFuzz achieves slightly better results than AFLplusplus. This indicates that if the speed of the CTFuzz model can be improved without compromising its overall performance, it has the potential to be highly useful.

With the rising popularity of ChatGPT and application, ChatGPT-assisted fuzzers can also be promising solutions.

Despite its massive database and diverse knowledge obtained from various sources, there are still limitations when considering ChatGPT in our approach. For instance, the provided APIs of ChatGPT require expenses and may lead to the proposed fuzzing tool dependent on network speed. Besides, the fuzzing process may define various restraints such as data and source code privacy and security, then it can be risky when based on a third-party approach like ChatGPT.

Question 3: What is the trade-off regarding speed that yields performance improvements?

In the 6-hour and 200,000-trial contexts, the speed difference between CTFuzz and the random mechanism remains marginal, at approximately 0.7%. Despite this small delay, CTFuzz enhances code coverage and path discovery slightly, highlighting the value of the RL mechanism. However, the speed trade-off does not appear to be significant.

Question 4: How would the model's effectiveness change if the speed gap between languages is reduced?

Comparing the execution speeds of the four models reveals a considerable discrepancy between AFLplusplus (implemented in C) and the other models (implemented in Python). However, the speed difference between CTFuzz and the random mechanism is minor, around 0.7%. This indicates that the primary contributor to the speed difference is the Python language rather than the RL component. Considering the slight advantage of CTFuzz over AFLplusplus in per-run effectiveness, optimizing the model's execution speed could yield promising results.

In conclusion, the achieved results partially reflect our expectations for the fuzzing model. CTFuzz boasts a higher execution speed, greater code coverage, and an increased number of found paths compared to the rlfuzz model in both the 6-hour and 200,000-trial experiments. Although CTFuzz operates slightly slower than the random mechanism, it simultaneously improves code coverage and path count. This suggests the RL mechanism has some positive effects, though not overwhelmingly substantial. While the discussion has provided valuable insights, further detailed comparisons and evaluations between different RL models and their hyperparameters are necessary for optimal and efficient fuzzing. Additionally, some applications reached stagnation in code coverage and path discovery, indicating either the initial input ineffectiveness or the need for longer experimentation periods.

Regarding time constraints, we have not conducted comprehensive comparisons and performance evaluations across different RL models, nor have we thoroughly assessed the hyperparameters. Consequently, the choices may not have resulted in optimal and efficient fuzzing. Moreover, several applications exhibited coverage and path discovery stagnation, which suggests that either the initial inputs employed were ineffective for those applications or the experimentation duration was insufficient for the model to explore new paths.

VI. FUTURE DIRECTIONS

Much more research effort is needed for RL-based fuzzing models to be effectively applicable in practice. One of the significant challenges is speed, which can be possibly improved by transitioning the model to other faster programming languages, like Rust. Moreover, our proposed fuzzer can be redesigned in a multithreaded processing manner to speed it up significantly. Besides, a comprehensive comparison with other RL models with more complicated architecture or using various RL algorithms is also a promising direction. Besides, instead of focusing on input selection and scheduling, the process of input creation can also be enhanced by deploying state-of-art data generation techniques, such as Generative adversarial networks (GANs) in RapidFuzz [18], CGFuzzer [19]. Such GANs-based approaches can be used to learn data structures to effectively create input, which is useful for dealing with complicated input types or strict input verification in some target programs.

In the evolving landscape of operating system security, in addition to software fuzzing in userland mode, kernel fuzzing has emerged as a critical technique for uncovering vulnerabilities that could potentially impact billions of devices globally. One of the foundational tools in this domain is Syzkaller, which utilizes a domain-specific language, syzlang, to meticulously define system call (syscall) sequences and their interdependencies. Despite the progress in automating kernel fuzzing, the generation of Syzkaller specifications has largely remained a manual endeavor, with a significant number of syscalls still not effectively covered. Recognizing this gap, the recent study of Chenyuan Yang et al. introduced in KernelGPT approach [29], marks a significant advancement. It harnesses the capabilities of Large Language Models (LLMs) to infer Syzkaller specifications, leveraging the extensive kernel code, documentation, and use case data encoded during the pre-training of these models. The KernelGPT model utilizes an iterative method to derive and refine syscall specifications, integrating feedback mechanisms to enhance the accuracy and comprehensiveness of the generated sequences. Preliminary findings highlighted in the study indicate that KernelGPT achieves greater coverage and identifies numerous previously undetected bugs, leading to a collaboration with the Syzkaller team to incorporate these automatically inferred specifications. This development underscores the potential of LLMs to transform the scope and efficacy of kernel fuzzing practices, providing a more automated and systemic approach to securing operating systems against a broad spectrum of threats.

In terms of protocol fuzzing, pretrained LLMs have shown significant potential in advancing fuzzing protocols, particularly in overcoming the limitations of traditional mutation-based fuzzing techniques. By leveraging the extensive knowledge embedded within LLMs in the ChatAFL approach by Meng et al. [30], which have been trained on vast amounts of human-readable protocol specifications, it becomes possible to extract machine-readable information that aids in generating valid and diverse test inputs. This is especially useful given that protocol implementations often lack formal, machine-readable specifications, relying instead on extensive natural language documentation. The LLM-guided approach enhances state and code coverage by constructing grammar for various message types within a protocol and predicting subsequent messages in a sequence.

The potential of LLMs like GPT in enhancing fuzzing techniques extends beyond kernel security to help fuzzers encompass userland applications in binary software and protocols [30]. LLMs demonstrate a profound capability to understand and generate complex code patterns, which can be leveraged to automate the generation of fuzzing inputs for userland binary applications. This is particularly valuable in scenarios where conventional fuzzing struggles due to the complexity of the input structures required by these applications. By automating input generation, LLMs can uncover vulnerabilities that might be missed by more traditional methods, thus broadening the scope of security testing in userland environments. Furthermore, the adaptability of LLMs to understand context from documentation and prior code enables them to tailor fuzzing approaches to the specific nuances of userland binaries. This enhances the efficacy and coverage of fuzz tests, pushing the boundaries of what can be achieved with current technology in identifying and mitigating potential software vulnerabilities.

In summary, while ChatGPT and other LLMs can enhance certain aspects of the fuzzing process, especially in test case generation and automation [31], its effectiveness in fuzzing binary software is limited by its lack of specialized knowledge in low-level computing and potential scalability issues. Hence, to make it more suitable for fuzzing, LLMs need to be customized, fine-tuned, and pretrained in specific large-scale datasets. In our future works, it can be best used as a supplementary tool alongside more specialized fuzzing tools and frameworks. Specifically, in the test case generation task for fuzzing, ChatGPT or other LLMs can significantly enhance the process by leveraging its language processing capabilities to create diverse and contextually appropriate inputs. It can generate various forms of user-like data, develop complex user scenarios, and create semantic variants of input data, which are critical in exploring different execution paths in software. Additionally, LLMs can aid in automating test script writing and can integrate and interpret outputs from other tools to suggest relevant test cases, thereby enriching the depth and coverage of the fuzzing process in uncovering potential vulnerabilities. We also intend to improve our work with the LLM model in the future, to boost fuzzing performance in many open-source software projects, kernel applications, and protocols.

VII. CONCLUSION

Combining RL and current fuzzing techniques holds the potential to accelerate fuzzing effectively. However, there are still significant limitations that hinder its practicality. These include the imbalance between exploitation and exploration in the model, slow speed, and the absence of comparative studies regarding practical effectiveness against real-world fuzzing tools. In this article, we introduce an RL-based fuzzing model with code coverage that can balance the exploitation and exploration aspects by efficient input selection and scheduling algorithms. Moreover, to improve efficiency, multi-level input mutation algorithms and early termination mechanisms are also implemented. The effectiveness of our proposed CTFuzz model has been demonstrated via experiment results, where it is compared with other modern fuzzing tools or RL-based fuzzing models in the capability of discovering new paths, increasing code coverage as well as time-effectiveness. Despite its potential outcomes, the contribution of CTFuzz is still bounded, such as insufficiently rapid speed, limited length of test cases, only aiming to binary software, and insignificant improvement compared to the random mechanism, all of which need to be considered in future efforts.

REFERENCES

[1] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.

[2] F. Rustamov, J. Kim, J. Yu, and J. Yun, "Exploratory review of hybrid fuzzing for automated vulnerability detection," *IEEE Access*, vol. 9, pp. 131166–131190, 2021.

[3] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, pp. 1–36, Jan. 2022.

[4] S. Mallissery and Y.-S. Wu, "Demystify the fuzzing methods: A comprehensive survey," *ACM Comput. Surv.*, vol. 56, no. 3, pp. 1–38, Mar. 2024.

[5] X. Zhao, H. Qu, J. Xu, X. Li, W. Lv, and G.-G. Wang, "A systematic review of fuzzing," *Soft Comput.*, vol. 28, no. 6, pp. 5493–5522, Mar. 2024.

[6] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.

[7] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2021.

[8] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018.

[9] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, Aug. 2020, pp. 2307–2324.

[10] Z. Zhang, B. Cui, and C. Chen, "Reinforcement learning-based fuzzing technology," in *Proc. 14th Int. Conf. Innov. Mobile Internet Services Ubiquitous Comput. (IMIS)*. Cham, Switzerland: Springer, 2020, pp. 244–253.

[11] J. Ye, R. Li, and B. Zhang, "RDFuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation," *Math. Problems Eng.*, vol. 2020, pp. 1–12, Mar. 2020.

[12] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu, "A systematic review of fuzzing based on machine learning techniques," *PLoS ONE*, vol. 15, no. 8, Aug. 2020, Art. no. e0237749.

[13] J. Hao, T. Yang, H. Tang, C. Bai, J. Liu, Z. Meng, P. Liu, and Z. Wang, "Exploration in deep reinforcement learning: From single-agent to multiagent domain," *IEEE Trans. Neural Netw. Learn. Syst.*, early access, Jan. 19, 2023, doi: 10.1109/TNNLS.2023.3236361.

[14] T. Ji, Z. Wang, Z. Tian, B. Fang, Q. Ruan, H. Wang, and W. Shi, "AFLPro: Direction sensitive fuzzing," *J. Inf. Secur. Appl.*, vol. 54, Oct. 2020, Art. no. 102497.

[15] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing deeply nested branches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 499–513.

[16] H. Zhang, A. Zhou, P. Jia, L. Liu, J. Ma, and L. Liu, "InsFuzz: Fuzzing binaries with location sensitivity," *IEEE Access*, vol. 7, pp. 22434–22444, 2019.

[17] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: Survey, challenges and future directions," *Comput. Secur.*, vol. 120, Sep. 2022, Art. no. 102813.

[18] A. Ye, L. Wang, L. Zhao, J. Ke, W. Wang, and Q. Liu, "RapidFuzz: Accelerating fuzzing via generative adversarial networks," *Neurocomputing*, vol. 460, pp. 195–204, Oct. 2021.

[19] Z. Yu, H. Wang, D. Wang, Z. Li, and H. Song, "CGFuzzer: A fuzzing approach based on coverage-guided generative adversarial networks for industrial IoT protocols," *IEEE Internet Things J.*, vol. 9, no. 21, pp. 21607–21619, Nov. 2022.

[20] Y. Wang, Z. Wu, Q. Wei, and Q. Wang, "NeuFuzz: Efficient fuzzing with deep neural network," *IEEE Access*, vol. 7, pp. 36340–36352, 2019.

[21] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2018, pp. 116–122.

[22] A. Kuznetsov, Y. Yeromin, O. Shapoval, K. Chernov, M. Popova, and K. Serdukov, "Automated software vulnerability testing using deep learning methods," in *Proc. IEEE 2nd Ukraine Conf. Electr. Comput. Eng. (UKRCON)*, Jul. 2019, pp. 837–841.

[23] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. (ICSE)*, Oct. 2020, pp. 1410–1421.

[24] X. Liu, R. Prajapati, X. Li, and D. Wu, "Reinforcement compiler fuzzing," in *Proc. ICML Workshop*, 2019.

[25] W. Drozd and M. D. Wagner, "FuzzerGym: A competitive framework for fuzzing and learning," 2018, *arXiv:1807.07490*.

[26] *LibFuzzer—A Library for Coverage-Guided Fuzz Testing*. Accessed: Dec. 2023. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[27] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. (2020). *American Fuzzy Lop Plus Plus (AFL++)*. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus

[28] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," 2016, *arXiv:1606.01540*.

[29] C. Yang, Z. Zhao, and L. Zhang, "KernelGPT: Enhanced kernel fuzzing via large language models," 2024, *arXiv:2401.00563*.

[30] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2024.

[31] Y. Tang, Z. Liu, Z. Zhou, and X. Luo, "ChatGPT vs SBST: A comparative assessment of unit test suite generation," *IEEE Trans. Softw. Eng.*, vol. 50, no. 6, pp. 1340–1359, Jun. 2024.

VAN-HAU PHAM received the bachelor's degree in computer science from the University of Natural Sciences, Ho Chi Minh City, Vietnam, in 1998, and the master's degree in computer science from the Institut de la Francophonie pour l'Informatique (IFI), Vietnam, in 2004. Then, he did his internship and worked as a full-time Research Engineer in France for two years. Then, he persuaded his Ph.D. thesis on network security under the direction of Professor Marc Dacier, from 2005 to 2009. Then, he did his internship and worked as a full-time Research Engineer in France for two years. Currently, he is a Lecturer with the University of Information Technology, Vietnam National University Ho Chi Minh City (UIT-VNU-HCM). His main research interests include network security, system security, mobile security, and cloud computing.

DO THI THU HIEN received the B.Eng. degree in information security from the University of Information Technology, Vietnam National University Ho Chi Minh City (UIT-VNU-HCM), in 2017, and the M.Sc. degree in information technology, in 2020. Since 2017, she has been a member with the Research Group, Information Security Laboratory (InSecLab), UIT. Her research interests include malware analysis and detection, information security and privacy, software-defined networking, and its related security-focused problems.

NGUYEN PHUC CHUONG received the degree in information security from the Honor Program, University of Information Technology, Vietnam National University Ho Chi Minh City (UIT-VNU-HCM), Vietnam, in 2023. Since 2022, he has been actively engaged as a Student Member with the Information Security Laboratory (InSecLab), UIT-VNU-HCM, with a focus on projects related to information security and AI-driven security. His primary research interests include software security, automatic exploitation, and AI cybersecurity endeavors, particularly in fuzzing techniques for binary exploitation with AI.

PHAM THANH THAI received the degree in information security from the Honor Program, University of Information Technology, Vietnam National University Ho Chi Minh City (UIT-VNU-HCM), Vietnam, in 2023. Since 2022, he has been actively engaged as a Student Member with the Information Security Laboratory (InSecLab), UIT-VNU-HCM, with a focus on projects related to information security and AI-driven security. His primary research interests include cybersecurity and AI cybersecurity endeavors, particularly in fuzzing techniques for binary exploitation with AI.

PHAN THE DUY received the B.Eng. degree in software engineering and the M.Sc. degree in information technology from the University of Information Technology (UIT), Vietnam National University Ho Chi Minh City (VNU-HCM), Ho Chi Minh City, Vietnam, in 2013 and 2016, respectively. Currently, he is pursuing the Ph.D. degree in information technology, specializing in cybersecurity with UIT. In 2016, he worked as a Research Member with the Information Security Laboratory (InSecLab), UIT-VNU-HCM, after five years in the industry, where he joined and created several security-enhanced and large-scale teleconference systems. His main research interests include concentrating on cybersecurity and privacy problems, including software security, software-defined networking (SDN), malware and cyber threat detection, digital forensics, machine learning and adversarial machine learning in cybersecurity domains, private machine learning, and blockchain.