IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 2 May 2024, accepted 20 May 2024, date of publication 24 May 2024, date of current version 3 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3404918

RESEARCH ARTICLE

LinFuzz: Program-Sensitive Seed Scheduling Greybox Fuzzing Based on LinUCB Algorithm

YINGHAO SU[1], DAPENG XIONG[2], YING WAN[2], CHENGHAO SHI[2], AND QINGYAO ZENG[1]
[1]Institute of Graduate, Space Engineering University, Beijing 101416, China
[2]Institute of Aerospace Information, Space Engineering University, Beijing 101416, China

Corresponding author: Dapeng Xiong (xiongdapeng@hgd.edu.cn)

This work was supported by the Science and Technology on Complex Electronic System Simulation Laboratory under Grant 614201002012204.

ABSTRACT The mutation-based greybox fuzz testing technique is one of the widely used dynamic vulnerability detection techniques. It generates testcases for testing by mutating input seeds. In the process of fuzz testing, the seed scheduling strategy and energy scheduling strategy impact the test results and efficiency. Existing seed scheduling strategies, however, only consider a few specific seed attributes and ignore contextual information during seed execution. This oversight makes it challenging to prioritize the selection of suitable seeds based on historical fuzz test results. Meanwhile, current methods for calculating coverage lack evaluation of software paths, which makes it easy to waste time on testing high-frequency and low-risk paths. This article proposes a new greybox fuzzing scheme, LinFuzz, which transforms the seed scheduling problem into a contextual multi-armed bandit machine model. It utilizes the LinUCB algorithm to assess the value of seeds for scheduling by considering their historical execution information. At the same time, LinFuzz improves the calculation method for fuzz testing path rewards and the seed energy scheduling algorithm. It allocates more energy for testing low-frequency paths in the testing program, thereby enhancing the efficiency of exploration and the path coverage ability of the testing tool. This article evaluated the proposed LinFuzz on 12 real programs in comparison with other open-source tools such as AFL, AFLFast, FairFuzz, Neuzz, etc. The results show that under the same testing time budget, LinFuzz outperforms other tools in terms of vulnerability discovery quantity and code coverage ability. Compared with complex fuzz testing optimization algorithms, LinFuzz has lower memory consumption and time complexity.

INDEX TERMS Coverage-guided fuzzing, LinUCB algorithm, seed scheduling.

I. INTRODUCTION

Fuzz testing, a dynamic method for uncovering vulnerabilities, has been widely employed in software testing due to its effectiveness and the absence of the necessity for static analysis of associated programs [1]. Fuzz testing is an approach that utilizes automation to produce random inputs, execute the test program, and monitor for exceptions to identify program defects. One of the most advanced testing techniques in this area is coverage-guided greybox fuzzing (CGF), which has been widely adopted by the industry and has proven effective in detecting tens of thousands of vulnerabilities in commonly used software.

In the coverage-guided fuzz testing, the fuzzer initially maintains a queue of seeds. Testcases for software are generated from seed files. Each execution of a testcase leads to the activation of a distinct execution pathway within the target program. Subsequently, the fuzzer enhances the performance of fuzz testing by allocating mutation energy to these seeds. Numerous studies have shown that there are differences in the effectiveness of seeds in generating compelling test cases. These test cases can induce distinct execution paths or cause crashes in the target program. The effectiveness of fuzzing in exploiting program vulnerabilities varies, as different test cases traverse distinct execution paths. Seed scheduling

The associate editor coordinating the review of this manuscript and approving it for publication was Chi-Yuan Chen.

optimization is widely recognized as a highly effective strategy for enhancing the efficiency of fuzzing [2]. Therefore, the implementation of a carefully designed seed scheduling and energy allocation strategy is essential for effectively generating a greater number of intriguing test cases. However, establishing a definitive causal relationship between vulnerabilities and seed files is challenging due to the inherent uncertainty associated with vulnerabilities in software. The presence of uncertainty in identifying the seed most likely to result in a crash adds complexity to the problem of optimizing seed scheduling. This uncertainty also makes the problem unsolvable from a mathematical perspective [3].

The classic seed scheduling strategy primarily relies on a single indicator to calculate seed fitness for guiding seed scheduling and prioritizing sorting [4], [5], [6], [7]. This method evaluates the relative value of candidate seeds in the test sequence by establishing a concise and intuitive measurement standard. These standards are based on characteristics such as coverage, number of executions, and execution speed, which guide the sorting and scheduling of seeds in the execution queue accordingly. Although this type of single-indicator-oriented seed scheduling strategy has the advantages of convenient implementation and easy understanding, relying on a single indicator to guide fuzz testing may not necessarily yield favorable results due to the different characteristics of the testing target program. This means that it may not fully capture the true potential of seeds in exploring deep-seated vulnerabilities or revealing complex program behaviors.

In order to address the limitations of scheduling methods based on the single fitness of seeds, researchers have utilized multi-armed bandit (MAB) modeling techniques to solve the seed scheduling problem [8], [9], [10]. The main obstacle encountered by seed schedulers involves the complex balance between seed exploration and exploitation [3]. On the one hand, using new seeds has the potential to achieve broader coverage. On the contrary, seeds with high fitness that have already been explored may be more prone to discovering vulnerabilities during the fuzzing process.

However, the MAB model did not fully utilize the feedback information from the fuzz testing process in making seed scheduling decisions. Epsilon-Greedy and UCB algorithms, based on the MAB model, only calculate rewards for seed scheduling and energy allocation based on the coverage of historical seed tests [11]. This makes it difficult to select the optimal seed based on the characteristics of the testing program and historical fuzz testing results [7]. These methods not only delay the selection of seeds that may lead to crashes but also reduce the resources allocated to effective mutation test cases, thereby impacting efficient crash detection. At the same time, fuzz testing tools utilize edge coverage as a seed value evaluation indicator, while disregarding the execution frequency of seed edges and the hazard function in basic blocks. They typically consume a significant number of resources for high-frequency and low-risk paths, which reduces the efficiency of fuzz testing.

To address the aforementioned challenges, this paper proposes LinFuzz to tackle the mentioned issues. In each iteration of fuzz testing, LinFuzz systematically records the attribute feature information of seeds, converts it into context vectors, and then uses the LinUCB algorithm to accurately estimate the potential value of each seed in the seed queue. This process helps determine the priority order of seed execution. To overcome the limitations of edge coverage evaluation indicators that cannot effectively reflect path execution frequency and potential risks, the LinFuzz proposed in this study innovatively adopts a path reward mechanism based on path frequency and risk. Under this mechanism, LinFuzz quantifies the frequency of each path execution and the level of risk that may trigger vulnerabilities, using this as the standard for the reward of the path. During fuzz testing, LinFuzz utilizes a path reward strategy to direct the intelligent distribution of seed energy. This strategy prioritizes allocating resources to paths with low-frequency and high-risk functions. Additionally, this reward mechanism is incorporated into the selection process of the seed queue to ensure the preservation of seeds that cover more valuable paths. In summary, our main contributions are described below:

- In order to fully utilize program execution and seed attribute information, we model the seed scheduling problem as a contextual multi-armed bandit machine model. We design a seed scheduling strategy based on the LinUCB algorithm, use context vectors to calculate expected seed returns, and aim to achieve optimal seed scheduling.
- We propose a program path reward calculation method based on path frequency and danger, which evaluates the program path value according to program execution frequency and the associated danger functions, effectively guiding fuzz testing. Meanwhile, our method is lightweight and consumes fewer resources compared to stain analysis and symbol execution methods.
- We have developed a seed energy allocation algorithm and a queue cull algorithm based on path value. These algorithms aim to enhance the exploration of low-frequency and high-risk paths while maintaining the exploration capacity of high-value seeds effectively.
- We have developed and implemented a new fuzz testing tool called LinFuzz, which is a greybox fuzz testing tool based on all the methods mentioned above. We compared LinFuzz with open-source fuzz testing tools such as AFL. We tested LinFuzz using the LAVA dataset and 12 real programs. Prove the effectiveness of LinFuzz in terms of discovering paths and the number of vulnerabilities.

The remainder of this paper is organized as follows.

In Section II, we provide an overview of Fuzz testing, the MAB model, and the LinUCB algorithm. Section III describes the specific architecture of the LinFuzz, including the LinUCB based seed scheduling algorithm, path value evaluation algorithm, energy allocation and queue cull algorithm used by LinFuzz. In Section IV, we evaluate the prototype system using the LAVA-M test suite and real programs and compare it with other fuzz testing tools. Section V

discusses the limitations of our approach. Section VI introduce the related work about the improve of fuzzing test.

II. BACKGROUND

A. COVERAGE GUIDED GREYBOX FUZZ TESTING BASED ON MUTATION

Mutation-based fuzz testing generates test data by modifying the content of seed files. Fuzz testing tools manipulate seed samples through operations such as bit flipping, mathematical operations, insertion, deletion, and concatenation, resulting in new testcases. This approach explores more program execution paths, achieving higher code coverage and discovering potential vulnerabilities in the software.

AFL [12] is a representative mutation-based greybox fuzzer. It implements dynamic instrumentation technology on each basic block during the compilation of program source code to collect real-time branch coverage information during program execution. This information guides its fuzz testing process. During the execution of fuzz testing in AFL, the program maintains a seed queue and follows a predetermined scheduling strategy to extract the test seed file. After selecting the seed file, AFL allocates energy to it. In AFL, the energy given to each seed is fixed. AFL mutates the seeds to generate a large number of test cases, which are then used to execute the target program. During the execution process, AFL relies on feedback information provided by pre-implanted instrumentation code to record and track the response of the target program to different test cases and the execution paths it traverses. AFL applies the coverage-oriented principle to evaluate the value of the test cases obtained. For intriguing test cases that cover new execution paths or trigger anomalies in program behavior, they are added to the seed queue for additional iterations and more thorough testing. Other test cases that fail to yield effective exploration increments are discarded. After completing the entire testing cycle for a seed file, the fuzz testing tool selects the next seed from the queue based on a loop mechanism for a new round of testing activities. This process continues until the preset testing time limit is reached or an external stop command is received. Seed scheduling is an important method to enhance the efficiency of coverage-guided fuzz testing. Seed scheduling is an important method to enhance the efficiency of coverage-guided fuzz testing.

Based on the analysis above, it can be seen that there is still room for improvement in AFL in terms of seed scheduling and seed energy allocation strategies. When selecting seeds, AFL only considers factors such as execution time and coverage, and cannot adaptively adjust seed selection and energy allocation strategies [12]. Usually, some seeds are continuously allocated more energy than the minimum required to discover new paths, resulting in energy waste. Based on AFL, we design improved seed scheduling strategy and energy scheduling strategy to improve the efficiency and detection rate of fuzzing.

B. CONTEXTUAL MULTI-ARMED BANDIT MODEL

The Multi-Armed Bandit model originates from probability theory and game theory, providing a fundamental theoretical framework for gradually learning optimal resource allocation strategies in uncertain and dynamically changing environments [13]. This model assumes that there are N bandit machines in front of the gambler, and he does not know the true profit situation of each bandit machine beforehand. He needs to choose which machine to pull down based on the results of each bandit machine play in order to maximize the final profit. This involves striking a balance between exploring unknown options and utilizing known information to achieve long-term benefit maximization.

The contextual bandit machine is an extension of the multi-armed bandit machine model. It takes into account the interaction between different bandit machines and the influence of environmental factors [14]. In contextual bandit machines, each bandit machine has contextual information related to it, such as time, location, user information, etc. This contextual information can affect the probability of profitability for each bandit machine. Therefore, when choosing the next bandit machine, gamblers need to consider the current contextual information, as well as previous choices and results, to determine their next action [15].

The context multi-armed bandit machine problem can be represented as a tuple $<A,X,R>$, where:

A is a set of actions, which includes all executable actions. Let a multi-armed bandit machine have a total of K pull rods, defined $a_i \in A$, $i = \{1, 2, \ldots, K\}$, to represent any action and obtain a set of action spaces $\{a_1, \ldots, a_K\}$.

X is the set of context vectors. For each action $a_i \in A$, the contextual multi arm bandit machine model defines feature vectors $\boldsymbol{x_{a,t}}$ for the contextual information of action a during the experimental process.

R is the reward probability distribution, and each action a_t corresponds to a specific reward $R(r|a)$ probability distribution. The distribution of rewards for different levers is typically varied.

Assuming that only one action can be selected at each moment, the goal of the model is to dynamically optimize the action selection strategy based on contextual information, and maximize the cumulative reward $\max \sum_{t=1}^{T} r_t$, $r_t \sim R(\bullet \mid a_t)$ income within a certain number of time steps t.

There are several issues with modeling seed scheduling as Multi-Armed Bandit models. First of all, when applying the MAB model, fuzzing models the seeds as the pull rod in the middle. But with the progress of fuzzing, the number of interesting seeds continue to increase, making it difficult to achieve a balance between exploration and exploitation [3]. Secondly, the problem with traditional multi-armed bandit machine models is that they assume that the profit probability of each bandit machine is fixed. However, in the fuzzing process, as the exploration path increases, the probability of discovering new paths decreases. This makes it difficult to

choose the optimal strategy by setting a fixed reward value for the path. In fuzzing, there is a mutual influence between the benefits of different seeds. However, the multi-armed bandit machine model ignores the interaction between different bandit machines, making it difficult to accurately model fuzzing seed scheduling. To address the aforementioned issue, this paper constructs program execution information in fuzz testing into context vectors and designs seed scheduling strategies based on contextual multi-armed bandit models.

C. LinUCB ALGORITHM

Due to the unknown reward for each action in the initial state, solving the MAB problem requires two stages. The exploration stage refers to attempting to pull more rods, which may not necessarily result in the maximum reward. However, this approach allows for the evaluation of the reward situation for all rods. The exploration stage refers to pulling the rod with the expectation of the maximum reward that is already known. Since the available information is based on a limited number of interactive observations, the current optimal rod may not necessarily be globally optimal. Therefore, it is necessary to design a strategy to balance the number of explorations and exploitation, in order to maximize cumulative rewards [16].

The LinUCB algorithm is an online learning algorithm originally proposed by Li, a researcher at Yahoo, in 2010 [17]. The LinUCB algorithm combines linear regression models with the Upper Confidence Bound strategy to dynamically estimate the expected benefits of each arm in a continuous, high-dimensional feature space. It intelligently explores based on confidence intervals, effectively solving the contextual multi-armed bandit machine problem. The LinUCB algorithm combines the concepts of linear prediction models and confidence intervals. It establishes a linear prediction model using feature vectors to estimate the expected reward of the action arm and selects the action with the maximum expected reward. As each action is taken and reward feedback is received, the algorithm updates the linear model based on the observational results, continuously optimizing its predictive ability for future action rewards. The LinUCB algorithm combines linear models and UCB strategies to effectively balance the trade-off between exploring new information and exploiting existing knowledge when dealing with decision tasks involving continuous and high-dimensional feature spaces. It aims to maximize the cumulative reward within a limited number of interactions [18].

The LinUCB algorithm defines a coefficient vector $\boldsymbol{\theta}_a^*$ for each action a in action set A of the multi arm bandit machine model, and sets the expected reward of each arm as a linear function of its feature vector

$$E\left[r_{t,a} \mid x_{t,a}\right] = \mathbf{x}_{\mathbf{t,a}}^{\mathbf{T}} \theta_{\mathbf{a}}^* \tag{1}$$

Based on historical data and returns, approximate estimates $\hat{\boldsymbol{\theta}}$ can be made. Defined $\mathbf{D_a}$ as the m $*$ d dimensional feature matrix at time t, which corresponds to the feature vectors of m historical inputs and the reward vectors of m inputs, ridge regression was applied to the historical data ($\mathbf{D_a}$, $\mathbf{c_a}$) to estimate the coefficients:

$$\widehat{\theta}_a = \left(\mathbf{D}_a^{\top}\mathbf{D}_a + \mathbf{I}_d\right)^{-1}\mathbf{D}_a^{\top}\mathbf{c}_a \tag{2}$$

Using substitution, its upper confidence bound as shown in equation (3)

$$P\left\{\left|\mathbf{x}_{t,a}^{\top}\widehat{\theta}_a - \mathbf{E}\left[\mathbf{r_{t,a}} \mid \mathbf{x_{t,a}}\right]\right| \le \alpha\sqrt{\mathbf{x^T}\left(\mathbf{D_a^{\top}D_a}+\mathbf{I_d}\right)^{-1}\mathbf{x}}\right\} \le 1-\delta \tag{3}$$

$$\text{where: } \alpha = 1 + \sqrt{\ln(2/\delta)/2} \tag{4}$$

α is defined as explore factors. The larger α, the wider the confidence interval, which is more inclined towards exploration; On the contrary, the smaller α, the more inclined it is to utilize.

At this point, the optimal action selected based on contextual historical information is:

$$a_t \overset{\text{def}}{=} \underset{a\in\mathcal{A}_t}{\arg\max}\left(\mathbf{x}_{t,a}^{\top}\widehat{\theta}_a + \alpha\sqrt{\mathbf{x}_{t,a}^{\top}\mathbf{A}_a^{-1}\mathbf{x}_{t,a}}\right) \tag{5}$$

$$\text{where: } \mathbf{A}_a \overset{\text{def}}{=} \mathbf{D}_a^{\top}\mathbf{D}_a + \mathbf{I}_d \tag{6}$$

Through the LinUCB algorithm, the system utilized contextual information to fully exploit and explored each action to be selected. This helped discover truly valuable actions, gradually eliminate less than ideal actions, and achieve maximum revenue.

III. METHODS AND IMPLEMENTATION

A. OVERVIEW

LinFuzz is derived from AFL 2.52b and adheres to the AFL framework, incorporating its key mechanisms such as feedback-driven coverage and crash filtering mechanisms. Based on the aforementioned considerations, we have incorporated the LinUCB seed scheduling algorithm and the path-sensitive energy allocation algorithm into the seed scheduling module in order to enhance the effectiveness of the seed scheduling strategy.

Figure 1 illustrates the overall framework of the LinFuzz system. The LinFuzz fuzz testing tool comprises a feature extraction module, a seed scheduling module, and a program execution module. In the process of fuzz testing, the seed file is first added to the seed queue. The feature extraction module removes low-scoring seeds based on the feedback information from fuzz testing, extracts feature values from seed execution, and constructs feature vectors. The seed scheduling module utilizes the LinUCB algorithm to estimate the expected seed return and choose the next round of test seeds. The energy scheduling module utilizes an energy scheduling algorithm based on path heat to allocate energy to the seeds. The program execution module collects program information through instrumentation and tests the seeds. The test case execution information is fed back to the feature extraction module to update the seed features, and new seeds discovered during testing are added to the seed queue.

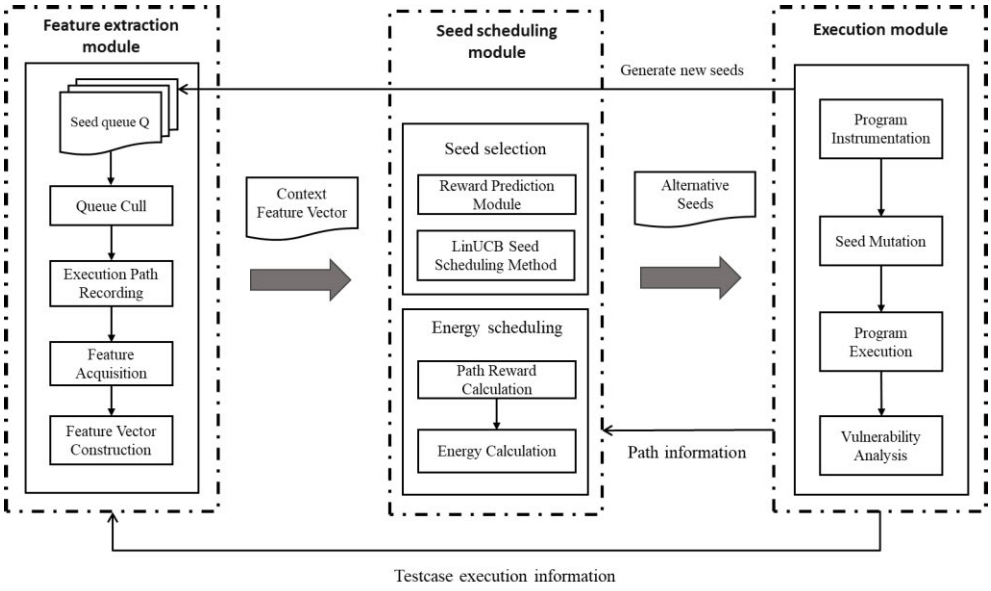

FIGURE 1. The overview of LinFuzz.

The subsequent section of this chapter provides a comprehensive elaboration on the aforementioned perspective. In Section III-B, the context attribute and the LinUCB based seed scheduling algorithm were defined for the purpose of seed selection. In Section III-C, the concept of path value based on frequency and danger was introduced. In Section III-D, the seed energy allocation and queue cull algorithm based on path value were described.

B. SEED SCHEDULING ALGORITHM BASED ON LinUCB

When conducting fuzzing on a specific program, it has been observed that seeds with identical attributes exhibit varying levels of efficiency across different programs. Reference [9] demonstrates, through the application of statistical testing methods, that seeds possessing distinct attributes exhibit varying performance within a single program, while seeds possessing identical attributes exhibit divergent performance across different programs. The current fuzzing tools used for seed evaluation primarily focus on identifying unique paths and crashes, often overlooking the interrelation between various attributes and programs. This oversight may result in the failure to detect certain vulnerabilities in seed discovery during the fuzzing process.

In order to fully utilize the feedback information from each round of fuzz testing, this paper abstractly expresses the seed selection problem as a contextual multi-armed bandit machine problem. Specifically, LinFuzz constructs a feature vector $\mathbf{X}_{t,s}$ for each seed s. Based on the contextual information accumulated during the historical iteration process, it constructs a global seed context matrix $\mathbf{A_t}$, while recording the actual feedback reward p_t brought by each seed. On this basis, LinFuzz utilizes the LinUCB algorithm to predict the upper limit of potential rewards for seeds. Subsequently, it selects the seed with the maximum reward for the next round of fuzz testing. The constructed context feature vector $\mathbf{X}_{t,s}$ is crucial as it ensures the inclusion of attributes closely related to the efficiency of fuzz testing while avoiding unnecessary increases in algorithm complexity due to excessive dimensionality. To achieve this goal, we integrated the actual testing scenario of real programs with the detailed feedback offered by instrumentation technology. We systematically identified several key seed attributes that directly impact the efficiency of fuzz testing and developed the context attribute vector of the seeds. The seed attributes contained in the feature vector are shown in Table 1, covering the basic attributes of the seeds and information during program execution. LinFuzz compiles and executes programs using AFL's LLVM mode to obtain seed attribute information. During the process

TABLE 1. The Properties of seed in context vector.

Seed Properties	Definition
Execution time	Execution time of seed file when testing program
File size	Seed file size
Number of memory operations	The total number of memory access operations during the execution of seed mutation cases, obtained by the LLVM compiler
Number of loop executions	The testcase triggered the number of program cycles during execution
Max Path depth	Seed mutation test cases start from the testing entry point, follow the branching structure of program execution, and experience the number of function call levels or other control flow transitions.
Edge variation	Seed mutation produces execution edge changes

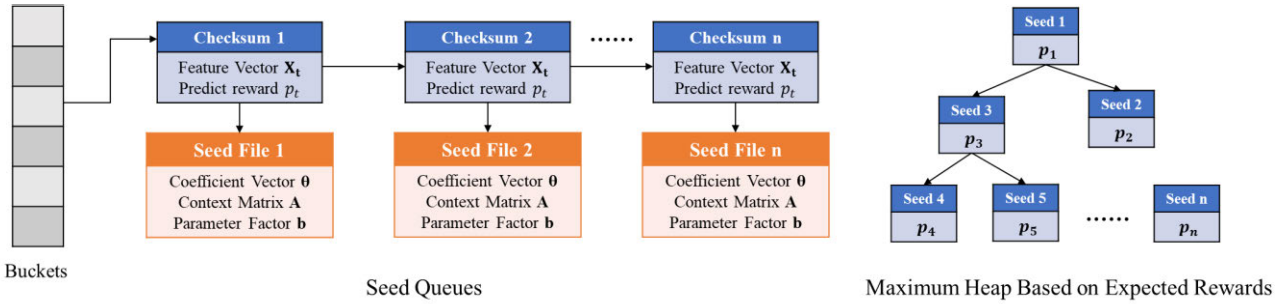

FIGURE 2. The new data structure of the seed queue.

of constructing feature vectors, some attributes need to be standardized. We have adopted a standardized method for label encoding for two types of attributes: execution time and file size. Based on the practical experience of fuzz testing, the execution time is categorized into four groups: long, short, normal, and timeout. Similarly, the file size is classified into three levels: large, small, and medium. Other fundamental attributes are documented according to the highest value of the test samples produced following seed mutation throughout the execution process. After each round of fuzzing, LinFuzz updates the changing attribute information to recalculate the seed reward in the next round of fuzzing testing.

Algorithm 1 describes a seed scheduling approach that leverages the LinUCB algorithm. The algorithm involves updating the seed feature vector using historical testing outcomes prior to commencing each round of fuzz testing, as well as initializing the context feature matrix for newly added seeds. It then proceeds to calculate the expected reward and confidence upper bound of the seed based on historical execution data. Additionally, the algorithm updates the context matrix parameters based on the feedback received during each round of fuzz testing. The seed attributes and reward values are updated before each round of fuzz testing, and the expected reward and confidence upper bound of the seed are computed using the LinUCB algorithm formula. The seed with the highest expected reward is selected for the subsequent round of fuzz testing. Following the fuzz test, the context matrix parameters are updated based on the actual reward obtained, and the process continues to the next round of testing.

To operationalize a seed scheduling algorithm reliant on context vectors, modifications were made to the seed storage data structure to incorporate attributes and files associated with the seeds, as depicted in Figure 2. The seed queue is modeled after the AFL queue storage system, which retains the initial stored data while incorporating the seed context matrix and feature vectors. LinFuzz employs a maximum heap to oversee the seed queue, organizing seed IDs according to the anticipated value p, sequentially selecting seeds for testing, and giving precedence to seeds with greater expected reward values to optimize testing efficacy.

Algorithm 1 Seed Select Algorithm

Input: Seed Queues Q, Feature Vector $\mathbf{x}_t$,
Seeds Context Matrix $\boldsymbol{A_t}$,Seed rewards r_t;
Output: The Seed in Next Fuzzing Loop

for t = 1 to T
  Update Feature vectors of all Seeds in Q
  for all s **in** Q **do**
    If s **is** *newseed* **then**
      $\boldsymbol{A_s} \leftarrow \mathbf{I_d}$ *(d-dimensional identity matrix)*
      $\boldsymbol{b_s} \leftarrow \mathbf{0}_{d\times 1}$ (d-dimensional zero vector)
    End if
    $\widehat{\theta_s} \leftarrow \mathbf{A_s^{-1}b_s}$
    $p_{t,s} \leftarrow \widehat{\theta}_s^T \mathbf{x_{t,s}} + \alpha\sqrt{\mathbf{x_{t,s}^T A_s^{-1} x_{t,s}}}$
  end for
  $k \leftarrow arg\max_{s\in Q} p_{t,s}$
  fuzzing (s_k)
  get_reward $r_{t,s}$
  $\mathbf{A}_{s_k} \leftarrow \mathbf{A}_{s_k} + \mathbf{x_{t,s_k} x_{t,s_k}^T}$
  $\mathbf{b}_{s_k} \leftarrow \mathbf{b}_{s_k} + r_{t,s_k}\mathbf{x}_{t,s_k}$
end for

C. FREQUENCY AND DANGER BASED PATH VALUE EVALUATION ALGORITHM

In the current field of fuzz testing tools, the assessment of edge coverage serves as a crucial evaluation metric and a benchmark for rewards, relying on reinforcement learning fuzzification [19]. At present, coverage-based fuzzing tools do not differentiate the importance of different code coverage segments, irrespective of the functionality of code calls and their potential security impacts. Therefore, all input samples that aid in discovering new statements or code conversions will be saved for future mutations [20]. Although this approach may be deemed acceptable in software testing for achieving thorough code coverage, it is considered ineffective in identifying vulnerabilities. During the fuzzification process, as the number of specific path executions increases, the likelihood of discovering new paths and vulnerabilities decreases. This leads to a gradual decrease in the effectiveness of seed exploration and vulnerability discovery.

In addition, paths in the program that involve dangerous functions such as memory calls are more likely to expose vulnerabilities. Therefore, relying solely on path coverage to evaluate the value of paths may lead to fuzz testing seed scheduling algorithms falling into local optima. To address this particular issue, this study introduces a new method for calculating program path fitness using path heat and path danger.

The danger level of program edges is determined by the danger function of path execution. Usually, vulnerabilities are caused by developers mistakenly using standard library functions that involve memory operations. Table 2 compiles common hazard functions that are susceptible to triggering vulnerabilities like buffer overflow and UAF, and categorizes function hazards according to factors such as the complexity of triggering vulnerabilities. The fuzzer assesses the level of risk at the edge by considering the quantity of risky functions and risk levels present in the basic block of the edge target that is executed by the program.

TABLE 2. Common dangerous functions and risk levels.

Function	Description	Risk
memcpy, memcmp	Commonly used memory operation functions in C language, improper parameter usage can easily cause buffer overflow and memory crossover access.	4
strcpy, strcat	A classic library function used in C language to handle strings, used for string copying and concatenation. If the target string does not have enough space to accommodate the source string, it can easily cause stack overflow.	4
read,fread	C language reading function. If the buffer size is not specified correctly when calling a function, buffer overflow may occur.	2
printf	C language output function. Carefully constructed format strings may cause format string vulnerabilities.	2
malloc, realloc, free	Improper use of memory management functions can easily lead to UAF vulnerabilities.	4
gets	The Gets function does not check whether the read data will exceed the size of the target buffer, which can easily cause buffer overflow.	2

Define V as the set of dangerous functions, Des_e as the target basic block of edge e, and $Func(Des_e)$ as the set of functions called during the execution of basic block Des_e. The danger level of edge e is expressed as:

$$Risk(e) = \begin{cases} Vul(Func\,(Des_e) \cap V) & Func\,(Des_e) \cap V \neq \emptyset \\ 1 & Func\,(Des_e) \cap\ V = \emptyset \end{cases} \quad (7)$$

$Vul\,(\cdot)$ represents the sum of all hazard functions in the set according to the hazard level in the table. LinFuzz calculates path danger by using the LLVM compiler's getCalledFunction() to retrieve the function names in the basic blocks of the program. It then matches these function names with a set of dangerous functions in a table during the compilation of the test program.

The heat of program edges corresponds to the number of times each edge is executed in the fuzzing test. In fuzz testing, LinFuzz constructs a shared edge execution frequency matrix and records the number of times each edge is executed as $N(e)$. The frequency fitness of the edge is related to the heat of the edge, defined as:

$$fre(e) = \begin{cases} 1 & N(e_k) \leq \dfrac{n}{10} \\ \dfrac{n}{10 * N(e)} & N(e_k) > \dfrac{n}{10} \end{cases} \quad (8)$$

where n represents the number of seeds in the current queue.

The frequency fitness calculation method assigns higher fitness to seeds with lower heat. When the edge heat is greater than one-tenth of the number of seeds, we believe that it is difficult to increase the new coverage by continuing to execute the seeds passing through the edge. The frequency fitness of the edge is inversely proportional to the heat of the edge; in other words, the more times the edge is covered, the lower the score. When the heat of the edge is less than one-tenth of the number of seeds, we believe that there is still a possibility of discovering new edges after the seeds pass through the edge. The frequency fitness is assigned as 1, and the higher fitness guides the next fuzz testing process to continue selecting this path.

Based on the *Risk (e)* and frequency fitness *Fre (e)* of edges, define the exploratory value of edges:

$$r\,(e) = Risk\,(e) * fre(e) \quad (9)$$

To calculate path fitness, LinFuzz captures the number of program branch executions through instrumentation detection and manages them using shared memory. In order to record the frequency of path execution and exploration value, LinFuzz enhanced the AFL bitmap structure by incorporating the path frequency matrix and exploration value matrix. These matrices record the number of edge executions and exploration values, with each index corresponding one-to-one with a bitmap index. The method for updating the value of path exploration is illustrated in the Figure 3. After each seed is executed, Linfuzz updates the bitmap based on the execution path, records the newly discovered path, and increases

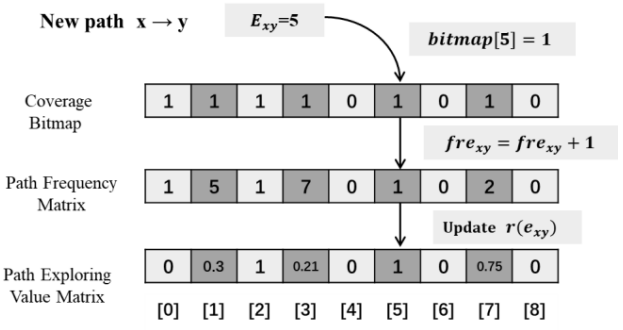

FIGURE 3. Path value update process.

the frequency value of the corresponding seed's execution path by 1. Finally, the exploration value matrix updates the path exploration value based on changes in path frequency.

D. PATH VALUE UPDATE AND ENERGY ALLOCATION ALGORITHM

The reward value of a seed is determined by the exploration value of the path it covers. Improving the calculation method of seed reward value serves a dual purpose: firstly, it offers seed reward feedback for seed scheduling algorithms; secondly, LinFuzz allocates seed energy and removes seed queue elements based on the fitness of the seed execution path. In the LinUCB algorithm, the initial feedback value is determined by the exploration value of the edges it covers and whether the seed triggers vulnerabilities. The seed reward value is the sum of the exploration values of the edges in the seed execution path. If the seed triggers a program vulnerability, it will add additional value. Set the seed s_k to cover the edge set of the program, E_k, and the reward value of the seed:

$$r(s_k) = \sum_{e_i \in E_k} r(e_i) + 5 * Is\_crashed \tag{10}$$

where *Is_crashed* indicates whether the test case of seed mutation triggers a vulnerability. When the execution process triggers a vulnerability, the value of *Is_crashed* is set to 1. When the seed does not trigger a vulnerability during the testing process, the value of *Is_crashed* is 0. The seed reward value gives higher importance to seeds with longer execution paths, which is advantageous for exploring deeper code blocks. Meanwhile, as the value of edge exploration decreases with the increase of edge heat, the seed value of executing the same path multiple times will decrease. This helps avoid limiting path exploration to certain branches and achieves a balance between exploration and utilization.

The energy scheduling strategy also affects the efficiency of fuzz testing. AFL calculates the seed score by considering various factors, including the execution time of the testcases, the size of the bitmap, and the average size of the seed file. Based on this score, seed energy is assigned to the seed. However, this approach fails to account for the varying significance of paths, thereby overlooking the influence of testcases execution paths on vulnerability detection. Consequently, it becomes challenging to effectively leverage the seed's capacity for path exploration. This paper presents a novel approach for energy allocation that takes into account the sensitivity of the path. In order to allocate energy to seeds, the average value of the covered path of each testcase is computed. Subsequently, the energy adjustment factor is determined based on the average path value, enabling the secondary distribution of seed energy.

In each round of fuzzing, calculate the average path value for the selected seed:

$$\overline{r}(s_k) = \frac{\sum_{e_i \in E_k} r(e_i)}{length(E_k)} \tag{11}$$

Set an energy adjustment factor based on the average path value and improve the source AFL program to allocate energy to seeds.

$$factor = 2^{2(\overline{r}(s_k)-0.5)} \tag{12}$$

$$Seed.energy = \max\{Seed.energy * rate, M\} \tag{13}$$

The secondary allocation of seed energy is accomplished by employing energy adjustment factors. Seeds that follow low-frequency paths are allocated more energy compared to seeds that follow high-frequency paths. Algorithm 2 outlines the procedural steps of this method. When the average path value range of the seed falls between 0.125 and 0.5, the range of the energy adjustment factor is between 0.6 and 1. For seeds that follow high-frequency paths, it is necessary to appropriately decrease the energy. When the average path value range of the seed falls between 0.5 and 1, the range of the energy adjustment factor is between 1 and 2. Exploration is fostered through the augmentation of the seed's energy. Finally, in order to maintain optimal efficiency, the maximum energy value of an individual seed is constrained to a value denoted as M, thereby preventing any excessive energy from negatively impacting the overall efficiency.

The aim of queue cull is to simplify the input while ensuring equal edge coverage. In coverage-guided fuzz testing tools, fuzzers typically add test cases that increase coverage to the test queue. However, many retained test cases cover the same edges, leading to redundant testing. Meanwhile, a large number of seeds are retained in the seed queue, which reduces the efficiency of the LinUCB algorithm. The queue cull algorithm of AFL first marks all edges as uncovered. Next, AFL iteratively selects an uncovered edge, chooses the input that covers the edge with the smallest size and execution delay, and marks all edges accessed by the input as covered. AFL repeats this process until all edges are marked as covered.

LinFuzz enhances the queue cull algorithm for AFL. As described in Algorithm 2, LinFuzz Seed maintains a temporary coverage bitmap *Temp_map* and marks all edges as uncovered during initialization. Subsequently, LinFuzz iteratively selects the seed based on its reward value, marks it as "favor" and assigns a value of 1 to the edge covered by the seed in the corresponding position of *Temp_map*. If it is found that the current *Temp_map* edge set contains a seed-covering edge set, meaning the seed has not changed the *Temp_map*, it is marked as "no favor." Due to the correlation between the reward value and the number and depth of seed coverage paths, this algorithm can prioritize retaining seeds with high coverage and mutation potential while removing duplicate coverage seeds.

IV. EXPERIMENTS

In our evaluation, we aim to answer the following research questions:

Research Question 1:Can the seed scheduling method based on LinUCB algorithm improve the efficiency of fuzzing?

Algorithm 2 Queue Cull

Input: Reward value based seed priority queue Q

```
for s = Q.head → Q.end do
    s.favor ← 0
end for
temp_map ← ∅
for s = Q.head → Q.end do
    s.favor ← 1
    if s.coveredge ∈ temp_map then
        s.favor ← 0
        continue
    end if
    temp_map ← temp_map ∪ s.coveredge
end for
```

TABLE 3. Subjects for Crash Reproduction in experiment A.

Subjects	Version	CVE-ID	Type of Vulnerability
Xpdf	3.02	CVE-2019-13288	infinite recursion
Libexif	0.6.14	CVE-2009-3895	heap buffer overflow
		CVE-2012-2836	Out-of-bounds Read
TCPdump	4.9.2	CVE-2017-13028	Out-of-bounds Read
Libtiff	4.0.4	CVE-2016-9297	Out-of-bounds Read
libpng	1.5.4	CVE-2011-3328	Division by Zero
Libxml2	2.9.4	CVE-2015-7497	Heap buffer overflow
		CVE-2017-9048	Stack buffer overflow

Research Question 2:Are we proposed method's performance and efficiency better than those of the comparison targets in various programs?

Research Question 3:How does our proposed method affect the efficiency of fuzz testing?

We design three sets of experiments to answer the above questions separately.We conduct all the following experiments on a machine running 64-bit Debian 11.3.0 equipped with one Intel i7-10750H @3.60GHz and 8GB RAM. Each fuzzing instance is bound to a core to avoid interference.

A. SEED SCHEDULING METHOD EFFECTIVENESS

In this experiment, we evaluated the effectiveness of our proposed LinUCB algorithm in the seed scheduling process. We integrate the seed scheduling method based on the LinUCB algorithm into AFL. In order to conduct a comprehensive comparative analysis, we selected the simple heuristic scheduling algorithm built into the original AFL, as well as two widely recognized optimization algorithms based on multi-armed bandit machine models: the Epsilon-Greedy algorithm and the UCB algorithm, as control benchmarks. The experimental subjects are a group of programs containing six specific vulnerability features, and the specific types of these vulnerabilities are detailed in Table 3. In order to rigorously test the effectiveness of the seed scheduling algorithm, we standardized the instrumentation and compilation of all test programs in the same experimental environment. Especially in the section involving Libexif library functions, we have specifically developed corresponding C language call interfaces to ensure compatibility and accuracy. We consistently utilize the same seed file as the source of initialization test data and meticulously set up the seed file format dictionary in advance for XML special format input scenarios to standardize and direct the accurate generation and parsing of seed files.

Based on the experiment presented in reference [21], the efficiency and performance improvement of fuzzing can be evaluated by quantifying the exposure time. TTE (Time to Exposure) refers to the duration between the initial discovery of a specific vulnerability and its identification during the process of fuzzing. The measurement of factor improvement involves calculating the performance gain by dividing the average time to event (TTE) of the AFL program by the TTE of the improved program. A higher factor suggests that the method is superior. For each program, we conducted five iterations and calculated the average of the experimental results. Table 4 presents the findings obtained from the conducted experiment.

Using different scheduling algorithms to test specified vulnerabilities in the same environment, the TTE is shown in Table 4. The table indicates that, in most cases, the fuzz testing tool employing the enhanced seed scheduling algorithm requires less time to exploit the target vulnerability. However, in CVE-2009-3895 reappearance, the scheduling algorithm is less efficient than the original AFL algorithm. Due to the time required for seed fitness calculation in scheduling algorithms, and the fact that this vulnerability is relatively easy to trigger, the discovery of this vulnerability takes more time compared to AFL. Compared to other scheduling algorithms, the LinUCB algorithm has higher efficiency in discovering vulnerabilities, showing an efficiency improvement of over 10%. Due to the fact that the LinUCB algorithm places more emphasis on the influence of program execution information and seed properties on testing, while Epsilon-Greedy and UCB algorithms only depend on historical rewards and seed coverage for scheduling, their effectiveness in triggering specific vulnerabilities is lower compared to the LinUCB algorithm. The experiment shows that the LinUCB based seed scheduling strategy adopted by Linfuzz is effective, especially for complex vulnerabilities that may take a long time to trigger. The vulnerability discovery efficiency has been significantly improved.

The experiment shows that Linfuzz's seed scheduling strategy is effective and has a significant effect in testing large-scale software, and can play an effective role in vulnerability replication.

TABLE 4. Performance of seed schedule algorithm in experiment A.

CVE-ID	AFL	AFL-Epsilon-Greedy		AFL-UCB		AFL-LinUCB	
	TTE	TTE	factor	TTE	factor	TTE	factor
CVE-2009-3895	6m15s	5m48s	1.07	7m10s	0.87	7m19s	0.85
CVE-2012-2836	9m37s	10m22s	0.92	8m56s	1.07	8m12s	1.17
CVE-2019-13288	18m21s	10m10s	1.80	11m09s	1.65	11m57s	1.53
CVE-2016-9297	32m08s	22m45s	1.41	24m13s	1.33	20m16s	1.59
CVE-2011-3328	3h15m	2h32m	1.28	2h08m	1.52	1h12m	2.71
CVE-2017-9048	9h22m	8h26m	1.11	8h34m	1.09	7h48m	1.2
CVE-2015-7497	15h47m	14h13m	1.11	13h58m	1.13	12h43m	1.24
CVE-2017-13028	16h23m	14h33m	1.13	12h55m	1.27	11h54m	1.38

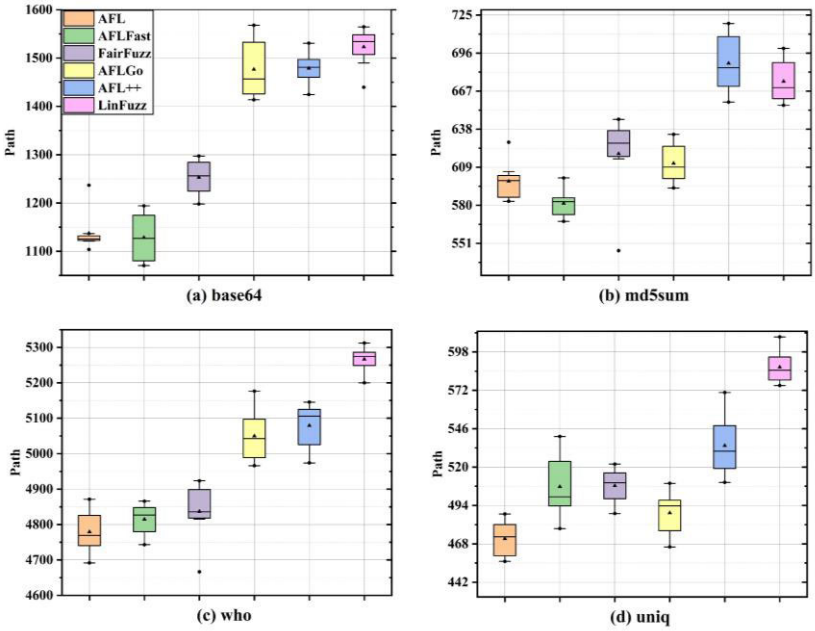

FIGURE 4. The boxplot of region coverage found in 8 trials on LAVA-M, where '▲' and '—' represent the mean and median, respectively.

Answer to RQ 1. The proposed seed schedual algorithm based on LinUCB can improved crash detection efficiency over comparison targets by appropriately evaluating the crash detection potential of the seed and preferential selecting suitable seeds in fuzz testing.

B. EVALUATION ON LAVA-M

In order to verify the efficiency of LinFuzz vulnerability discovery, we conducted comparative experiments on the LAVA-M dataset with other coverage oriented greybox fuzzing tools. The LAVA-M dataset is proposed as a benchmark for evaluating fuzzers performance. The dataset contains four programs: base64, md5sum, uniq, and who [22]. Each program is generated by injecting some bugs into the source code. The triggering conditions of the vulnerability add certain path constraints. Testing the LAVA-M dataset can verify the effectiveness of the greybox fuzzing tool.

The experiment selected open-source fuzzers AFL [12], AFLFast [7], AFLGo [21], FairFuzz [23] and AFL++ [24] for comparison. For each fuzzer, use the same seed file to test for 24 hours. Conduct 8 experiments using different seeds and take the average of the results.

Each fuzzer conducted 8 experiments, and their path coverage performance is shown in Figure 4. From the upper quartile and median of the box plot, it can be observed that LinFuzz performs the best in achieving path coverage for the

TABLE 5. Number of paths covered and bugs found by each fuzzer on LAVA-M.

Project	Number of paths covered on average							Number of bugs found					
	AFL	AFLFast	FairFuzz	AFLGo	AFL++	LinFuzz	High than AFL	AFL	AFL-Fast	FairFuzz	AFLGo	AFL++	Lin-Fuzz
base64	1137	1128	1253	1476	1490	**1523**	0.33	4	7	5	12	**15**	**15**
md5sum	598	581	619	611	**688**	675	0.12	6	6	8	16	**25**	21
who	4779	4814	4836	5050	5079	**5266**	0.1	12	25	16	13	41	**62**
uniq	471	507	507	488	534	**588**	0.24	5	7	6	8	16	**19**
total	6985	7030	7215	7625	7791	**8052**	0.15	27	45	35	49	97	**117**

base64, who, and uniq programs. Table 5 indicates that in the four programs of the LAVA-M test set, LinFuzz has a 15% increase in total path coverage compared to AFL. However, as shown in Figure 4, LinFuzz does not always achieve the best results in every test. AFL++ performs better in the md5sum program test, with a 2% increase in path coverage compared to LinFuzz. Although LinFuzz changes the order of seed testing and the allocation of mutation energy, it does not modify the mutation method. Therefore, the initial seed selection has a significant impact on the experimental results of LinFuzz during program testing. However, LinFuzz has the highest total path discovery count among the four test programs. From this, we conclude that our method outperforms other open-source tools in terms of average path coverage on the LAVA-M dataset.

Table 5 also shows the vulnerability discovery situation of the fuzz testing tool in LAVA-M. As depicted in Table 5, LinFuzz detected a higher number of bugs in base64, who, and uniq compared to other fuzzers. In md5sum, AFL++ achieved more path coverage, resulting in a higher number of vulnerability discoveries than LinFuzz. However, overall, LinFuzz discovered the most bugs in the LAVA-M dataset. Therefore, it can be concluded that LinFuzz outperforms other fuzzers in the LAVA-M dataset.

C. EVALUATION ON REAL-WORLD PROGRAMS

In this segment of our experimentation, we assessed the performance of LinFuzz by employing it on real-world software programs. To identify appropriate exemplars that effectively gauge the efficacy of fuzz testing, we undertook extensive research into scholarly articles published in esteemed journals within the realms of information security and software engineering. This meticulous process culminated in the identification of a collection of representative real-world applications, presented in Table 6.

These 12 meticulously chosen programs serve as benchmarks for assessing both the effectiveness and efficiency of fuzz testing methodologies. They encompass a broad range of functionalities, spanning six distinct categories: image processing, audio parsing, video manipulation, compressed file handling, binary file processing, among others. Notably, these programs exhibit not just functional diversity but also typify a variety of security vulnerabilities. These include prevalent security issues such as heap buffer overflow, stack overflow, segmentation faults, global buffer overflows, different variations of stack buffer overflows, memory leaks, use-after-free errors, floating-point operation exceptions, and more.

TABLE 6. The configuration of target programs.

Subjects	Version	Format
nm -C @@	Binutils-2.32	elf
objdump -d @@	Binutils-2.32	elf
readelf -a @@	Binutils-2.32	elf
size @@	Binutils-2.32	elf
C++filt @@	Binutils-2.32	elf
djpeg @@	libjpeg-turbo-1.5.3	jpeg
readpng @@	libpng-1.6.37	png
jhead @@	jhead-3.03	jpeg
bsdtar -xf @@ /dev/null	libarchive-3.4.0	tar
exiv2 @@/dev/null	exiv2-0.26	jpeg
mp3gain @@	mp3gain-1.5.2	mp3
mp42aac @@/dev/null	Bento4-1.5.1-628	mp4

For comparative analysis, we selected four leading fuzzing tools—AFL, AFLFast, FairFuzz, and Neuzz. The Neuzz [25], specifically, is an enhanced coverage-guided fuzzing tool built upon AFL, integrating neural networks to refine program exploration and thereby enhancing the efficiency and performance of fuzz testing. Each test program underwent a rigorous 24-hour fuzz testing experiment, with random seed selection drawn from a seed corpus for every trial. These experiments were executed within a single-core processor environment, designed to emulate the performance of fuzz testing tools under identical resource constraints. Moreover, each experiment was repeated five times to mitigate the effects of randomness and ensure robust, reliable results.

Seed scheduling aims to discover additional paths and vulnerabilities with the minimum number of execution cases.

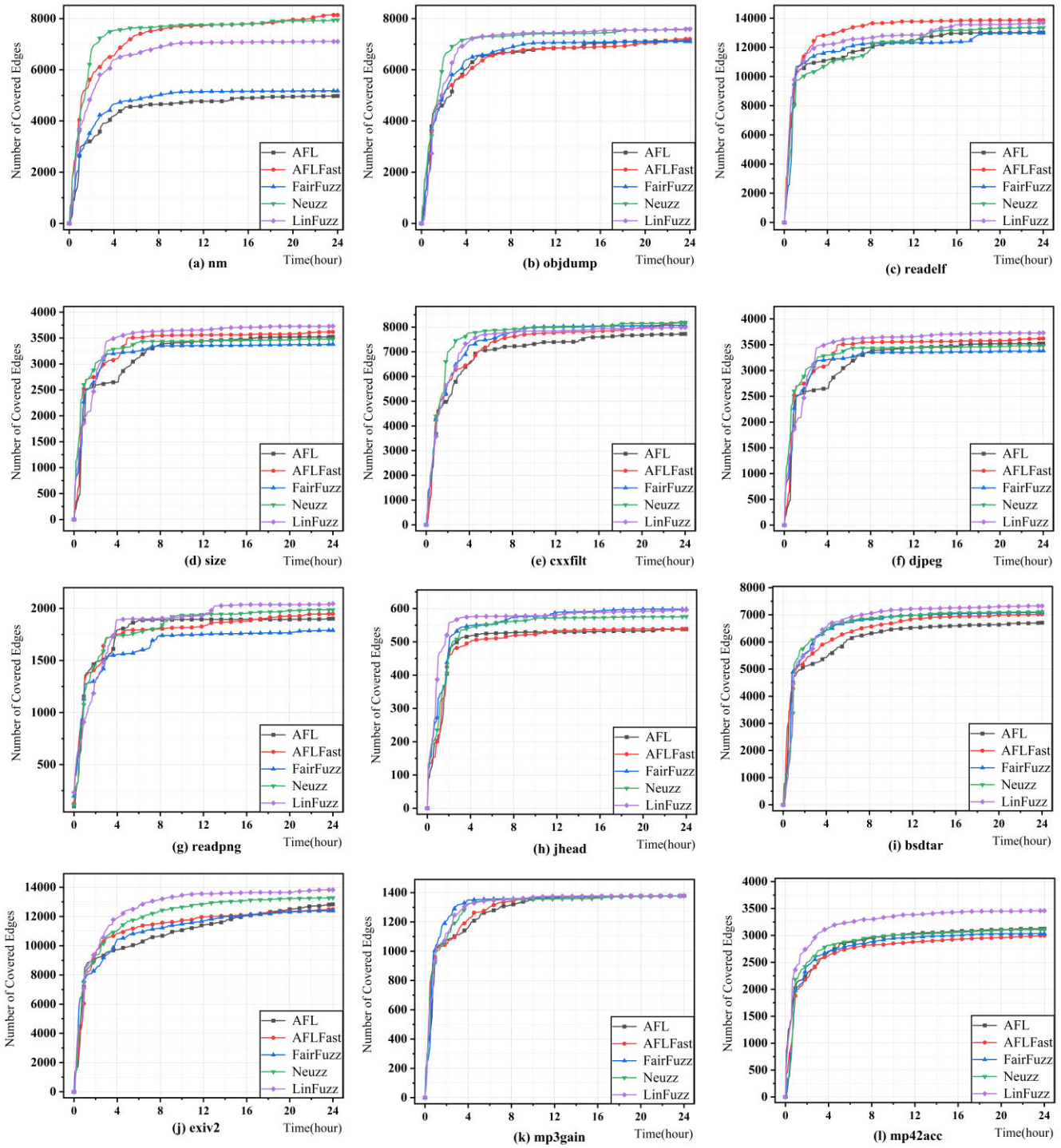

FIGURE 5. Number of total paths discovered by different fuzzers.

Therefore, we select the number of path coverage and vulnerability discovery as the evaluation criteria for fuzz testing within the same time frame. Figure 5 illustrates the variation of 24-hour path coverage, while Table 7 displays the total number of path coverage and the average number of vulnerabilities discovered by the fuzzers in 5 experiments.

The total number of path coverage for each fuzz testing tool during the execution of the test program is shown in Table 7. The data in Table 7 shows that in the coverage evaluation of 12 target programs, LinFuzz achieved higher coverage indicators in 8 of them and surpassed other comparison tools in overall path coverage statistics. Specifically, compared to

TABLE 7. Number of paths covered and bugs found by each fuzzer on real-world programs.

Subjects	Number of total paths					Number of bugs				
	AFL	AFLFast	FairFuzz	Neuzz	LinFuzz	AFL	AFLFast	FairFuzz	Neuzz	LinFuzz
nm	4981	8139	5178	7948	**8268**	7	11	10	9	**13**
objdump	7156	7203	7104	**7590**	7586	5	14	**26**	20	**26**
readelf	13017	**13864**	13005	13345	13683	1	1	1	1	1
size	3401	3522	3234	3748	**3989**	0	1	1	1	1
cxxfilt	7723	8180	8068	**8185**	7983	2	4	5	**6**	4
djpeg	3524	3619	3384	3478	**3729**	2	4	4	**8**	6
readpng	1902	1948	1789	1987	**2045**	3	5	3	5	**7**
jhead	538	539	**598**	576	596	5	5	9	8	**16**
bsdtar	6707	7021	7103	7101	**7326**	0	0	0	0	0
exiv2	12834	12493	12389	13274	**13825**	6	10	11	10	**13**
mp3gain	**1379**	1377	1378	1377	**1379**	60	65	61	66	**67**
mp42aac	3124	2998	3033	3110	**3458**	22	26	26	**28**	24
Total	66286 (-11.44%)	70903 (-4.17%)	66263 (-11.48%)	71719 (-3.00%)	**73867**	113 (-57.52%)	146 (-21.91%)	157 (-13.37%)	162 (-9.87%)	**178**

AFL and FairFuzz, LinFuzz achieved a significant improvement, increasing path coverage by 11.44% and 11.48%, respectively. For Neuzz, LinFuzz achieved a path coverage increment of 3.00%. Based on the above data analysis, we can conclude that the scheduling algorithm and energy allocation strategy adopted by LinFuzz are effective in improving code coverage.

Figure 5 illustrates the evolution trend of path coverage over time for the test program using various fuzz testing tools. The results indicate that in most cases, the LinFuzz path coverage rate is generally higher than that of other comparative testing tools. Although LinFuzz's path discovery rate was slightly slower than AFLFast in the testing of the readelf program, it still maintains a leading position in overall performance on other programs. Due to LinFuzz's improved seed scheduling and seed energy scheduling algorithms, seeds with higher mutation potential are prioritized for execution. This reduces the energy required to execute high-frequency path seeds, guiding testing tools to discover paths faster. On average, LinFuzz has the highest path discovery rate among the five fuzzing testing tools.

Table 7 shows that LinFuzz exhibits significant advantages over comparative testing tools in terms of the total number of vulnerabilities discovered. Among the six tested programs, LinFuzz discovered the highest number of vulnerabilities. For example, in the nm program, LinFuzz can discover two additional unique vulnerabilities compared to AFLFast. The Neuzz tool enhanced seed scheduling using neural networks and determined that it outperformed AFL, AFLFast, and FairFuzz in identifying vulnerabilities. The best vulnerability discovery results were achieved in the cxxfilt and djpeg program tests. However, overall, LinFuzz found the highest number of independent vulnerabilities. Further precise analysis shows that the number of unique vulnerabilities revealed by LinFuzz has increased by 57.52%, 21.91%, 13.37%, and 9.87% compared to AFL, AFLFast, FairFuzz, and Neuzz, respectively. This result clearly demonstrates the superiority and deep mining ability of LinFuzz in vulnerability discovery performance.

In order to further verify the exact effectiveness of the vulnerabilities detected by LinFuzz, we documented the known public vulnerabilities and their corresponding generic vulnerability disclosure CVE identifiers identified by various fuzz testing tools in the target application during the experimental process. These data are summarized in Table 8. The results showed that LinFuzz performed particularly well in identifying CVEs, successfully discovering a total of 22 CVE IDs, which is 3 more than the second-ranked Neuzz tool. These CVE IDs contain denial of service and stack overflow vulnerabilities, posing serious threats to businesses and users. Compared with the original AFL tool, LinFuzz identified a total of 5 additional CVE IDs across all testing objectives, demonstrating the effectiveness of our method in enhancing vulnerability detection efficiency. The significant performance of the discovered and published CVE IDs demonstrates the effectiveness and efficiency of LinFuzz in detecting serious vulnerabilities.

TABLE 8. The published CVE IDs found by each fuzzer.

Subjects	CVE ID	AFL	AFLFast	FairFuzz	Neuzz	LinFuzz
nm	CVE-2019-17451	●	●	●	●	●
	CVE-2019-9070	●	●	●	●	●
objdump	CVE-2018-1000876	●	●	●	●	●
	CVE-2021-3487			●		●
	CVE-2019-17450				●	●
	CVE-2019-9072	●	●	●	●	●
readelf	CVE-2019-9077	●	●	●	●	●
cxxfilt	CVE-2019-9071			●	●	●
	CVE-2019-9070		●	●	●	●
jhead	CVE-2020-26208					●
	CVE-2020-6625	●	●	●	●	●
	CVE-2020-6624	●	●	●	●	●
	CVE-2019-19035	●	●		●	●
exiv2	CVE-2018-4868	●	●		●	●
	CVE-2019-13109			●	●	●
	CVE-2019-13110			●	●	●
mp3gain	CVE-2017-14406	●	●	●	●	●
	CVE-2018-10777				●	●
	CVE-2018-10783	●	●	●		●
	CVE-2018-10784	●	●	●		●
mp42aac	CVE-2018-10791		●	●	●	●
	CVE-2018-10790		●	●	●	●
	CVE-2018-10785	●			●	
	Total	13	15	17	19	22

Answer to RQ 2. LinFuzz is effective and efficient in coverage improvement and vulnerability discovery. LinFuzz significantly outperforms the other test fuzzers in terms of vulnerabilities detection. As for coverage, LinFuzz achieves more average edge coverage on real programs and LAVA-M dataset.

D. FUZZ OVERHEAD EVALUATION

LinFuzz collects more seed execution information at runtime and constructs a feature matrix to calculate rewards, requiring more steps in seed scheduling than AFL and AFLFast. In addition, LinFuzz considers multiple seed attributes for selecting seeds, but it does not prioritize seed size as the main factor in seed selection. In fuzz testing, larger and more complex seeds may be selected, leading to longer execution times. Therefore, we anticipate that LinFuzz incur additional overhead in fuzz testing, as its fuzz throughput is lower than other testing tools such as AFL.

To assess the extra resource costs associated with LinFuzz in improving fuzz testing performance, we systematically recorded the average and peak memory usage of different fuzz testing tools in a set of experiments aimed at real-world programs. The results are summarized in Table 9. Based on the data presented in Table 9, we have made the following substantive observations. Overall, the classic fuzz testing tools AFL and AFLFast perform excellently in terms of memory resource utilization efficiency, with an average memory consumption of only 13.3MB and 14.4MB, respectively. In contrast, during the process of enhancing fuzziness testing, LinFuzz had an average memory usage of 27.5MB, which is approximately double the average consumption of AFL and slightly surpassed FairFuzz's average memory usage. In comparison with another tool, Neuzz, LinFuzz demonstrated relatively better memory usage efficiency, with significantly lower memory consumption than Neuzz. Due to LinFuzz's lightweight scheduling algorithm, it performs better in terms of memory usage compared to Neuzz.

In order to accurately measure the impact of LinFuzz on the throughput of fuzz testing, we focus on examining the time ratio consumed by LinFuzz during the scheduling phase. This is specifically manifested as the proportion of non-direct program execution time, which includes seed feature extraction, running the LinUCB algorithm, seed scheduling algorithm, and energy scheduling time. We conducted a control experiment and selected three other representative fuzz testing tools for reference: Mopt [26], which utilizes the particle swarm optimization algorithm to optimize the seed mutation strategy; Neuzz, which employs neural networks to enhance the efficiency and performance of fuzz testing by smoothing programs; and Angora [27], which utilizes data flow tracking technology to guide its fuzz testing process. These three testing tools enhance the efficiency of fuzz testing through various algorithms but also decrease the throughput of fuzz testing. In the experimental design, we selected a diverse set of program samples and varied seed inputs for each fuzz testing tool to ensure the universality and accuracy of the test results. Each tool conducted a single fuzz test lasting 3 hours for the selected program and seed combination, and a total of 20 independent experiments were carried out. Throughout the entire experiment, we recorded the proportion of non-program execution time relative to the total fuzz testing time. This was done to gain a deeper understanding and evaluate the time cost of each fuzz testing tool in enhancing fuzz testing strategies during actual operation.

In Figure 6, we systematically calculated the proportion of time each testing tool spent on non-program execution tasks compared to their total fuzz testing time during the experiment. The results show that the average time spent by LinFuzz on seed scheduling in fuzz testing accounts for approximately 1.5% of the total time spent in fuzz testing. This percentage is lower compared to the time spent by Neuzz and Angora tools, demonstrating the superiority of LinFuzz's scheduling algorithm in terms of time efficiency. Mopt has the lowest non-program execution time among the four tools, which is about 0.02 percentage points lower than LinFuzz. Due to Mopt not requiring complex feature calculations and matrix operations during runtime, the time cost is relatively low. However, overall, although LinFuzz consumes a certain amount of time resources in the scheduling process,

TABLE 9. The memory consumption (MB) of each fuzzer.

Subject	AFL		AFLFast		FairFuzz		NeFuzz		LinFuzz	
	ave	max	ave	max	ave	max	ave	max	ave	max
nm	8.4	34.2	8.9	26.7	35.5	50.4	47.1	64.2	33.8	47.3
objdump	15.3	150.9	15.4	91.9	23.2	144.8	26.4	177.7	22.6	164.2
readelf	11.0	25.9	12.8	25.4	14.5	69.8	33.3	81.6	15.1	40.7
size	13.3	16.6	15.2	18.8	41.4	92.3	60.9	147.5	51.3	150.2
cxxfilt	14.5	23.2	15.8	40.3	24.2	27.7	28.8	41.2	15.8	30.7
djpeg	18.8	47.4	17.9	55.2	19.8	90.5	43.0	124.4	24.3	156.5
readpng	16.8	213.4	19.4	223.3	49.4	240.2	28.5	461.0	24.7	293.1
jhead	7.3	12.5	7.2	20.1	13.6	28.3	60.9	77.8	18.3	27.3
bsdtar	23.2	44.5	25.5	47.8	26.5	84.6	29.8	93.0	31.1	106.1
exiv2	11.7	27.9	13.4	37.7	21.8	44.5	49.0	124.3	30.5	66.7
mp3gain	9.5	12.5	12.4	16.4	34.7	46.3	64.1	88.0	35.2	47.1
mp42aac	9.9	27.2	9.3	24.3	18.4	176.4	58.3	110.2	27.4	61.6
average	13.3	53.0	14.4	52.3	26.9	91.3	44.2	132.6	27.5	99.3

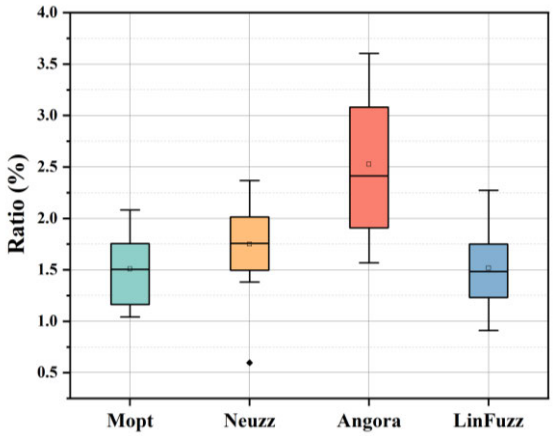

FIGURE 6. The boxplot of fuzzers' seed scheduling time ratio, where '■' and '—' represent the mean and median, respectively.

the scheduling algorithm introduced by LinFuzz has a lesser impact on the overall efficiency of fuzz testing. The increased cost compared to tools that depend on complex optimization methods like static analysis and neural networks seems to be more moderate.

Answer to RQ 3. Although LinFuzz incurs additional time overhead while optimizing fuzz testing performance, this increase has a relatively low impact on fuzz testing efficiency.

V. DISSCUSSION AND LIMITATION

This paper proposes a LinUCB seed scheduling algorithm for enhancing fuzz testing efficiency. The algorithm aims to accelerate path coverage and vulnerability discovery. However, the LinUCB algorithm utilizes matrix operations to calculate expected rewards, which may reduce program execution efficiency. However, based on the LAVA-M dataset and real program testing, it was found that LinFuzz can still achieve better results in long-term fuzz testing. In comparison to complex methods like static analysis, LinFuzz consumes fewer resources. In actual program testing, we found that LinFuzz has a higher path coverage rate compared to other fuzz tools.

The method of calculating path value based on path frequency and danger level improves the ability of fuzz testing to explore low-frequency and high-risk paths. It also enhances performance in terms of path coverage and the quantity of vulnerability detection. The path value calculation method proposed in this article relies on dynamic program instrumentation technology and is suitable for greybox fuzz testing. For black box fuzz testing without source code, it is still essential to develop a method for calculating path values based on program execution.

This paper proposes a path-sensitive energy allocation strategy to allocate higher energy to seeds executing low-frequency and high-risk paths. However, the efficiency improvement of fuzz testing through energy allocation is limited. In the future, it will still be necessary to integrate seed execution information and program analysis to develop seed-directed mutation algorithms. This will enable seeds

to explore more paths with the fewest mutations, thereby enhancing the efficiency of fuzz testing.

VI. RELATEDWORK

A. COVERAGE-GUIDED GREYBOX FUZZING

As one of the most widely utilized methodologies in software testing, fuzzing has experienced significant advancements in recent years, particularly in the domain of coverage-guided greybox fuzzing. In the conducted fuzzing campaign [23], [26], [28], [29], [30], [31]. Numerous techniques have been introduced to enhance the effectiveness of fuzzing across various domains since its inception in the 1990s.

Several research studies employ static analysis and symbol execution techniques to enhance the efficacy of fuzz testing. QSYM [32] presents a novel approach that integrates dynamic symbol execution with fuzz testing in order to reduce superfluous program execution branches and enhance the efficiency of vulnerability detection. Profuzzer [33] employs static analysis techniques to analyze the correlation between input bytes and program execution. GreyOne [34] employs stain analysis technology to analyze input data and execute program branches, thereby enhancing the efficiency of the fuzz testing procedure. Meanwhile, numerous studies have employed deep learning techniques to enhance the effectiveness of fuzz testing. NEUZZ [25] employs a neural network model to identify significant bytes in the program input and generates test cases using the gradient model. Zong et al. [35] have developed deep learning algorithms to filter out use cases that are unable to achieve program objectives, thereby enhancing testing efficiency.

Several studies have addressed the optimization of seed scheduling problems by introducing fitness objectives. Hongfuzz [4] adopts a strategy of selecting seed files in order, which is simple and intuitive, but may lack targeted optimization. LibFuzzer [5] prioritizes selecting seeds that can trigger more uncovered program blocks, emphasizing expanding code coverage to discover more potential problem areas. VUzzer [6] prioritizes deep exploration in seed scheduling by focusing on seeds with longer execution paths, including error handling blocks, and high-frequency execution paths. This strategy helps identify deep paths that are difficult to reach with conventional methods while minimizing ineffective exploration on common error-handling paths. Entropic [36] proposes an entropy-based power schedule that aims to allocate more energy to seeds that possess more information. Angora [27] exhibits a preference for specific seeds that have the potential to initiate conditional statements with unexplored branches.

In terms of improving seed energy scheduling efficiency of fuzzing, AFLfast [7] employs a systematic exploration of the Markov chain state space as coverage-based greybox fuzzing modeling. This model defines the execution path of an input seed as a state and calculates transition probabilities based on the variation time of paths during fuzzing. This guides the energy allocation during the fuzzing process. VUzzer [6] treats basic blocks of the program's control flow graph as states, and state transitions are jumps from one basic block to another. Transition probabilities are calculated using Monte Carlo methods, and the fitness value of executing testcase is computed to guide seed energy allocation. The Markov chain model determines seed fitness based on transition probabilities, but this approach suffers from efficiency issues due to the exploration of many low-probability states, leading to reduced decision-making efficiency.

B. MAB MODEL IN FUZZING

The MAB model addresses the challenge of maximizing the cumulative reward in a finite number of trials while making decisions. In the field of fuzzing, there exist numerous scenarios in which it is necessary to optimize the reward. Therefore, numerous studies have been dedicated to utilizing MAB models as a means to address challenges encountered in fuzz testing.

EcoFuzz [8] proposed a VAMAB model to address the issue of seed energy allocation, specifically focusing on the challenge of scaling the number of seeds with the increasing number of tests. This model introduces the concept of seed reward probability and utilizes the Self-Transfer Probability Estimation Method (SPEM) to estimate the value of seeds. AFL-HIER [10] transforms the given path into a multi-armed bandit problem. However, compared to EcoFuzz, AFL-HIER utilizes multi-level coverage metrics, including functions, edges, and basic blocks encountered during test case execution, to assess the efficacy of new seeds. In addition, AFL-HIER addressed the issue of single antagonism by employing the UCB1 algorithm. SLIME [9] integrates relevant attributes obtained from fuzz processes, determines the most suitable attribute queue, and selects the optimal seed based on attribute features [5]. SLIME determines the selected seed energy through the UCB algorithm. MobFuzz [37] has formulated the seed allocation problem as a multi-objective optimization problem and has introduced a multi-player multi-armed bandit (MPMAB) model to effectively schedule optimal seed scheduling.

Patil and Kanade [38] employed a contextual multi-armed bandit machine model to ascertain the allocation of seed energy in the fuzzing process. Additionally, they utilized reinforcement learning techniques to attain the optimal energy allocation. CMFuzz [39] formulates the issue of seed mutation as a contextual multi-armed bandit machine model. CMFuzz dynamically adapts the seed mutation strategy by employing the LinUCB algorithm, which utilizes testcases execution information to modify the uniform mutation strategy employed by AFL.

Unlike previous research that has utilized the MAB model for fuzzing tests, LinFuzz adopts a contextual multi-armed bandit machine model to address seed scheduling issues. It selects the most suitable seed for each round of fuzz testing by leveraging historical data, thereby enhancing the overall efficiency of the fuzz testing process.

VII. CONCLUSION

In this paper, we propose a path sensitive greybox fuzz testing program called LinFuzz. LinFuzz integrates a seed scheduling algorithm based on LinUCB. This algorithm addresses the issue of current seed scheduling strategies overlooking the influence of seed execution information on program fuzz testing efficiency and the inability to prioritize the selection of appropriate seeds based on historical fuzz testing results. Meanwhile, LinFuzz integrates a seed energy allocation algorithm and a queue cull algorithm based on path frequency and danger. Mainly addresses the issue of current fuzz testing tools consuming a significant amount of energy in high-frequency and low-risk paths. We compared our method with other state-of-the-art fuzz testing tools in the open-source test set. The findings indicate that our tool outperforms other tools in terms of path coverage and the number of vulnerabilities discovered, thereby demonstrating the efficacy of our approach.

REFERENCES

[1] P. Godefroid, "Fuzzing: Hack, art, and science," *Commun. ACM*, vol. 63, no. 2, pp. 70–76, Jan. 2020.

[2] 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.

[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] honggfuzz. (2016). *Robert Swiecki*. [Online]. Available: http://honggfuzz.com/

[5] K. Serebryany, "Continuous fuzzing with libFuzzer and AddressSanitizer," in *Proc. IEEE Cybersecur. Develop. (SecDev)*, Nov. 2016, pp. 157–157.

[6] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–14.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1032–1043.

[8] 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.*, 2020, pp. 2307–2324.

[9] C. Lyu, H. Liang, S. Ji, X. Zhang, B. Zhao, M. Han, Y. Li, Z. Wang, W. Wang, and R. Beyah, "SLIME: Program-sensitive energy allocation for fuzzing," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2022, pp. 365–377.

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

[11] Y. Zhang, S. Tong, and L. Cheng, "Evaluation of fuzz testing improvement technology," *Comput. Syst. Appl.*, vol. 31, no. 10, pp. 1–14, 2022.

[12] Google. (2021). *American Fuzz Lop*. [Online]. Available: https://github.com/google/AFL

[13] P. Whittle, "Multi-armed bandits and the Gittins index," *J. Roy. Stat. Soc. B, Stat. Methodol.*, vol. 42, no. 2, pp. 143–149, Jan. 1980.

[14] S. Bubeck, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," *Found. Trends Mach. Learn.*, vol. 5, no. 1, pp. 101–122, 2012.

[15] T. Lu, D. Pál, and M. Pál, "Contextual multi-armed bandits," *J. Mach. Learn. Res.*, vol. 9, pp. 485–492, Jun. 2010.

[16] L. Zhou, "A survey on contextual multi-armed bandits," 2015, *arXiv:1508.03326*.

[17] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proc. 19th Int. Conf. World Wide Web*, 2010, pp. 661–670.

[18] W. Chu, L. Li, L. Reyzin, and R. E. Schapire, "Contextual bandits with linear payoff functions," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2011, pp. 208–214.

[19] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," in *Proc. Netw. Distrib. Syst. Secur. Symp.* San Diego, CA, USA: Internet Society, 2020, doi: 10.14722/NDSS.2020.24422.

[20] G. Choi, S. Jeon, J. Cho, and J. Moon, "A seed scheduling method with a reinforcement learning for a coverage guided fuzzing," *IEEE Access*, vol. 11, pp. 2048–2057, 2023, doi: 10.1109/ACCESS.2022.3233875.

[21] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.* Dallas, TX, USA: ACM, Oct. 2017, pp. 2329–2344, doi: 10.1145/3133956.3134020.

[22] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 110–121.

[23] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2018, pp. 475–485.

[24] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. 14th USENIX Workshop Offensive Technol.*, 2020.

[25] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program smoothing," in *Proc. IEEE Symp. Secur. Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2019, pp. 803–817, doi: 10.1109/SP.2019.00052.

[26] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, and Y. Song, "MOPT: Optimized mutation scheduling for fuzzers," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1949–1966.

[27] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 711–725.

[28] Y. Li, S. Ji, Y. Chen, S. Liang, W. H. Lee, and Y. Chen, "UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 2777–2794.

[29] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided fuzzing with processor trace feedback," *IEEE Access*, vol. 6, pp. 37302–37313, 2018.

[30] Y. Zou, W. Zou, J. Zhao, N. Zhong, Y. Zhang, J. Shi, and W. Huo, "PosFuzz: Augmenting greybox fuzzing with effective position distribution," *Cybersecurity*, vol. 6, no. 1, p. 11, Jun. 2023.

[31] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, and C. Zhou, "EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1967–1983.

[32] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. USENIX Secur. Symp.*, 2018, pp. 745–761.

[33] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 769–786.

[34] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, and D. Wu, "GREYONE: Data flow sensitive fuzzing," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2577–2594.

[35] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2255–2269.

[36] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," *Commun. ACM*, vol. 66, no. 11, pp. 89–97, Nov. 2023.

[37] G. Zhang, P. Wang, T. Yue, X. Kong, S. Huang, X. Zhou, and K. Lu, "MobFuzz: Adaptive multi-objective optimization in gray-box fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2022.

[38] K. Patil and A. Kanade, "Greybox fuzzing as a contextual bandits problem," 2018, *arXiv:1806.03806*.

[39] X. Wang, C. Hu, R. Ma, D. Tian, and J. He, "CMFuzz: Context-aware adaptive mutation for fuzzers," *Empirical Softw. Eng.*, vol. 26, no. 1, pp. 1–34, Jan. 2021.

YINGHAO SU received the bachelor's degree from Space Engineering University, China, in 2021, where he is currently pursuing the Ph.D. degree with the College of Aerospace Information. His research interests include fuzzing test and vulnerability detection.

DAPENG XIONG received the B.S. and M.S. degrees from China University of Geosciences, National University of Defense Technology, in 2010 and 2012, respectively, and the Ph.D. degree in information science from Space Engineering University, China, in 2017. He is currently a Lecturer with Space Engineering University. His major work has been applied to system security and network security analysis. He has published many papers and authored several books. His current research interests include machine learning and vulnerability detection.

YING WAN received the M.S. degree from Space Engineering University, Beijing, China, in 2022. Her main research interests include network system security and software defined network.

CHENGHAO SHI received the M.S. degree from the College of Aerospace Information, Space Engineering University, in 2023. His research interests include reinforcement learning and system security.

QINGYAO ZENG received the bachelor's degree from Space Engineering University, China, in 2022, where he is currently pursuing the M.S. degree with the College of Aerospace Information. His research interests include fuzzing test and system security.