electronics

Article

Reinforcement Learning-Based Multi-Phase Seed Scheduling for Network Protocol Fuzzing

Mingjie Cheng [1,2], Kailong Zhu [1,2,*], Yuanchao Chen [1,2], Yuliang Lu [1,2], Chiyu Chen [1,2] and Jiayi Yu [1,2]

[1] College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; chengmingjie22@nudt.edu.cn (M.C.)
[2] Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China
* Correspondence: zhukailong@nudt.edu.cn

Abstract: In network protocol fuzzing, effective seed scheduling plays a critical role in improving testing efficiency. Traditional state-driven seed scheduling methods in network protocol fuzzing are often limited by imbalanced seed selection, monolithic scheduling strategies, and ineffective power allocation. To overcome these limitations, we propose SCFuzz, specifically by employing a multi-armed bandit model to dynamically balance exploration and exploitation across multiple fuzzing phases. The fuzzing process is divided into initial, middle, and final phases with seed selection strategies adapted at each phase to optimize the discovery of new states, paths, and code coverage. Additionally, SCFuzz employs a power allocation method based on state weights, focusing power on high-potential messages to improve the overall fuzzing efficiency. Experimental evaluations on open-source protocol implementations show that SCFuzz significantly improves state and code coverage, achieving up to 17.10% more states, 22.92% higher state transitions, and 7.92% greater code branch coverage compared to AFLNet. Moreover, SCFuzz improves seed selection effectiveness by 389.37% and increases power utilization by 45.61%, effectively boosting the overall efficiency of fuzzing.

Keywords: protocol fuzzing; multi-armed bandit model; multi-phase; seed scheduling

Citation: Cheng, M.; Zhu, K.; Chen, Y.; Lu, Y.; Chen, C.; Yu, J. Reinforcement Learning-Based Multi-Phase Seed Scheduling for Network Protocol Fuzzing. *Electronics* **2024**, *13*, 4962. https://doi.org/10.3390/electronics13244962

Academic Editor: Myung-Sup Kim

Received: 28 October 2024
Revised: 8 December 2024
Accepted: 11 December 2024
Published: 17 December 2024

1. Introduction

Fuzzing is a widely used and effective software testing technique that is distinguished by its high level of automation, extensive applicability, and demonstrated efficacy in identifying software vulnerabilities [1]. At its core, fuzzing operates through an iterative process where input seeds are selected from an initial seed pool and modified through mutations to create diverse test cases; then, these cases are executed on the target software.

In recent years, the focus of fuzzing research has primarily centered on coverage-based graybox fuzzing (CGF), which improves testing efficiency by leveraging feedback from code coverage [2]. Building upon the foundations of CGF, researchers have extended these techniques to the domain of network protocol testing, leading to the development of stateful coverage-based graybox fuzzing (SCGF). SCGF not only relies on code coverage information but also integrates state information extracted from network protocol implementations. This allows it to manage complex protocol state transitions and interdependent interactions, providing a more nuanced and effective approach to identifying vulnerabilities in network protocol implementations.

An essential aspect of effective fuzzing lies in seed scheduling, which significantly impacts the efficiency and effectiveness of the fuzzing process. Seed scheduling generally comprises two core tasks: seed selection and power allocation. Seed selection determines which inputs from the seed pool should be prioritized for testing, while power allocation assigns computational effort to these selected seeds to maximize test coverage. Unlike CGF, where seed scheduling is typically guided by feedback on code coverage alone, SCGF

must also account for the specific states within the network protocol implementation. This necessitates first identifying the target state for testing and then selecting seeds capable of triggering this state. By incorporating protocol state information into the scheduling process, SCGF can more effectively navigate and explore complex state spaces.

Current research on seed scheduling within SCGF primarily focuses on state representation methods [3–5]. However, regardless of the state representation method used, the seed scheduling methods in network protocol fuzzing face three significant challenges:

Challenge 1: Imbalanced Seed Selection Strategy. Existing seed selection methods tend to repeatedly select a small subset of high-frequency seeds, resulting in the excessive testing of high-frequency states and code paths while neglecting the exploration of other potentially critical states and paths. This imbalance diminishes the breadth of fuzzing, limiting the effective utilization of valuable seeds within the seed pool and hindering a comprehensive exploration of the code and state space of the protocol implementation.

Challenge 2: Monolithic Seed Selection Strategy. The commonly adopted strategy is state-driven seed selection, where the target state is first chosen, and then a message sequence capable of reaching that target state is selected from the seed pool. However, in the later phases of fuzzing, the number of states tends to stabilize and remains relatively small. As shown in Table 1, across 13 protocol implementations in the ProFuzzBench [6], nearly two thirds of protocol implementations typically contain no more than 25 states regardless of the state representation method used. In contrast, the number of seeds can reach thousands. This single-pattern approach can result in numerous potentially valuable seeds being filtered out during the state selection phase, preventing them from progressing to the subsequent mutation phase.

Challenge 3: Uneven Power Allocation Strategy. Existing power allocation strategies typically concentrate power on mutating specific messages within target states, overlooking the potential value of other messages. Since seed files are often composed of multiple messages, this overly concentrated power allocation leads to imbalanced resource distribution, preventing the full exploitation of the seed file's potential and diminishing the overall efficiency of the fuzzing process.

Table 1. Comparison of state inference results by NSFuzz on the ProfuzzBench.

Subjects	AFLNet	StateAFL	NSFuzz
LightFTP	23	51	5
Bftpd	24	4	15
Pure-FTPd	29	3	6
ProFTPD	27	16	37
Dnsmasq	102	95	2
TinyDTLS	8	28	7
Exim	12	9	5
Kamailio	13	3	5
OpenSSH	42	48	19
OpenSSL	8	24	11
Forked-daapd	8	6	9
Live555	10	18	15
Dcmtk	3	11	7

In this paper, we propose SCFuzz, which is an effective solution to the seed scheduling challenges in protocol fuzzing. To address the first challenge, we model the seed selection problem as a multi-armed bandit problem, where the seed selection process must balance exploration and exploitation. This ensures that the fuzzer can discover new states and paths while also making efficient use of already identified high-value seeds. For the second challenge, we divide the fuzzing process into multiple phases, employing the multi-armed bandit model at each phase to guide seed selection. The selection strategy is dynamically adjusted according to the needs of each phase, enabling the scheduling mechanism to adapt to the increased number of seeds and the saturation of state discovery in the later

phases. This strategy ensures that valuable seeds within the seed pool are fully utilized, even as the test progresses. To address the third challenge, we refine power allocation by analyzing the states triggered by selected seeds and leveraging the weights of messages corresponding to those states. This allows power to be concentrated on messages more likely to uncover new states or paths, thereby enhancing the overall efficiency and coverage of the fuzzing process.

Specifically, we use the multi-armed bandit algorithm from reinforcement learning to address the balance between exploration and exploitation. In addition, we divide the fuzzing process into three distinct phases and dynamically adjust the focus of seed selection based on the testing requirements of each phase. In the initial phase, the fuzzing process is able to trigger numerous new states and state transitions, so the focus should be on broad exploration of the state space to maximize the discovery of new states. During the middle phase, the discovery of new states becomes increasingly difficult, and only a small number of state transitions are typically captured. At this point, the strategy must consider both state and path selection comprehensively. In the final phase, the state space approaches saturation, and the discovery of new states and transitions nearly ceases, leaving only a few new paths to be uncovered. Once seeds are selected, we analyze all potential target states by calculating the weights of the states triggered. Given that different states have varying levels of importance, we employ a set of heuristic rules to calculate the weight of each state. Based on these weights, we perform fine-grained power allocation by distributing more power to the messages corresponding to critical states. This enables a more precise allocation of resources, thereby enhancing the coverage and overall effectiveness of the fuzzing process.

We implemented a prototype of SCFuzz and evaluated its effectiveness on five widely used open-source protocol implementations. The evaluation results demonstrate that SCFuzz offers significant improvements over the state-of-the-art graybox protocol fuzzer, AFLNET, across multiple metrics. Specifically, SCFuzz achieved an average improvement of 17.10% in the number of states discovered, a 22.92% improvement in state transitions, and a 7.92% improvement in code branch coverage. Additionally, seed selection effectiveness improved by an average of 389.37%, while power utilization effectiveness improved by 45.61%.

In summary, the main contributions of this paper are as follows:

- We propose a multi-phase seed selection strategy, where the fuzzing process is divided into several phases. The seed selection problem is modeled as a multi-armed bandit problem, and the selection strategy is dynamically adjusted at each phase to prioritize seeds with the greatest potential to discover new states and paths.
- We propose a power allocation strategy based on state weights. By calculating the weight of messages corresponding to states within the selected seed, the strategy allocates more power to the messages deemed more significant.
- We implemented SCFuzz and evaluated it on widely used protocol implementations, and the results show that SCFuzz outperforms the state-of-the-art fuzzer AFLNET across several key metrics, including state and code coverage, seed selection effectiveness, and power utilization effectiveness.

The paper is organized as follows. Section 2 provides the background of the study. Section 3 reviews the related work in the field. Section 4 offers a detailed description of the proposed methodology. Section 5 outlines the experimental setup and presents the results. Section 6 discusses the limitations of the proposed approach. Finally, Section 7 concludes the paper and suggests directions for future work.

2. Background

In this section, we introduce the background of CGF, SCGF, and the multi-armed bandit model.

2.1. CGF

CGF is one of the most popular and effective techniques in the field of vulnerability discovery, particularly in the detection of vulnerabilities in stateless programs. The core idea of CGF is to collect code coverage information through instrumentation and use this information to guide the fuzzing process, thereby maximizing code coverage. CGF inserts lightweight code instrumentation at critical locations to monitor execution and capture coverage information. During the execution of test cases, these monitoring points record the triggered code paths. If a test case triggers a new path or covers previously unexplored code, it is added to the seed pool for subsequent fuzzing.

Although CGF performs well in vulnerability discovery for stateless programs, it faces many challenges when applied to stateful network protocols. Network protocols are inherently stateful, as their execution depends on the context and sequence of preceding messages. This characteristic complicates the exploration of the state space using traditional stateless fuzzing strategies. Moreover, the input and output of network protocols often rely on intricate timing and event-driven mechanisms, further increasing the complexity of fuzzing. Since CGF primarily focuses on code coverage and inadequately handles state information, its effectiveness is limited when dealing with complex network protocols.

2.2. SCGF

To address the limitations of CGF in handling stateful network protocols, researchers have developed SCGF. SCGF combines the strengths of traditional CGF, which leverages code coverage to guide fuzzing. At the same time, it enhances the management of state information, allowing more efficient exploration of the state space in stateful programs. SCGF captures critical information during the execution of network protocols, infers the state of the server, and dynamically constructs a state machine. This state machine then guides the fuzzing process, prioritizing the exploration of states and code paths that are more likely to expose vulnerabilities.

Figure 1 illustrates the general workflow of SCGF. Prior to initiating fuzzing, testers capture network communication data between the client and server (e.g., pcap files generated by tcpdump) to obtain a series of actual message exchange sequences. These message sequences are then parsed and used to construct the initial seed pool. With this initial seed pool, SCGF initiates the fuzzing process, which can be divided into four phases during the fuzzing loop:

Phase 1: Target State Selection. SCGF prioritizes states that were less explored during the fuzzing process or those that may lead to new state transitions to enhance the coverage of the protocol's state space.

Phase 2: Message Sequence (Seed) Selection. Once the target state is identified, SCGF selects a message sequence M from the seed pool that can reach that state.

Phase 3: Sequence Mutation. SCGF divides the original message sequence M into three parts: the prefix M_1, which is the message sequence necessary to execute the target state s; the infix M_2, containing all messages executed after the prefix M_1 while still remaining in state s; and the suffix M_3, which is the remaining sequence of messages. SCGF mutates only the infix M_2 to ensure that the mutated sequence M' can still maintain the stability of the target state s.

Phase 4: Execution and Feedback Collection. The mutated test cases are sent to the Service Under Test (SUT), and the feedback collected during execution is used to update the state machine and bitmap.

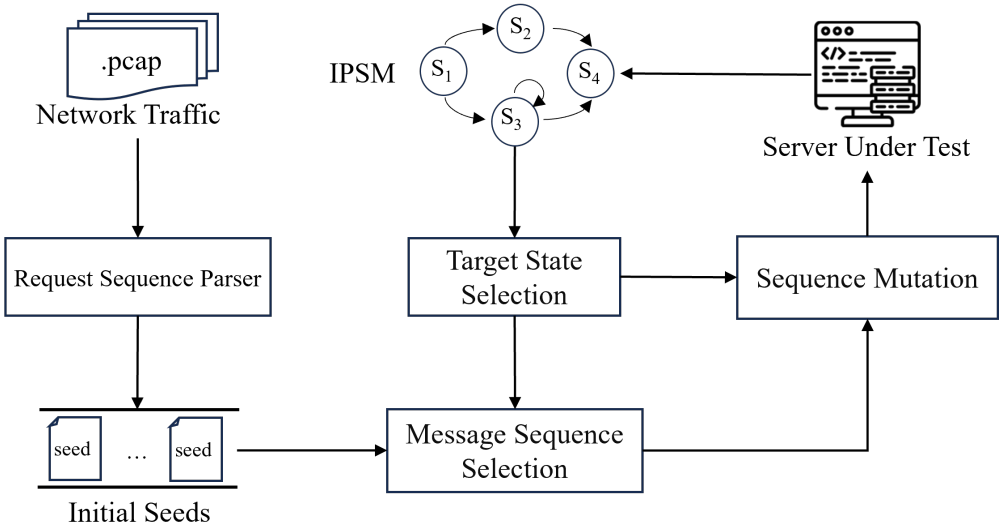

Figure 1. The general workflow of SCGF.

2.3. Multi-Armed Bandit Model

The multi-armed bandit (MAB) model is a classic optimization problem widely applied in statistics, machine learning, and decision theory [7]. The name originates from slot machines in casinos, where each machine has multiple arms, and each pull of an arm yields a reward. The core of the MAB problem lies in balancing exploration and exploitation: within limited resources or time, the decision-maker must choose among several available arms, each of which may produce different rewards, but the distribution of these rewards is unknown. Through continuous trials, the decision-maker gradually learns the expected value of rewards for each arm, aiming to maximize cumulative rewards in the long run.

In fuzzing, the seed selection problem bears significant similarity to the MAB problem, as it also involves balancing exploration and exploitation. Under constraints of limited time and resources, the tester must weigh between the use of known effective seeds and the exploration of new ones to achieve an optimal strategy for discovering vulnerabilities and maximizing code coverage. However, traditional MAB models assume a fixed number of arms with the reward of each arm being either deterministic or following a predefined distribution. In fuzzing, high-performing test cases during execution are saved in a seed pool, resulting in a continually increasing number of seeds that each have dynamically changing potential as testing progresses. Therefore, traditional MAB models cannot be directly applied to the seed selection problem in fuzzing.

The adversarial multi-armed bandit (Adversarial MAB) [8] is a variant of the traditional MAB problem, where the reward distribution of each arm can dynamically change over time. Unlike the standard MAB problem, Adversarial MAB requires algorithms to quickly adapt and respond to these reward changes to maximize cumulative rewards in a dynamic environment. This characteristic makes Adversarial MAB particularly well suited for the seed selection process in fuzzing. In this context, seeds can be viewed as the arms of the bandit. As fuzzing progresses, the effectiveness of seeds tends to diminish because the probability of discovering new code coverage or vulnerabilities decreases over time. Therefore, the Adversarial MAB model can dynamically adjust seed selection strategies according to the changing effectiveness of seeds throughout the testing process, thereby maximizing the efficiency of fuzzing at different phases.

3. Related Work

3.1. Protocol Fuzzing

The development of network protocol fuzzing has primarily gone through two stages: black-box fuzzing and gray-box fuzzing . Black-box protocol fuzzing [9–13] does not rely on the internal structure of protocol implementations; it focuses solely on inputs and outputs

and generates test cases based on prior knowledge of the protocol format. Peach [14] configures "models" described in XML to automatically generate diverse test cases, while SPIKE [15] uses a block-based testing approach, dividing protocols into multiple blocks and generating different test cases according to predefined rules. These tools effectively generate test cases without requiring internal protocol information, but they are limited by low coverage and struggle to identify complex logical flaws.

Graybox protocol fuzzing [5,16–19] uses feedback information from the execution process to guide the generation of test cases, thereby enhancing testing effectiveness. Pham et al. [20] developed the first stateful graybox protocol fuzzing tool, AFLNet, based on AFL [21]. Following this, graybox protocol fuzzing has rapidly advanced. For example, SGPFuzzer [22] improves the effectiveness of generated test cases by refining message sequence mutation strategies, while SnapFuzz [23] employs snapshot technology to increase the speed of test case generation and system throughput.

3.2. Seed Scheduling

Research on the seed scheduling phase of stateful protocol fuzzing is relatively limited with most studies focusing on state selection and state representation methods. Some researchers have attempted to optimize state selection methods to enhance the effectiveness of seed scheduling. Liu et al. [24] introduced the Monte Carlo Tree Search [25] in AFLNetLegion, expanding the server state model into a tree structure and distinguishing state nodes based on response codes. Borcherding et al. [26] proposed modeling the state selection problem as a multi-armed bandit problem [7], optimizing the state selection process by balancing exploration and exploitation. Additionally, some studies have sought to improve seed scheduling through more precise state representations. StateAFL [3] captures snapshots of long-lived memory and employs a locality-sensitive hashing algorithm to map memory content to unique state identifiers. In contrast, NSFuzz [4] and SGFUZZ [5] utilize static analysis and lightweight instrumentation to identify state variables in the source code of protocol implementations, using the values of these variables to define states.

Although research on seed scheduling in stateful protocol fuzzing is limited, there has been extensive investigation into seed scheduling in traditional coverage-based fuzzing. The first category of methods tends to assign higher potential to seeds that explore paths with lower execution frequencies [27–30]. AFLFast [31] employs a Markov chain model to explain that deeper paths require more executions, thereby allocating more power to low-frequency paths. Truzz [32] prioritizes seeds with higher potential to discover new paths. FairFuzz [33] enhances test coverage by allocating power toward seeds that are likely to reach rare branches, thereby controlling the mutation frequency of these seeds. The second category prefers mutating high-benefit seeds that are closer to uncovered code blocks [34,35]. K-scheduler [35] converts the control flow graph into an edge horizon graph and calculates seed scores based on graph centrality, where higher scores indicate higher seed rewards. BELIEFuzz [36] calculates the potential number of uncovered paths for each seed based on the control flow graph, combining this with execution frequency to comprehensively evaluate the mutation potential of each seed.

4. Methodology

To overcome the limitation of traditional state-driven seed selection, we first provide an overview of the overall design of SCFuzz and then explain its key technical aspects.

4.1. Overview

As shown in Figure 2, SCFuzz consists of three primary modules: **Multi-Phase Seed Selection**, **Power Allocation Based on State Weights**, and **Mutation and Execution**.

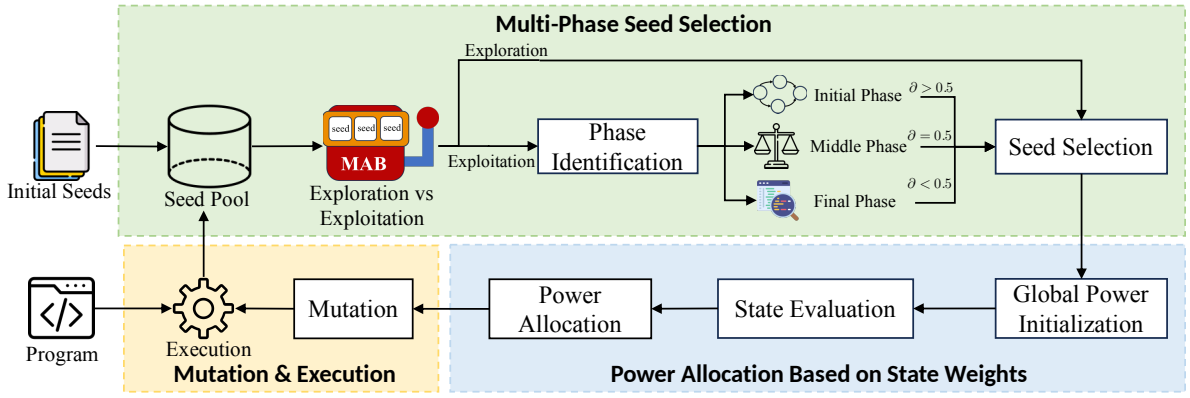

Figure 2. Overview of SCFuzz.

Multi-Phase Seed Selection. We propose a mutil-phased seed selection strategy, departing from traditional state-driven seed selection methods. This strategy employs a multi-armed bandit model to dynamically balance exploration and exploitation across different phases of the fuzzing process. The fuzzing process is divided into initial, middle, and final phases with the seed reward probability calculation method adjusted at each phase to optimize the discovery of new states, paths, and code coverage.

Power Allocation Based on State Weights. To improve the effective utilization of power, we employ a power allocation strategy based on state weights. In protocol fuzzing, seeds are composed of sequences of multiple messages, requiring power allocation to individual messages. By calculating the weights of the states corresponding to each message sequence, we indirectly assess the exploration potential of the protocol implementation in each state. Based on these weights, power is allocated appropriately to the corresponding messages.

Mutation and Execution. We mutate the messages corresponding to each state based on the power allocated to the states in the seed. The power assigned to a state determines the number of mutations applied to its associated messages with states receiving more power undergoing more mutation operations. After mutation, the seed generates new test cases, which are then sent to the target server for testing. During testing, the test cases are evaluated and filtered based on the feedback received. If a test case triggers new state coverage or code coverage, it is saved and added to the seed pool for subsequent testing.

In the following, we present a detailed description of the core designs of SCFuzz, including *Multi-Phase Seed Selection* and *Power Allocation Based on State Weights*.

4.2. Multi-Phase Seed Selection

In this subsection, we introduce multi-phase seed selection, modeling it as a multi-armed bandit problem, dividing fuzzing into three phases, and detailing seed selection methods for each phase.

4.2.1. Seed Selection Model

Before modeling the seed selection problem as an MAB problem, we propose the following two assumptions for the server under test, which are denoted as *SUT*:

Assumption 1. *Under the same input conditions, the execution results of the SUT should remain consistent. Specifically, the triggered states and executed code paths are expected to remain consistent across executions.*

This assumption establishes consistency in the behavior of the SUT under identical input conditions, thereby ensuring the repeatability and reliability of the testing outcomes.

Assumption 2. *The number of state chains and code paths in SUT is finite. The set of state chains is denoted as* $SC = \{sc_1, sc_2, \ldots, sc_{n_1}\}$*, and the set of paths id denoted as* $CP = \{cp_1, cp_2, \ldots, cp_{n_2}\}$.

This assumption ensures that the number of state chains and code paths in the SUT is limited, thereby simplifying the analysis process and enhancing the controllability of program behavior.

Building on the two assumptions above, we introduce the following key definitions, laying the theoretical foundation for the subsequent seed selection strategy.

Definition 1. *At time t, the set of seeds in the seed pool is denoted as* $Seed = \{seed_1, \ldots, seed_m\}$*. The set of state chains triggered by these seeds is denoted as* $SC = \{sc_1, sc_2, sc_3, \ldots, sc_{m_1}\}$*, and the set of code paths is denoted as* $CP = \{cp_1, cp_2, cp_3, \ldots, cp_{m_2}\}$.

The definition clarifies the set of seeds in the seed pool during protocol fuzzing as well as the corresponding state chains SC triggered and the paths CP covered. The state chains SC represent the state transitions triggered by the seeds, while the paths CP reflect the code paths that have been executed. By analyzing the state chains and paths, the effectiveness of the fuzzing can be evaluated, and the seed selection strategy can be optimized to improve coverage further.

Definition 2. ***Transition Probability****. At time t, assume that a seed* $seed_t$ *in the seed pool triggers a state chain* sc_i *and a path* cp_i*. The state chain transition probability* $p_{sc_i sc_j}$ *represents the probability that a test case mutating the seed triggers state chain* sc_j*. Similarly, the path transition probability* $p_{cp_i cp_j}$ *represents the probability that a test case mutating the seed executes code path* cp_j.

This definition explains that in protocol fuzzing, transition probabilities measure the likelihood of generating different state chains and code paths during seed mutation. These probabilities highlight the seeds most likely to improve coverage, providing important guidance for optimizing the fuzzing process.

Definition 3. ***Transition Frequency****. Transition frequency quantifies the rate at which a mutated seed* $seed_t$ *triggers a specific state chain* sc_j *or executes a specific code path* cp_j *during the testing process.*

- *The transition frequency for state chains is calculated as follows:*

$$f_{sc_i sc_j} = \frac{f_{sc_i}(sc_j)}{f(seed_t)} \tag{1}$$

  where $f_{sc_i}(sc_j)$ *denotes the number of times the mutated seed* $seed_t$ *triggers the state chain* sc_j*, and* $f(seed_t)$ *represents the total number of mutations for seed* $seed_t$.
- *Similarly, the transition frequency for code paths is given by the following formula:*

$$f_{cp_i cp_j} = \frac{f_{cp_i}(cp_j)}{f(seed_t)} \tag{2}$$

  where $f_{cp_i}(cp_j)$ *denotes the number of times the mutated seed* $seed_t$ *executes the code path* cp_j*, and* $f(seed_t)$ *represents the total number of mutations for seed* $seed_t$.

Transition frequency provides a quantitative measure of the likelihood that a given seed will generate specific state chains or code paths, offering critical insights for seed scheduling in fuzz testing. A higher transition frequency suggests that the seed is more likely to produce specific state chains or code paths, thereby aiding in increasing test coverage and exploration efficiency.

Definition 4. ***Reward Probability***. *In protocol fuzzing, the reward for each seed mutation is defined as the test case that generates and triggers a new state chain or executes a new code path. When seed $seed_t$ is selected for testing and receives a reward, denoted as $R(seed_t) = 1$, the probability of the mutated seed triggering a new state chain is given by $P(sc_{new} \mid R(seed_t) = 1)$. Similarly, the reward probability for executing a new code path is $P(cp_{new} \mid R(seed_t) = 1)$.*

Based on Definitions 1 and 2, the probability of seed $seed_t$ receiving a reward for triggering a new state chain can be derived as follows:

$$\begin{aligned} P(sc_{\text{new}} \mid R(seed_t) = 1) &= \sum_{j=m_1+1}^{n_1} p_{sc_i sc_j} \\ &= 1 - \sum_{j=1}^{m_1} p_{sc_i sc_j} \end{aligned} \tag{3}$$

Similarly, the probability of seed $seed_i$ receiving a reward for executing a new code path can be derived as shown below:

$$\begin{aligned} P(cp_{\text{new}} \mid R(seed_t) = 1) &= \sum_{j=m_2+1}^{n_2} p_{cp_i cp_j} \\ &= 1 - \sum_{j=1}^{m_2} p_{cp_i cp_j} \end{aligned} \tag{4}$$

Based on the above assumptions and definitions, we map the seed selection problem in SCGF to an adversarial multi-armed bandit problem. The task of the agent is to select a seed from the seed pool for fuzzing and choose a seed that maximizes both state coverage and code coverage.

Arms. Let $\mathcal{A}$ represent the set of arms available for selection by the agent. Each arm $a_i \in \mathcal{A}$ corresponds to a distinct seed t_i within the seed pool. In each selection round, the agent is constrained to selecting a single arm, which corresponds to choosing one seed for fuzzing. The outcome of this selection influences future decisions, as the agent iteratively adjusts its strategy based on feedback, progressively selecting the most promising seed from the pool.

Reward Probability. In the MAB model for seed selection, it is necessary to calculate the expected reward for each arm in order to select the arm with the highest expected reward, i.e., the seed with the greatest potential to discover a new state or code coverage. However, accurately calculating the expected reward for each arm is impractical. To address this issue, we propose measuring the potential of a seed based on its reward probability. The greater the seed's reward probability, the higher the likelihood of discovering new state or code coverage. Since the goal of seed selection is to maximize state and code coverage, the reward probability of a seed can be formulated by combining Equations (3) and (4) as outlined below:

$$P(R(seed_t) = 1) = \alpha \cdot P(sc_{\text{new}} | R(seet_t) = 1) + (1 - \alpha) \cdot P(cp_{\text{new}} | R(seed_t) = 1) \tag{5}$$

where $\alpha \in [0, 1]$.

As shown in Equations (3) and (4), the calculation of the reward probability depends on the transition probability. In AFLFast [31], transition probabilities are computed through path constraint analysis; however, this method becomes almost infeasible for programs with complex constraint relationships. To address this issue, we adopt the self-transition frequency estimation method used in EcoFuzz [28] to compute the transition probability of each seed. Since the reward probability gradually decays during the fuzzing process, we balance this characteristic using the seed index. According to Equations (1) and (3), the transition probability of the state chain can be approximately formulated as

$$P(sc_{\text{new}}|R(seed_t) = 1) \approx 1 - \frac{f_{sc_i sc_i}}{\sqrt{index(seed_t)}} \tag{6}$$

Similarly, the transition probability of the path can be approximately formulated as

$$P(cp_{\text{new}}|R(seed_t) = 1) \approx 1 - \frac{f_{cp_i cp_i}}{\sqrt{index(seed_t)}} \tag{7}$$

As shown in Equations (5)–(7), the reward probability of a seed can be approximated as

$$P(R(seed_t) = 1) \approx \alpha \cdot \left(1 - \frac{f_{sc_i sc_i}}{\sqrt{index(seed_t)}}\right) + (1 - \alpha) \cdot \left(1 - \frac{f_{cp_i cp_i}}{\sqrt{index(seed_t)}}\right) \tag{8}$$

where $\alpha \in [0, 1]$.

4.2.2. Exploration vs. Exploitation

After modeling the seed selection problem as a multi-armed bandit problem, each seed selection involves a balance between exploration and exploitation. Exploration refers to selecting different arms to gather more information, even though these arms may not necessarily provide the maximum immediate reward. However, through exploration, we can gain a more comprehensive understanding of the potential reward distribution of each arm. Exploitation, on the other hand, refers to selecting the arm expected to yield the highest reward based on current observations to maximize immediate rewards. Nevertheless, since the known information is based on limited observations, the currently optimal arm may not be the globally optimal one. Therefore, it is essential to find a reasonable balance between exploration and exploitation to maximize cumulative rewards.

In fuzzing, the seed selection process involves balancing exploration and exploitation, which essentially means choosing between selected and unselected seeds. The seed pool can be categorized into two states: *exploration* and *exploitation*. During the *exploration* state, when there are unselected seeds in the pool, these seeds should be prioritized to ensure that each one is thoroughly tested. Once all seeds have been selected, the pool transitions into the *exploitation* state, where Equation (8) is applied to assess the seeds and select the most promising one for further testing.

4.2.3. Phase Identification

The goal of protocol fuzzing is not only to maximize code coverage but also to maximize state coverage. New states are typically discovered during the initial phase of protocol fuzzing. As fuzzing progresses, it becomes increasingly challenging to find new states in the middle phase with only a few state transitions captured. In the final phase, it is nearly impossible to discover new states or state transitions, and only a limited number of new code paths are covered. Therefore, the protocol fuzzing process is divided into three phases, each with distinct objectives based on the evolving goals of the test.

The phases of fuzzing are mainly divided based on runtime. Specifically, the tester needs to clearly define the runtime intervals for the initial, middle, and final phases before the fuzzing starts. By dividing time in this way, the fuzzing can adjust its strategy according to the needs of different phases, thereby improving the coverage of states and code paths.

SCFuzz compares the current runtime in real time with the pre-set phase time to determine the current phase of the fuzzing. The focus of testing shifts according to the characteristics of each phase: in the initial phase, the emphasis is on broadly covering protocol states and state transitions, so $\alpha > 0.5$ is set in the seed reward probability evaluation shown in Equation (8); in the middle phase, the testing balances the coverage of states and code paths, gradually investing more effort into exploring code paths, setting $\alpha = 0.5$; in the final phase, as the exploration of the state space approaches saturation, the focus shifts to thoroughly exploring the uncovered code paths with $\alpha < 0.5$ to increase the depth of testing in the code space.

4.2.4. Seed Selection

After completing the "Exploration vs. Exploitation" and "Phase Identification" steps, the state of the seed pool and the current fuzzing phase are determined. This allows SCFuzz to define the scope for seed selection and the formula for calculating seed reward probabilities.

Algorithm 1 illustrates the seed selection process in detail. The algorithm takes the seed pool, the state of the seed pool, and the current fuzzing phase as inputs. First, the algorithm checks the state of the seed pool. If the seed pool is in the *exploration* state, meaning there are seeds that have not yet been selected, a random selection is made from these unfussed seeds (lines 1–2). If the seed pool is in the *exploitation* state, where all seeds have already been selected, the algorithm selects a seed based on the reward probability of each seed in the seed pool (lines 4–15). The reward probability is calculated according to Equation (8) with the parameter α determined by the current fuzzing phase.

Algorithm 1 Seed Selection

Input: P: Seed pool, P_{state}: State of the seed pool, F_{phase}: Fuzzing phase
Output: *seed*: Selected seed

1: **if** P_{state} is *exploration* **then**
2: seed ← RandomChooseFromUnfuzzedSeeds (P)
3: **else**
4: **if** F_{phase} is *initial* **then**
5: $\alpha \leftarrow \delta_1$
6: **else if** F_{phase} is *middle* **then**
7: $\alpha \leftarrow \delta_2$
8: **else**
9: $\alpha \leftarrow \delta_3$
10: **end if**
11: **for** $i = 0$ to $length(P) - 1$ **do**
12: **if** $i = 0$ or computeProbability ($P[i], \alpha$) > computeProbability ($P[seed], \alpha$) **then**
13: seed ← $P[i]$
14: **end if**
15: **end for**
16: **end if**
17: **return** seed

During the initial phase of fuzzing, α is set to a higher value, prioritizing state exploration. In the middle phase, α is set to a moderate value, balancing both state and code path coverage. In the final phase, α is reduced to favor the deeper exploration of code paths. Once the value of α is determined, the reward probabilities of all seeds in the pool are computed, and the seed with the highest reward probability is selected for the next round of fuzzing.

4.3. Power Allocation Based on State Weights

In this subsection, we propose a fine-grained power allocation method based on state weights. The process begins with global power initialization, which allocates initial power to the selected seed to ensure optimal resource distribution. Next, state evaluation calculates the weights of the states corresponding to each message in the seed. Finally, fine-grained power allocation assigns greater power to messages linked to states with more weight, thereby enhancing the overall coverage and effectiveness of fuzzing.

4.3.1. Global Power Initialization

SCFuzz initializes power for the selected seed based on the power initialization strategy of the existing protocol graybox fuzzing tool, AFLNet. This strategy considers factors such as seed execution time, code coverage, and introduction timing within the test. Through this mechanism, higher initial power is directed to seeds that are efficient in execution, achieve greater coverage, or are newly introduced, thereby increasing their chances

of mutation in subsequent fuzzing cycles. By dynamically optimizing resource distribution in real time, this power initialization strategy enhances the overall testing efficiency.

4.3.2. State Evaluation

In protocol fuzzing, seeds are composed of message sequences, making it impractical to mutate the entire seed directly. Instead, power needs to be allocated to individual messages within the seed. Since it is difficult to directly assess the mutation potential of a single message, we evaluate the states corresponding to the message sequence to estimate the mutation potential of each message, thereby enabling a more effective allocation of power.

Not all states possess the same mutation potential. We apply the following heuristic rules to quantify the weight of each state within the seed.

- Allocate more power to states with lower selection frequency. States that are selected less often indicate they have rarely been explored in previous fuzzing attempts. Due to insufficient testing, these states may hide potential vulnerabilities or undiscovered behaviors and should therefore be prioritized for power allocation.
- Allocate more power to states with lower trigger frequency. States that are triggered less often during testing may have more untapped potential. Prioritizing these states can help uncover more unknown behaviors and security risks.
- Allocate more power to states with a higher number of state transition edges. States with more transition edges are usually more complex and may lead to multiple untested paths. Allocating more power to these states can significantly increase path coverage.
- Allocate more power to states that are deeper in the implemented protocol state machine. Deeper states, which require multiple transitions to reach, typically indicate higher testing difficulty. By allocating more power to these states, we ensure that these hard-to-reach states are adequately tested.
- Allocate more power to states that have historically discovered new paths multiple times. These states have already proven their potential to trigger new execution paths during previous testing. Continuing to allocate power to these states could reveal even more new paths, boosting the overall effectiveness of fuzzing.

Based on the five heuristic rules mentioned above, we design a formula for calculating the state weight. For a given state s_i, its weight is calculated as follows:

$$w_{s_i} = \frac{discoverd\_paths_{s_i} \times depth_{s_i} \times trans_{s_i}}{\lg(\lg(fuzzs_{s_i} + 2) \times selected\_times_{s_i} + 2)} \tag{9}$$

In Equation (9), $discoverd\_paths_{s_i}$ represents the number of new paths discovered when state s_i is selected as the target state for fuzzing. $depth_{s_i}$ refers to the depth of state s_i within the state machine, while $trans_{s_i}$ denotes the number of transitions from state s_i. $fuzzs_{s_i}$ represents the number of times state s_i has been triggered, and $selected\_times_{s_i}$ represents the number of times state s_i has been selected as the target state. Since the magnitudes of $fuzzs_{s_i}$ and $selected\_times_{s_i}$ are relatively large, to avoid an excessively large denominator and consequently low weight differentiation, Equation (9) reduces their magnitude. Using the above equation, we can perform precise quantitative evaluations of each state that the selected seed can trigger, thereby gaining a more comprehensive understanding of the testing potential and value of these states.

4.3.3. Fine-Grained Power Allocation

After determining the weights of the states that the selected seed can trigger, we need to further allocate the global seed power to each state in a fine-grained manner. Based on the weight of each state, the seed power is proportionally distributed among different states, ensuring that states with higher weights receive more testing resources. This approach effectively increases the mutation frequency of high-potential states, thereby enhancing the overall coverage and efficiency of the fuzzing process.

In the global power initialization phase, assuming the total power of the seed is *power_total*, the power allocated to each of the n states in the seed file is calculated as follows:

$$power_{s_i} = power\_total \times \frac{w_{s_i}}{\sum_{i=1}^{n} w_{s_i}} \tag{10}$$

where, w_{s_i} represents the weight of state s_i, and $\sum_{i=1}^{n} w_{s_i}$ is the total sum of the weights for all states. This allocation method ensures that states with higher weights receive more power, allowing rare or high-potential states to be prioritized for testing, thereby improving the overall coverage and efficiency of the fuzzing process.

Allocating power to the states within a seed essentially means allocating power to individual messages within the seed. Taking the seed and its corresponding state chain in Figure 3 as an example, the weight of each state in the chain is calculated to assess its mutation potential, after which the power is distributed accordingly. For instance, if the power allocated to state S_2 is $power_1$, the subsequent message M_3 will be assigned $power_1$ units of power, meaning that M_3 will undergo $power_1$ mutation attempts.

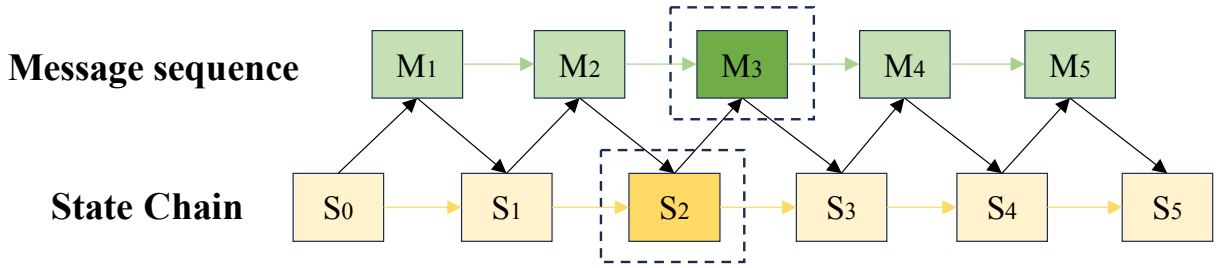

Figure 3. An example to explain the relationship between message sequence and state chain.

5. Evaluation

In this section, we evaluate the performance of SCFuzz and compare it to state-of-the-art protocol fuzzers. We concentrate on the following research questions in the evaluation of SCFuzz.

- RQ1: **State coverage.** How much improvement in state coverage does SCFuzz achieve compared to the baseline?
- RQ2: **Code coverage.** How much improvement in code coverage does SCFuzz achieve compared to the baseline?
- RQ3: **Seed selection effectiveness.** How does SCFuzz enhance seed selection effectiveness compared to the baseline?
- RQ4: **Power utilization effectiveness.** How does SCFuzz improve power utilization effectiveness compared to the baseline?

5.1. Experimental Setup

Implementation. We develop SCFuzz based on the widely used protocol fuzzing framework AFLNET. We run all experiments on a server equipped with 64-bit Ubuntu 20.04, featuring dual Intel (R) Xeon (R) E5-2690@2.90 GHz CPUs and 128 GB of RAM. Each selected protocol implementation and each fuzzer are individually set up in separate Docker containers, utilizing identical computational resources for experimental evaluation. To ensure the fairness of the experimental results, each fuzzer was subjected to 24 h of fuzzing on each protocol implementation with the experiments repeated five times.

In the 24 h of fuzzing, we divide the process into three phases. The first 6 h represent the initial fuzzing phase, during which we set the parameter α to 0.8, focusing primarily on exploring the state space. The middle 8 h represent the mid-fuzzing phase, with α set to 0.5, balancing the exploration of both the state space and code space. Finally, the final 10 h represent the late fuzzing phase, where α is set to 0.2, focusing more on code space exploration.

Benchmark. Table 2 provides detailed information on the network protocol implementations used in our evaluation. We select fuzzing targets from the network protocol fuzzing benchmark, i.e., ProFuzzBench [6]. Our evaluation encompasses three distinct network protocols, which are represented by five different network service implementations, including widely used protocols such as FTP, RTSP, and DAAP.

Table 2. The basic information of target protocol implementations.

Subject	Protocol	Version	Language
Live555	RTSP	31284aa	C++
LightFTP	FTP	139af7c	C
Pure-FTPD	FTP	10122d9	C
ProFTPD	FTP	61e621e	C
Forked-daapd	DAAP	2ca10d9	C

Baseline. We conducted a comparison between SCFuzz and the baseline open-source graybox fuzzing tool for network protocols, AFLNet. AFLNet is specifically designed for fuzzing stateful network protocols. It extends the widely used AFL by introducing a state-aware feedback strategy. It monitors protocol-specific state transitions during fuzzing, enabling better coverage in stateful network services. Other widely used fuzzers, such as StateAFL [3] and NSFuzz [4], are also built on AFLNet but have not modified their seed scheduling method. The enhancements made by these tools are orthogonal to the enhancements introduced by SCFuzz. By comparing our experiments with AFLNet, we can assess the effectiveness of the seed scheduling method proposed in this paper in relation to existing fuzzers.

5.2. State Coverage (RQ1)

To evaluate the state space exploration capability of SCFuzz, we compared the number of states and state transitions triggered by different fuzzers within 24 h of fuzzing. Since SCFuzz enhances the seed scheduling method without altering the state representation method, it uses response codes from response messages as state representations, which is similar to AFLNet. To quantify the improvement of SCFuzz relative to the baseline, we reported the percentage improvement in state coverage achieved within 24 h (*Improv*) and the probability that a random campaign of SCFuzz outperforms a random campaign of the baseline ($\hat{A}_{12}$), which is measured by the Vargha–Delaney effect size [37].

Table 3 presents the experimental results on state coverage. It can be observed that SCFuzz triggers a greater number of states and state transitions compared to AFLNet. Specifically, SCFuzz achieves an average improvement of 17.10% in the number of states and 22.92% in state transitions within 24 h of fuzzing compared to AFLNet. For all tested protocol subjects, the Vargha–Delaney effect size ($\hat{A}_{12} \geq 0.6$) indicates that SCFuzz demonstrates a clear advantage over AFLNet in exploring the state space.

The improvements of SCFuzz on LightFTP are not significant. Specifically, the average increase in the number of states is 4.35%, while the increase in state transitions is 4.90%. Analysis shows that LightFTP is a lightweight program based on the FTP protocol, with relatively simple functionality, which limits the effectiveness of SCFuzz on this program. Similarly, SCFuzz does not perform well on Forked-daapd, where state transitions only increased by 4.55%. Upon analysis, we found that the main reason is the slow execution speed of this protocol program, which hinders the full potential of SCFuzz.

In conclusion, SCFuzz achieves greater state coverage than the baseline. Through deeper state space exploration, SCFuzz shows clear advantages in protocol fuzzing. It identifies more new states and transitions, improving the effectiveness and coverage of the fuzzing process.

Table 3. The average state coverage by different fuzzers.

Subject	AFLNet		SCFuzz					
	States	**Trans**	**States**	**Improv**	$\hat{A}_{12}$	**Trans**	**Improv**	$\hat{A}_{12}$
Live555	11.00	77.60	13.00	18.18%	1.00	99	27.58%	1.00
LightFTP	23.00	220.20	24.00	4.35%	0.72	231	4.90%	1.00
Pure-FTPD	26.20	202.40	29.00	10.69%	1.00	266	31.42%	1.00
ProFTPD	22.00	152.60	28.00	27.27%	1.00	223	46.13%	1.00
Forked-daapd	8.00	22.00	10.00	25.00%	1.00	23	4.55%	0.60
Average	-	-	-	17.10%	-	-	22.92%	-

5.3. Code Coverage (RQ2)

Code coverage has consistently been a key metric for evaluating fuzzing tools. Generally, higher code coverage indicates a more comprehensive test of the target program, increasing the likelihood of triggering program vulnerabilities. Table 4 presents the branch coverage, line coverage, and path coverage achieved by different fuzzers over five 24 h fuzzing sessions. To quantify the improvements of SCFuzz relative to the baseline, we report the percentage improvement (*Improv*) across various coverage metrics.

Table 4. The average code coverage by different fuzzers.

Subject	AFLNet			SCFuzz					
	Branch	**Line**	**Path**	**Branch**	**Improv**	**Line**	**Improv**	**Path**	**Improv**
Live555	2796.4	5716	852	2982	6.64%	6010	5.14%	908	6.57%
LightFTP	321.8	660.4	566	332	3.17%	696	5.39%	580	2.47%
Pure-FTPD	1027.6	1907.2	1254	1239	20.57%	2169	13.73%	1617	28.95%
ProFTPD	4893.2	10708	2299	5233	6.94%	11,464	7.06%	2730	18.75%
Forked-daapd	2294.2	7356.2	242	2347	2.30%	7482	1.71%	262	8.26%
Average	-	-	-	-	7.92%	-	6.61%	-	13.00%

The results in Table 4 show that SCFuzz achieves higher branch, line, and path coverage across all five protocol implementations compared to AFLNet. Specifically, SCFuzz achieves an average improvement of 7.92% in branch coverage, 6.61% in line coverage, and 13.00% in path coverage relative to AFLNet. The improvement rates across the three coverage metrics are generally consistent. Protocols that exhibit larger improvements in branch coverage also tend to show higher line and path coverage. For instance, in the Pure-FTPD protocol implementation, SCFuzz improves branch coverage by 20.57%, line coverage by 13.73%, and path coverage by 28.95%.

Figure 4 illustrates the branch coverage over time during the 24 h fuzzing process for both SCFuzz and AFLNet. As shown in the figure, SCFuzz and AFLNet exhibit similar code coverage performance in the early phases of testing. However, as time progresses, SCFuzz begins to demonstrate higher coverage, particularly in the later phases, where its branch coverage significantly surpasses that of AFLNet.

In conclusion, SCFuzz achieves higher code coverage compared to existing protocol fuzzers. Its seed scheduling method effectively selects seeds with the greatest potential to explore the code space, thereby accelerating code coverage during the testing process.

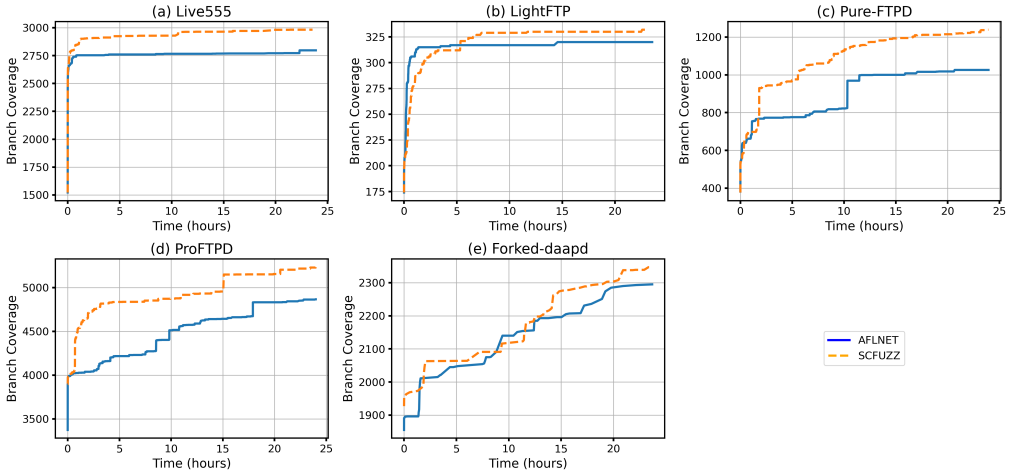

Figure 4. The branch coverage over time for different fuzzers during five 24 h fuzzing sessions.

5.4. Seed Selection Effectiveness (RQ3)

To evaluate whether the multi-phase seed selection strategy in SCFuzz can effectively select seeds with the greatest potential for rewards, we calculated the proportion of selected seeds that received rewards (i.e., discovered new code or state coverage) during the 24 h fuzzing process. This proportion is measured against the total number of seed selections to assess the effectiveness of the seed selection process. A higher rate of effective selection indicates that the fuzzer is more efficient in identifying seeds that can discover new states or code coverage, thereby enabling a deeper exploration of the target protocol implementation.

Table 5 presents the seed selection effectiveness of SCFuzz and AFLNet across different protocol implementations. It is evident that SCFuzz consistently outperforms AFLNet in terms of effective seed selection rate across all protocol implementations. On average, AFLNet achieves an effective selection rate of 9.86%, whereas SCFuzz achieves 27.45%, representing an improvement of 389.37%. In more complex protocol implementations like ProFTPD and Forked-daapd, SCFuzz reaches effective selection rates of 33.10% and 42.86%, respectively, which are significantly higher than AFLNet's 10.41% and 26.43%. This indicates that SCFuzz is more effective in selecting seeds with higher potential for discovering new state and code coverage, thereby improving the overall fuzzing efficiency. Even in the relatively simple LightFTP protocol implementation, SCFuzz achieves an effective selection rate of 29.73%, significantly surpassing the 3.11% rate achieved by AFLNet, further demonstrating the advantage of SCFuzz in handling lightweight protocol implementations.

Table 5. The proportion of selected seeds receiving rewards in different fuzzers.

Fuzzer	Live555	LightFTP	Pure-FTPD	ProFTPD	Forked-daapd	Average
AFLNet	1.75%	3.11%	7.61%	10.41%	26.43%	9.86%
SCFuzz	13.56%	29.73%	18.01%	33.10%	42.86%	27.45%
Improv	674.11%	855.95%	136.66%	217.96%	62.16%	389.37%

In conclusion, the multi-phase seed selection strategy of SCFuzz demonstrates significant advantages in selecting effective seeds. Through this strategy, SCFuzz is able to efficiently identify and select seeds with greater potential during each fuzzing iteration, and these seeds exhibit superior performance in discovering new code or state coverage. This seed selection mechanism not only improves the efficiency of the fuzzing process but also facilitates a deeper exploration of the protocol implementation.

5.5. Power Utilization Effectiveness (RQ4)

To evaluate whether the power allocation strategy based on state weights in SCFuzz improves power utilization effectiveness in fuzzing, we calculated the proportion of power used to successfully trigger new code or state coverage during the 24 h fuzzing process. A higher power utilization effectiveness indicates that the fuzzer is able to allocate resources more effectively, ensuring that the power is directed toward seeds with the highest potential for discovering new code paths and state transitions, thereby maximizing the overall effectiveness of the fuzzing process.

Table 6 presents the power utilization effectiveness of SCFuzz and AFLNet across different protocol implementations. The experimental results show that SCFuzz exhibits a higher power utilization effectiveness in all protocol implementations. On average, AFLNet achieves an average power utilization effectiveness of 0.37%, while SCFuzz achieves 0.54%, representing a 45.61% improvement over AFLNet. Specifically, in the ProFTPD protocol implementation, SCFuzz reaches a power utilization effectiveness of 1.45% compared to the 0.99% achieved by AFLNet. This improvement demonstrates that the power allocation strategy based on state weights of SCFuzz allows for a more efficient distribution of power, allocating more resources to the messages within seeds that have higher potential, and applying more mutations to these messages, resulting in the discovery of new code or state coverage.

Table 6. The effectiveness of power utilization across different fuzzers.

Fuzzer	Live555	LightFTP	Pure-FTPD	ProFTPD	Forked-Daapd	Average
AFLNet	0.10%	0.22%	0.24%	0.99%	0.29%	0.37%
SCFuzz	0.14%	0.35%	0.38%	1.45%	0.36%	0.54%
Improv	40.00%	59.09%	58.33%	46.46%	24.14%	45.61%

In conclusion, the power allocation strategy based on state weights of SCFuzz improves power utilization, allowing the fuzzer to explore code and state spaces more efficiently under limited power resources, thereby enhancing the overall test coverage.

6. Discussion

Although SCFuzz enhances the effectiveness of seed scheduling and improves the overall efficiency of fuzzing, it still has certain limitations.

First, in dividing the fuzzing process into initial, middle, and final phases, we currently define the duration of each phase as fixed values set at the start of the fuzzing. While this phase division generally aligns with common fuzzing dynamics, it may not be universally applicable across all protocol implementations. Fuzzing dynamics can vary significantly between different protocols, and a fixed-phase duration model may not capture the nuances of each fuzzing session. SCFuzz does not yet have the ability to adapt phase durations based on real-time feedback, which could better reflect the actual progress of fuzzing. Developing this adaptive capability represents a promising direction for future research. In particular, the real-time monitoring of state discovery and transition rates could be leveraged to adjust phase durations dynamically, ensuring a more efficient use of resources.

Secondly, seed scheduling and seed mutation are two core factors that influence the effectiveness of fuzzing. Seed scheduling determines the direction of test selection, while seed mutation impacts the validity and quality of the generated test cases. These two factors are interdependent—optimal seed selection alone cannot fully exploit the potential of seeds without effective mutation methods. However, most existing studies optimize seed scheduling and mutation independently, which limits the generalizability and scalability of their approaches. In SCFuzz, although we focus on improving seed scheduling, we acknowledge that the performance of the seed selection strategy could be further improved if coupled with a more sophisticated mutation strategy. The current lack of synergy between

seed scheduling and mutation techniques is a key limitation. Therefore, exploring a more integrated approach that combines both elements—scheduling and mutation—could be a breakthrough for fuzzing performance. This combined strategy could consider the characteristics of individual seeds, adjusting not only the selection order but also the mutation approach applied to each seed. Further research into adaptive mutation strategies, potentially using machine learning or reinforcement learning techniques, could help tailor mutations based on the specific states triggered by seeds.

In summary, although SCFuzz has made significant progress compared to existing fuzzing tools, there are still several areas that can be further improved. Future research should focus on the design of adaptive phase management mechanisms and the deeper integration of seed scheduling and mutation strategies. These improvements will not only enhance the generalizability and performance of SCFuzz but also promote the application of fuzzing tools in more complex and diverse testing environments. Addressing these challenges will lay a solid foundation for developing more efficient and precise fuzzing tools, thereby advancing the widespread adoption of fuzzing technology in practical applications.

7. Conclusions

In this paper, we present SCFuzz, which is an innovative approach designed to optimize seed scheduling in network protocol fuzzing. By conceptualizing seed selection as a multi-armed bandit problem and partitioning the fuzzing process into distinct phases, SCFuzz dynamically balances exploration and exploitation, tailoring its approach to the unique requirements of each phase. Additionally, a power allocation strategy based on state weights ensures focused resource distribution to high-potential messages, thereby improving both the coverage and efficiency of the fuzzing process. The scientific contribution of this study is rooted in its novel integration of multi-phase seed selection with a dynamic exploration/exploitation trade-off and state-weighted power allocation. This integrated methodology significantly enhances fuzzing coverage and overall testing efficiency, addressing key challenges in the field of protocol fuzzing.

Experimental results substantiate the effectiveness of SCFuzz, demonstrating its superior performance compared to AFLNet across several widely used open-source protocol implementations. Specifically, SCFuzz achieves up to 17.10% more states discovered, 22.92% more state transitions, and 7.92% higher code branch coverage. Additionally, seed selection effectiveness is improved by 389.37%, while power utilization increases by 45.61%, thereby addressing the research questions formulated in this study. Beyond its technical contributions, this work holds societal implications by strengthening the security of communication protocol implementations. Through improved fuzzing efficiency, SCFuzz enables an earlier detection of vulnerabilities, mitigating the risks posed by security flaws in critical systems.

In conclusion, SCFuzz offers an effective solution to the challenges of protocol fuzzing, contributing to both the scientific understanding of fuzzing strategies and the practical security of communication protocols. Future work may explore adaptive phase management and the integration of seed scheduling with mutation strategies to further refine and enhance the performance of fuzzing techniques.

Author Contributions: Conceptualization, M.C., K.Z. and Y.L.; Formal analysis, Y.C. and M.C.; Funding acquisition, Y.L.; Investigation, M.C., K.Z., Y.C. and Y.L.; Methodology, M.C., K.Z. and Y.C.; Resources, M.C., K.Z. and Y.C.; Supervision, C.C. and J.Y.; Validation, M.C. and K.Z.; Visualization, C.C. and J.Y.; Writing—original draft, M.C.; Writing—review and editing, M.C., K.Z. and Y.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data that support the findings of this study are available from the corresponding author upon reasonable request.

Conflicts of Interest: All the authors declare that they have no conflicts of interest.

References

1. Manès, V.J.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2312–2331. [CrossRef]
2. Wu, W.C.; Nongpoh, B.; Nour, M.; Marcozzi, M.; Bardin, S.; Hauser, C. Fine-grained coverage-based fuzzing. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–41. [CrossRef]
3. Natella, R. Stateafl: Greybox fuzzing for stateful network servers. *Empir. Softw. Eng.* **2022**, *27*, 191. [CrossRef]
4. Qin, S.; Hu, F.; Ma, Z.; Zhao, B.; Yin, T.; Zhang, C. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Trans. Softw. Eng. Methodol.* **2023**, *32*, 1–26. [CrossRef]
5. Ba, J.; Böhme, M.; Mirzamomen, Z.; Roychoudhury, A. Stateful greybox fuzzing. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 3255–3272.
6. Natella, R.; Pham, V.T. Profuzzbench: A benchmark for stateful protocol fuzzing. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 11–17 July 2021; pp. 662–665.
7. Slivkins, A. Introduction to multi-armed bandits. *Found. Trends Mach. Learn.* **2019**, *12*, 1–286. [CrossRef]
8. Auer, P.; Cesa-Bianchi, N.; Freund, Y.; Schapire, R.E. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In Proceedings of the IEEE 36th Annual Foundations of Computer Science, Milwaukee, WI, USA, 23–25 October 1995; pp. 322–331.
9. Banks, G.; Cova, M.; Felmetsger, V.; Almeroth, K.; Kemmerer, R.; Vigna, G. SNOOZE: Toward a Stateful Network protocol fuzzer. In Proceedings of the Information Security: 9th International Conference, ISC 2006, Samos Island, Greece, 30 August–2 September 2006; Proceedings 9; Springer: Berlin/Heidelberg, Germany, 2006; pp. 343–358.
10. Miki, H.; Setou, M.; Kaneshiro, K.; Hirokawa, N. All kinesin superfamily protein, KIF, genes in mouse and human. *Proc. Natl. Acad. Sci. USA* **2001**, *98*, 7004–7011. [CrossRef] [PubMed]
11. Gorbunov, S.; Rosenbloom, A. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS* **2010**, *10*, 239.
12. Peng, H.; Shoshitaishvili, Y.; Payer, M. T-Fuzz: Fuzzing by program transformation. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 697–710.
13. De Ruiter, J.; Poll, E. Protocol state fuzzing of TLS implementations. In Proceedings of the 24th USENIX Security Symposium (USENIX Security 15), Washington, DC, USA, 12–14 August 2015; pp. 193–206.
14. Eddington, M. Peach Fuzzer: A Smart Fuzzer for Complex Inputs. Available online: https://www.peachfuzzer.com (accessed on 6 October 2024).
15. Aitel, D. *The Advantages of Block-Based Protocol Analysis for Security Testing*; Immunity Inc.: Miami, FL, USA, 2002; pp. 105–106.
16. Meng, R.; Mirchev, M.; Böhme, M.; Roychoudhury, A. Large language model guided protocol fuzzing. In Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 26 February–1 March 2024.
17. Hu, Z.; Pan, Z. A systematic review of network protocol fuzzing techniques. In Proceedings of the 2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Chongqing, China, 18–20 June 2021; Volume 4, pp. 1000–1005.
18. Zeng, Y.; Lin, M.; Guo, S.; Shen, Y.; Cui, T.; Wu, T.; Zheng, Q.; Wang, Q. Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols. *Sensors* **2020**, *20*, 5194. [CrossRef] [PubMed]
19. Zardus. Preeny Repository. Available online: https://github.com/zardus/preeny (accessed on 6 October 2024).
20. Pham, V.T.; Böhme, M.; Roychoudhury, A. AFLNet: A greybox fuzzer for network protocols. In Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, 24–28 October 2020; pp. 460–465.
21. Zalewski, M. American Fuzzy Lop. Available online: https://lcamtuf.coredump.cx/afl/ (accessed on 6 October 2024).
22. Yu, Y.; Chen, Z.; Gan, S.; Wang, X. SGPFuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access* **2020**, *8*, 198668–198678. [CrossRef]
23. Andronidis, A.; Cadar, C. Snapfuzz: High-throughput fuzzing of network applications. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 18–22 July 2022; pp. 340–351.
24. Liu, D.; Pham, V.T.; Ernst, G.; Murray, T.; Rubinstein, B.I. State selection algorithms and their impact on the performance of stateful network protocol fuzzing. In Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Honolulu, HI, USA, 15–18 March 2022; pp. 720–730.
25. Browne, C.B.; Powley, E.; Whitehouse, D.; Lucas, S.M.; Cowling, P.I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; Colton, S. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intell. Games* **2012**, *4*, 1–43. [CrossRef]
26. Borcherding, A.; Giraud, M.; Fitzgerald, I.; Beyerer, J. The Bandit's States: Modeling State Selection for Stateful Network Fuzzing as Multi-armed Bandit Problem. In Proceedings of the 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS & PW), Delft, The Netherlands, 3–7 July 2023; pp. 345–350.
27. Wang, J.; Song, C.; Yin, H. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In Proceedings of the Network and Distributed Systems Security Symposium, Virtual, 21–25 February 2021.
28. Yue, T.; Wang, P.; Tang, Y.; Wang, E.; Yu, B.; Lu, K.; Zhou, X. EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Berkeley, CA, USA, 12–14 August 2020; pp. 2307–2324.
29. Zhao, L.; Duan, Y.; Xuan, J. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 24–27 February 2019.

30. Zhao, Y.; Wang, X.; Zhao, L.; Cheng, Y.; Yin, H. Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing. In Proceedings of the 38th Annual Computer Security Applications Conference, Austin, TX, USA, 5–9 December 2022; pp. 534–547.
31. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1032–1043.
32. Zhang, K.; Xiao, X.; Zhu, X.; Sun, R.; Xue, M.; Wen, S. Path transitions tell more: Optimizing fuzzing schedules via runtime program states. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022; pp. 1658–1668.
33. Lemieux, C.; Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 475–485.
34. Li, Y.; Xue, Y.; Chen, H.; Wu, X.; Zhang, C.; Xie, X.; Wang, H.; Liu, Y. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 533–544.
35. Kim, S.; Kim, Y. K-Scheduler: Dynamic intra-SM multitasking management with execution profiles on GPUs. *Clust. Comput.* **2022**, *25*, 597–617. [CrossRef]
36. Huang, H.; Chiu, H.C.; Shi, Q.; Yao, P.; Zhang, C. Balance seed scheduling via monte carlo planning. *IEEE Trans. Dependable Secur. Comput.* **2023**, *21*, 1469–1483. [CrossRef]
37. Arcuri, A.; Briand, L. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Testing Verif. Reliab.* **2014**, *24*, 219–250. [CrossRef]