Fuzzing:Multi-objective optimization

Zhentao Zhu, Zhenyu Wen,

***Abstract*—Protocol implementations are critical to network communication systems, and their security directly impacts overall system integrity. Fuzzing has become a widely adopted technique for vulnerability detection in these implementations. However, existing fuzzing strategies for stateful protocols often favor short interaction sequences and rely on simplistic evaluation metrics, resulting in shallow state exploration and missed vulnerabilities in complex transition paths. These limitations hinder comprehensive protocol testing by failing to leverage the semantic richness of diverse state transitions.**

In this paper, we present ObjFuzz, a grey-box fuzzing tool designed for multi-objective optimization, aiming to enhance both execution efficiency and vulnerability discovery capability. ObjFuzz treats state-seed sequences as the fundamental execution unit and employs a multi-armed bandit-based objective selection algorithm to effectively balance execution efficiency with vulnerability discovery capability, including both crash triggering and path exploration, thereby enhancing the overall effectiveness of protocol implementation testing. In addition, ObjFuzz integrates a state-guided mutation mechanism and a message-aware module. By leveraging runtime feedback, it dynamically constructs a finite state machine to enable deeper exploration of protocol implementations and improve the generation of valid test cases. Experimental results demonstrate that, compared to mainstream network protocol fuzzing tools, ObjFuzz increases the number of discovered execution paths by more than 15% on average and improves state transition coverage by over 10%, providing a more comprehensive and efficient security assessment of protocol implementations.

***Index Terms*—Grey-box Fuzzing, Protocol Security, Multi-objective optimization, State-guided Mutation.**

I. Introduction

NETWORK communication protocols define the rules for interactions between devices on the Internet, and the security of their software implementations plays a critical role in ensuring system stability and reliability. However, flaws in protocol implementations can pose significant security risks. For example, the widely adopted TLS protocol once suffered from the Heartbleed vulnerability in its OpenSSL implementation, allowing attackers to extract sensitive information from servers[1]. Similarly, security flaws in Microsoft's SMB protocol were exploited by the WannaCry ransomware, causing a large-scale global outbreak that compromised millions of computer systems[2].

Yachen Wang, Yejian Zhou and Huayong Tang are with College of Information Engineering, Zhejiang University of Technology, Hangzhou, 310023, P.R.China.

Guanyong Wang is with the School of Information and Science, North China University of Technology, Beijing 100144, China.

Lei Zhang is with the School of Electronics and Communication Engineering, Sun Yat-sen University, Shenzhen 518107, China.

The paper is supported by National Natural Science Foundation of China (Grant No. 62471438 and 62401018), the Zhejiang Provincial Natural Science Foundation of China (Grant No. LY23F010012). (Corresponding author: Yejian Zhou, email: yjzhou25@zjut.edu.cn;)

In recent years, grey-box fuzzing has emerged as a mainstream technique for vulnerability discovery due to its high efficiency. By mutating existing inputs to generate new test cases, grey-box fuzzing achieves rapid bug detection while maintaining a low false positive rate. Tools based on this technique have demonstrated remarkable success across both software and hardware security testing domains [3]. However, when applied to the testing of protocol implementations, traditional grey-box fuzzing still faces notable challenges. Unlike conventional programs that process inputs from local files, protocol implementations typically interact with clients via network interfaces and maintain complex internal states. As a result, fuzzers that lack protocol semantic awareness and rely solely on code coverage feedback often struggle to thoroughly explore the protocol state space, limiting their overall testing effectiveness.

To overcome these limitations, several grey-box fuzzing tools have been proposed specifically for protocol implementations. AFLNET [4], the first grey-box protocol fuzzer, builds a state model by extracting status codes from response messages to guide the fuzzing process, which improves the exploration of protocol state transitions. However, the large volume of network interactions significantly limits its overall testing efficiency. To address this, SNPSFuzzer [5] extends AFLNET by introducing a process-level snapshot mechanism that allows the program to quickly resume from specific states, thereby reducing network overhead and improving fuzzing throughput. NSFUZZ [6] further optimizes testing efficiency by instrumenting state variables to track server states and reconstructing the network interface logic, which helps eliminate unnecessary wait times and enhances resource utilization during fuzzing.

Current research on grey-box protocol fuzzing tools primarily focuses on improving interaction speed and expanding state coverage. However, several critical challenges remain to be addressed: (1) **In terms of evaluation metrics:** existing grey-box fuzzers exhibit inconsistent optimization objectives—state selection typically prioritizes code coverage to enhance crash detection, while seed selection tends to favor structurally simple sequences to increase execution speed, thereby overlooking critical interaction information embedded in runtime state transitions. This fragmented optimization strategy makes it difficult for fuzzers to achieve a balanced trade-off between execution efficiency and vulnerability discovery capability, ultimately compromising overall testing effectiveness. (2) **In terms of test case generation:** most fuzzers rely on message mutation techniques to produce new test inputs. However, such approaches often generate a large number of test cases that violate protocol format specifications, which the target server fails to parse correctly, resulting in substantial waste of fuzzing resources.

To address the limitations of existing grey-box protocol fuzzing tools—namely, the lack of multi-objective optimization and insufficient exploration of deep program states—this paper proposes **ObjFuzz**, a state-guided protocol fuzzing approach. ObjFuzz treats each *state-seed sequence* as a fundamental execution unit and simultaneously considers execution efficiency and vulnerability discovery capability as optimization objectives. To effectively schedule among numerous state-seed combinations, ObjFuzz employs a multi-armed bandit model to dynamically balance competing objectives, thereby improving resource utilization and enhancing vulnerability detection. To strengthen protocol semantic awareness, ObjFuzz constructs a state transition guidance dictionary based on the target protocol's RFC documentation, which provides structured transition information to guide the fuzzing strategy. Additionally, we design a state-guided mutation module to improve the exploration of deep state spaces, and introduce a message content awareness module to validate the structural correctness of mutated test cases. This reduces the overhead caused by invalid inputs and further enhances the efficiency and accuracy of the fuzzing process. Finally, we integrate the proposed modules into an existing fuzzing framework to build an optimized testing system—ObjFuzz. By incorporating multi-objective optimization and strategically guided state-aware testing, ObjFuzz significantly improves the effectiveness of the fuzzing process.

In summary, our contributions are summarised as follows:

- We propose ObjFuzz, a state-guided protocol fuzzing method that addresses the lack of multi-objective optimization and weak state exploration in existing greybox fuzzers. ObjFuzz treats state-seed sequences as basic units and uses a multi-armed bandit model to dynamically select and optimize them, balancing execution speed and vulnerability discovery to improve overall fuzzing effectiveness.
- We construct a protocol-specific state transition guidance dictionary based on RFC documentation, which encodes structured state evolution information. This dictionary serves as the semantic knowledge base for ObjFuzz, guiding testing strategies and enhancing the understanding of protocol behavior.
- We design a state-guided mutation module and a message content awareness module to improve deep state space exploration and test input validity. These modules ensure that generated test cases conform to protocol structure, effectively reducing invalid inputs and minimizing resource waste, thereby improving both the efficiency and accuracy of the fuzzing process.

II. Background and Motivation

In this section, we introduce the primary technical concepts of protocol fuzzing and clarify the main challenges we aim to address in this paper.

A. Protocol Fuzzing

To facilitate standardized communication across networked systems, the Internet Engineering Task Force (IETF) established technical specifications through Requests for Comments (RFCs). These documents define protocol architectures and operational requirements, as exemplified by the File Transfer Protocol (FTP) standardized in RFC 959. Protocol implementations rely on well-defined message structures and stateful interactions. As illustrated in Fig.1, FTP messages comprise three core components: a message command type, structured headers with key-value parameters, and carriage return and line feed characters (CRLF). Fig.2 further demonstrates the state machine governing protocol progression, where implementations transition from the INIT state to the AUTH state upon successful authentication using USER and PASS commands. Subsequent advancement to the TRAN state requires processing additional message types beyond basic authentication credentials.

Fuzz testing tools systematically generate message sequences to evaluate protocol robustness, ideally producing syntactically valid inputs that follow specified state transitions. Effective test case generation must account for both message structure validity and state-dependent sequencing constraints to accurately validate protocol conformance. This structured approach enables comprehensive verification of implementation adherence to RFC specifications while identifying potential edge-case vulnerabilities.

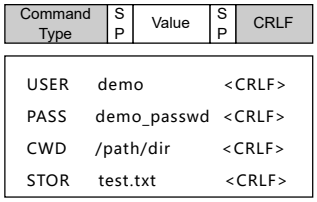

Fig. 1: FTP command structure and an example of FTP request from Lightftp.

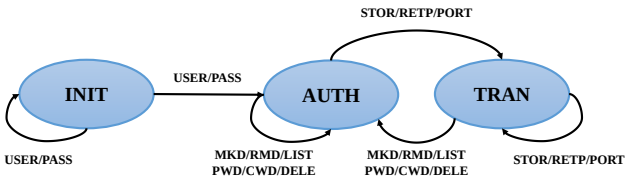

Fig. 2: FTP state model.

B. Task Scheduling

The protocol fuzzing framework (Fig.3) employs a modular architecture with three core phases: **test case generation, execution feedback, and vulnerability feedback.** The test case generation phase, as the central module, constructs specification-driven inputs through a three-tier mechanism: 1) Defining protocol state transitions and message syntax

via communication modeling ; 2) Optimizing test sequence prioritization using task scheduling algorithms to target critical states and edge-case paths; 3) Generating RFC-compliant test cases that ensure syntactic validity and state-machine consistency. This phase achieves dynamic adaptation of protocol rules through co-evolution of seed libraries and input models.

The remaining phases leverage dynamic injection and anomaly detection for state monitoring and defect identification, forming a closed-loop validation system. The framework ensures systematic exploration of protocol implementations through its layered, specification-aware design.

This study focuses on the task scheduling algorithm, which adopts a hierarchical mechanism to divide the fuzzing process into two phases: (1) Inter-state scheduling, which selects protocol states for testing based on state priority metrics; and (2) Intra-state scheduling, which generates targeted test cases within selected states using conventional methods such as seed scheduling, byte scheduling, and mutation strategy scheduling. This architecture enables fine-grained control through three heuristic strategies: Rarity-preferred[19, 110, 123, 162], which prioritizes less-explored states to uncover potential logic vulnerabilities; Performance-preferred[30, 55, 56, 110, 123], which emphasizes states with high code coverage or defect discovery rates; and Complexity-preferred[45, 57], which favors structurally complex or deep states (e.g., ICS3Fuzzer[45] prioritizes deep states, and Pulsar [57] uses mutable field weight calculations).

However, since these state selection algorithms are often implemented and evaluated independently across different platforms and targets, it is difficult to conduct fair comparisons or draw consistent conclusions. Liu et al.[85] evaluated three existing state selection algorithms in AFLNET[123], including a rarity-preferred strategy, a random selection strategy, and a sequential selection strategy. They found that these algorithms performed similarly in terms of code coverage, attributing this to AFLNET's coarse-grained state abstraction and inaccurate estimation of state productivity.

TABLE I: FTP state transition out-degree and in-degree matrix diagram.

	INIT	**AUTH**	**TRAN**
INIT	2	1	0
AUTH	0	6	3
TRAN	0	6	3

C. Motivation

Existing fuzzing strategies for stateful protocol implementations exhibit critical limitations: (1) **Existing methods favor short sequences, overlooking the vulnerability discovery potential of complex paths.** Conventional approaches prioritize minimizing message interactions by favoring shorter viable sequences for protocol state transitions. While optimizing resource efficiency, this strategy overlooks the cascading effects of intermediate state transitions. For instance, in FTP implementations, reaching the TRAN state can be achieved through multiple command sequences (e.g., [PASS, USER, STOR], [PASS, USER, LIST, STOR], or extended sequences like [PASS, USER, CWD, LIST, MKD, STOR]). Though longer sequences incur higher overhead from auxiliary operations (e.g., directory creation via MKD), these operations fundamentally modify the server's internal state, enabling richer feedback for subsequent test inputs.

Furthermore, protocol state transitions exhibit heterogeneous complexity levels (as shown in Table I), where the AUTH-to-TRAN transition involves significantly more paths than INIT-to-AUTH transitions. This observation reveals the insufficiency of state coverage metrics alone for guiding effective testing. Current tools' neglect of intricate long-sequence patterns restricts their ability to exercise critical edge-case scenarios, thereby limiting deep-state vulnerability detection. To address this gap, this work proposes a state-aware testing methodology that systematically prioritizes path diversity over conventional state coverage metrics, thereby advancing vulnerability discovery in complex protocol implementations.

(2) **Existing grey-box fuzzing approaches for stateful network protocols exhibit significant limitations in their evaluation methodology.** While existing approaches acknowledge the importance of state selection and implement heuristic algorithms (e.g., RANDOM, ROUND-ROBIN, FAVOR), their oversimplified evaluation criteria inadequately capture the intrinsic value of state sequences. Current methods predominantly prioritize defect detection metrics at the expense of operational efficiency and favor minimal-path sequences rather than exploratory alternatives with higher diversity potential.To address these shortcomings, this study proposes a dual-objective optimization framework that simultaneously considers both execution time and bug discovery rate. Consequently, the current evaluation methodology exhibits limited capability in precisely quantifying the intrinsic value of discrete protocol states and their associated transition sequences, thereby adversely affecting the fuzzer's overall testing efficacy.

III. Method and Implementation

In this section, we first model the system using a finite-state machine and define our optimization objectives. Subsequently, we outline the details of the key components, including .

A. Problem Definition

In protocol fuzzing, state transitions often correlate with specific event occurrences. Finite State Machines (FSMs) provide a formal framework to characterize system behaviors by defining discrete states and their transition rules, making them particularly effective for modeling event-driven systems such as protocol implementations. By formalizing protocol logic into FSMs, explicit mappings between input sequences and state transitions can be established. These structured mappings guide fuzzing tools in generating context-aware test cases that probe deep program states and validate transition-path security. Key parameters used in this framework are summarized in Table II.

A Mealy FSM is a six-tuple:

$$\mathcal{A} = (Q, q_0, I, \Lambda, \delta, \lambda) \tag{1}$$

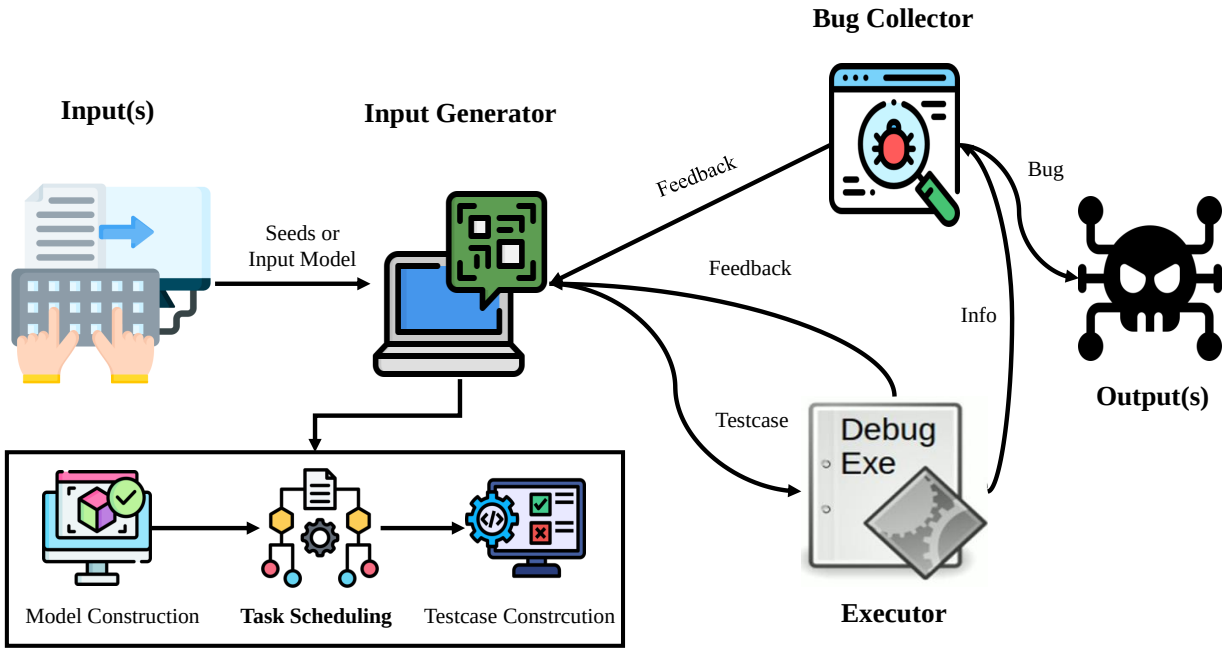

Fig. 3: Summarized workflow of existing protocol fuzzers.

TABLE II: Main Parameters

Parameter	Sign	Content
Finite Set of States	Q	$q_1, q_2, \ldots, q_n$
Initial State of The System	q_0	/
Input Alphabet	I	$\sigma_1, \sigma_2, \ldots, \sigma_m$
Output Alphabet	Λ	$o_1, o_2, \ldots, o_n$
State Transition Function	δ	$\delta(q, \sigma) = q'$
Set of Outputs for State Transition	λ	$Q \times I \rightarrow \Lambda$
State-seed Sequences	C	$C_1, C_2, \ldots, C_l$

where $Q = \{q_1, q_2, \ldots, q_n\}$ represents a finite set of states, q_0 represents the initial state of the system, $I = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ is the input alphabet, and $\Lambda = \{o_1, o_2, \ldots, o_n\}$ is the output alphabet; $\delta : Q \times I \rightarrow Q$ represents the state transition function and can be expressed as $\delta = \{(q, \sigma, q') : \delta(q, \sigma) = q'\}$. $\lambda : Q \times I \rightarrow \Lambda$ represents the set of potential outputs or observations for this transition.

Meanwhile, let I^* represents the set of finite-length sequences on I, and let Λ^* represents the state transition outputs over Λ. In this regard, by successive iterations, for any $q_0 \in Q$, δ can be extended over a k-length string $s_k = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_k} \in I^*$ by $\delta(q_0, s_k) := \delta(\delta(\ldots\delta(\delta(q_0, \sigma_{i_1}), \sigma_{i_2})\ldots), \sigma_{i_k})$. Also the output function λ can be extended over a k-length string $p_k = o_{j_1}o_{j_2}\ldots o_{j_k} \in \Lambda^*$ by $p_k = \lambda(q_0, s_k) := \lambda(\lambda(\ldots\lambda(\lambda(q_0, \sigma_{i_1}), \sigma_{i_2})\ldots), \sigma_{i_k})$.

Therefore, given a Mealy FSM $\mathcal{A}$, there exist different input-output pairs (Δ_i, O_j) composed of sequences $(\sigma_{i_1}, o_{j_1})(\sigma_{i_2}, o_{j_2})\ldots(\sigma_{i_k}, o_{j_k})$, where the state transition function $\delta(q_0, s_k)$ and the output function $\lambda(q_0, s_k)$ are defined over different existing sequences of events $s_k = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_k}$, ensuring that $|\delta(q_0, s_k)| > 0$ and $|\lambda(q_0, s_k)| > 0$, respectively.

For any protocol implementation, a corresponding Mealy FSM $\mathcal{A}$ can be constructed. During each round of fuzzing, the fuzzing tool $\mathcal{F}$ selects appropriate *state-seed sequence* $C = (q_i, s_k^j)$ for testing, where $q_i \in Q$ represents the target state to be tested in the current round. Let m_q denote the number of seed sequences (non-fixed and state-dependent) that can reach q_i, from which one seed sequence s_k^j is selected for testing.

Formally, when the protocol implementation triggers a crash or discovers a new code path during the execution of a given state-seed sequence C, the system records the corresponding crash type and coverage information. This process is iteratively performed until a predefined maximum execution time limit is reached.

The optimization objective of this study is to maximize the combined yield of unique crash discoveries and newly covered code paths within a limited time budget, thereby enhancing both the vulnerability detection capability and execution efficiency of fuzzing for complex stateful protocol implementations.

B. Optimization Goal

In the fuzzing process, our goal is to maximize both the number of unique crashes(M) and the number of newly discovered code paths(P) while ensuring that the total execution time does not exceed the predefined threshold $T_{\max}$. To evaluate these two key metrics in a unified manner, we normalize them into a single composite score function (Score). The final optimization objective is formally defined as:

$$\begin{aligned} &\arg\max Score \\ &\text{s.t.} \quad T \leq T_{\max} \end{aligned} \tag{2}$$

First, define the non-negative integer variable x_{q,s_k} to represent the number of times the state sequence pair (q, s_k)

is selected, where q represents a decision option from the candidate choice set Q, and s_k represents a seed sequence from the option q's corresponding seed sequence set m_q, with the constraint $x_{q,s_k} \in \mathbb{Z}^+$.

$$x_{q,s_k} \in \mathbb{Z}^+ \quad \forall q \in Q,\ s_k \in m_q \tag{3}$$

Score Model. In this section, we aim to model the score $Score$ obtained during the fuzzing process. We define $Score$ as the cumulative score generated across all rounds by each state-seed sequence C in the fuzzing process:

$$Score = \sum_{q=1}^{n} \sum_{s_k \in m_q} \sum_{j=1}^{x_{q,s_k}} \left(M_{q,s_k}^{j} + P_{q,s_k}^{j} \right) \tag{4}$$

Where M_{q,s_k}^{j} represents the number of unique crash types triggered in the j-th round of the experiment, and P_{q,s_k}^{j} represents the newly discovered code paths in the j-th round of the experiment.

Execution Time Model. We define the execution time T during testing as the sum of execution times for each state-seed sequence C:

$$T = \sum_{q=1}^{n} \sum_{s_k \in m_q} \sum_{j=1}^{x_{q,s_k}} t_{q,s_k}^{j} \tag{5}$$

Where t_{q,s_k}^{j} represents the cumulative execution time of each state-seed sequence in the j-th round.

Value Model. To accurately evaluate the exploration potential of each state-seed sequence and prioritize the selection of more valuable pairs in subsequent fuzz testing, thereby maximizing the model's score within limited execution time, we propose an innovative value assessment model E_{q,s_k}. This model quantitatively evaluates the exploration value of state-seed sequences, considering crash count, code paths, and execution time to optimize testing performance:

$$E_{q,s_k} = \sum_{j=1}^{x_{q,s_k}} E_{q,s_k}^{j} \tag{6}$$

where E_{q,s_k} represents the total exploration value summed across all $selected\_time$ experimental rounds for each state-seed sequence C, calculated as the accumulation of E_{q,s_k}^{j} values obtained in each round. The exploration value per experiment E_{q,s_k}^{j} is defined as:

$$E_{q,s_k}^{j} = f_{\text{score}}^{(j)}(q, s_k) \times f_{\text{time}}^{(j)}(q, s_k) \tag{7}$$

$$f_{\text{crash}}^{(j)}(q, s_k) = 2^{\log(P_{q,s_k}^{j} + M_{q,s_k}^{j} + \alpha_j \cdot M_{q,s_k}^{total} + \epsilon)} \tag{8}$$

$$\alpha_j = \frac{M_{q,s_k}^{total}}{P_{q,s_k}^{j} + M_{q,s_k}^{total} + \epsilon} \tag{9}$$

Our proposed state value evaluation formula consists of two key components. The first component is the score-based value model, formally defined in Equation (8). This metric is adapted from the state value estimation method used in AFLNET, with targeted improvements and extensions introduced to better suit our testing framework. This component incorporates the following three core factors: (1) Code path coverage (P_{q,s_k}^{j}): Higher path coverage indicates a greater likelihood of uncovering potential vulnerabilities, and thus this factor is assigned substantial weight in the overall value assessment. (2) Number of Unique Crash Types (M_{q,s_k}^{j}): This metric captures the number of distinct crash types triggered during the testing process. Since each unique crash type often corresponds to a potential independent vulnerability with high specificity and practical exploitability, it is assigned a higher weight in the overall value evaluation model to highlight its critical role in enhancing vulnerability discovery effectiveness. and (3) Total number of crashes (M_{q,s_k}^{total}): This metric represents all crash events triggered during the current round. Since each crash generally indicates the presence of a potential vulnerability, it still holds significant value in the overall evaluation. However, some of these crashes may be duplicates and thus contribute less to the discovery of new vulnerabilities. To address this, we introduce a weighting function α_j to appropriately reduce its influence. Here, α_j denotes the proportion of M_{q,s_k}^{total} relative to the total number of crashes and newly discovered code paths in the current round, and $epsilon$ is a smoothing factor (set to $epsilon = 1$) to avoid division by zero. The logarithmic relationship in this factor demonstrates a negative regulatory effect on exploration value as testing iterations increase, thereby effectively preventing excessive retesting of low-efficiency regions and improving the overall efficiency of fuzz testing.

$$f_{\text{time}}^{(j)}(q, s_k) = \beta_j \times \left(2 \log_{10} \left(\log_{10}(t_{q,s_k}^{j} + \epsilon) \right) \right)^{-1} \tag{10}$$

$$\beta_j = \beta_{\max} \times \left(1 - \sigma \left(\gamma \left(T_p - T_0 \right) h^{-1} \right) \right) \tag{11}$$

The second part quantifies execution efficiency of state sequence instance pairs, as formulated in Equation (10). In order to accurately characterize the dynamic variation of path coverage and crash discovery efficiency over time during the fuzzing process, this study introduces a time-weighted dynamic reward mechanism. Experimental observations reveal that the rate of new path discovery and crash detection per unit time is significantly higher during the initial phase of fuzzing, indicating a pronounced time-dependent feature. To address this phenomenon, we propose the incorporation of a time gain function, which provides progressive reward compensation for exploration activities in later stages, thereby effectively mitigating the issue of diminishing returns in fuzz testing. Specifically, we define a dynamic weighting factor β_p that evolves with the cumulative execution time, formally expressed in Equation (11).

In this formulation, $\sigma(x)$ represents the standard Sigmoid function. The parameter $\beta_{\max}$ is used to set the maximum reward value during the early stages, ensuring that the fuzzer actively explores new paths and crashes at the beginning of the testing process. T_p represents the cumulative execution time of the ongoing fuzzing process, which dynamically tracks the progress of the testing. T_0 represents the temporal inflection point at which the reward begins to decay, guiding the fuzzer

towards deeper exploration as the testing progresses. The conversion coefficient h = 60 min/h (minutes/hour) enables standardized temporal unit conversion, based on the SI definition where 1 h = 60 min, ensuring dimensional consistency in experimental calculations. The parameter γ controls the rate of reward decay, ensuring a smooth and gradual reduction in practice. Through this design, β_p maintains a high reward value in the early testing stages to accelerate the discovery of paths and crashes. As the testing progresses, the reward gradually converges, directing resources towards inputs with higher exploration potential. This dynamic adjustment mechanism effectively balances exploration and exploitation, significantly enhancing the efficiency of fuzz testing. These parameters will be optimized in subsequent experiments.

Meanwhile, to optimize the execution time of individual test cases, the model incorporates a time penalty factor with negative derivative properties (corresponding to the right-hand side of the equation). This design is theoretically grounded in two fundamental observations: prolonged execution time linearly reduces testing throughput, while execution efficiency demonstrates significant positive correlation with potential exploration value. The dual-regulation mechanism, integrating time-gain rewards and execution-time penalties, enables optimal resource allocation across both temporal and test-case dimensions through dynamic value assessment. This approach not only adheres to the fundamental exploration-exploitation trade-off principle in fuzz testing but also adaptively prioritizes high-potential test cases, thereby significantly improving overall testing efficiency. Where $t_{q,s_k}^{(p)}$ denotes the execution time of the current test cycle, whose value is determined based on the theoretical framework of AFLNet's state evaluation formula and subsequently validated through experimental studies.

C. MAB

This study employs the Multi-Armed Bandit (MAB) algorithm as the core optimization strategy to dynamically balance exploration and exploitation. By selecting the most promising state-seed sequence combinations from a pool of candidates, the algorithm aims to maximize the objective function. This approach enhances optimization efficiency while ensuring high-quality results. A comprehensive discussion of the MAB algorithm's theoretical basis and its relevance to this work will be presented in subsequent sections.

The Exploration-Exploitation Trade-off.

In the experiment, the system comprises n states, where each state q is associated with a set of reachable seed sequences m_q.

$$a_{i,j} \equiv C_l, \quad C_l \in \mathbf{C} \tag{12}$$

$$N = \sum_{q \in Q} |m_q| \tag{13}$$

Each state-seed sequence pair represents a fuzzing operation unit, i.e., a bandit arm $a_{i,j}$, resulting in a total of N distinct bandit arms, where N is the sum of reachable seed sequences across all states.

The reward $R(C_l, n_l)$ of each bandit arm $a_{i,j}$ is defined as E^{q,s_k}, representing its expected fuzzing effectiveness. In the algorithm design, the parameter n_l represents the cumulative selection count of state-seed sequence C_l (where $n_l \equiv x_{q,s_k}$, with x_{q,s_k} denoting the original counting variable). For notational simplicity, we consistently adopt n_l as the standardized representation throughout this paper.

$$R(C_l, n_l) = E_{q,s_k} = \sum_{p=1}^{n_l} E_{q,s_k}^{(p)} \tag{14}$$

Next, we can calculate a final score for the combinations to make decisions. UCB1 [27] is a classic answer to the MAB problem, and we calculate scores based on it as

$$\begin{aligned} \text{Score}(C_l, n_l) &= \overline{R}(C_l, n_l) + U(C_l, n_l) \\ &= \frac{\sum_{p=0}^{n_l} R(C_l, p)}{n_l} + \gamma \cdot \sqrt{\frac{\ln\left(\sum_{C_l \in C} n_l\right)}{n_l}} \end{aligned} \tag{15}$$

In the fuzzing framework, $\mathbf{C}$ denotes the set of all instance pairs. The scoring function $\text{Score}(C_l, n_l)$ consists of two components:

Where $\overline{R}(C_l, n_l)$ represents the average reward of pair C_l over n_l previous rounds and reflects historical performance for exploitation. $U(C_l, n_l)$ represents the upper confidence bound component. The algorithm begins with a **pioneering phase** that enforces $\forall C_l \in C, n_l = 1$. In subsequent rounds, the system dynamically selects:

$$C_{\text{next}} = \underset{C_i \in C}{\text{argmax}}\ \text{Score}(C_i, n_i) \tag{16}$$

The hyperparameter γ controls the exploration-exploitation trade-off.

D. Algorithm Design

1) Overview: To enhance algorithmic dynamism and mitigate errors induced by fuzzing randomness, we propose Fuzzobj, an optimized variant of UAB1, with the following workflow: The proposed state-guided protocol fuzzing framework takes four key inputs: the target protocol implementation Pt, an initial sequence s_{k0} containing the starting state q_0, dictionary corpus d, and the fuzzing depth D. The primary output is the total number of crashes M detected during testing.

The fuzzing process begins with an initialization and pioneer phase (Lines 1-2), where all state-seed sequences undergo preliminary testing. During the state-seed sequence pair selection phase, the fuzzer first computes the SWVA-UCB score for each state-seed sequence (Lines 4-5), then constructs a Top-K candidate pool based on these scores (Line 6). The BoltzmannSelect algorithm is subsequently employed to select the target state-seed sequence Cl from this pool (Line 7), effectively guiding the fuzzer's exploration of the target server. The energy allocation phase dynamically distributes fuzzing resources according to historical testing results and selection frequencies (Line 8). In the mutation testing phase, the framework applies different fuzzing strategies to mutate sequence s_{k0} based on both the protocol's program state and current fuzzing depth (Lines 10-14). Following mutation, each sequence undergoes validity verification through a content-aware module (Lines 15-16). Sequences causing protocol

specification violations are recorded as protocol errors (M_E) and immediately proceed to the next iteration. Valid sequences are sent to P_t for execution (Line 17). The fuzzer maintains two categories of mutated sequences s'_k : those that trigger crashes in P_t (Lines 18-19), and those that improve code or state coverage. For the latter case, the system correspondingly updates the Finite State Machine (Lines 20-22). This iterative process continues until the allocated fuzzing resources for the current round are exhausted, at which point a new state-seed sequence is selected for subsequent testing. The final output M represents the cumulative count of valid crashes detected throughout the testing campaign.

Algorithm 1 The Fuzzobj algorithm

Input : P_t: Protocol Implementation
s_{k0}: Initial Seed Queue with state q_0
d: Dictionary Corpus
D: Current fuzzing depth
Output: M: Crash sequence count

1 Initialize FSM S and Seed Queue s_{k0} with q_0;
2 **Pioneer Phase:** Each C_l is selected once to initialize all $n_l \leftarrow 1$; After each fuzzing iteration, compute $SWVA\text{-}UCB(C_l, n_l)$ to guide future selection;
3 **repeat**
4 **State-seed sequence Selection Phase:**
foreach $C_l \in C$ **do**
5 SWVA-UCB$(C_l, n_l) \leftarrow$ ComputeScore(C_l, n_l)
6 $\mathcal{K} \leftarrow$ GetTopKPairs(C, K) ; `// Rank by SWVA-UCB score`
7 $C_l^* \leftarrow$ BoltzmannSelect$(\mathcal{K})$
8 **Energy Allocation Phase:**
$E \leftarrow$ AssignEnergy(C_l^*)
9 **Mutation Testing Phase:**
for $i \leftarrow 1$ **to** E **do**
10 $INIT \leftarrow$ CalculateDepth(D, R)
11 **if** $NotINIT$ **then**
12 $s'_k \leftarrow$ StateGuidedMutate(s_k, d, q)
13 **else**
14 $s'_k \leftarrow$ RandomMutate(s_k, q)
15 **if** $ContextAware(s'_k, d)$ **then**
16 $M_E \leftarrow M_E \cup \{s'_k\}$
17 $R' \leftarrow$ SendToServer(P_t, s'_k)
18 **if** $IsCrash(s'_k, P_t)$ **then**
19 $M_N \leftarrow M_N \cup \{s'_k\}$
20 **else if** $IsInteresting(s'_k, P_t, q)$ **then**
21 $m_q \leftarrow m_q \cup \{s'_k\}$
22 $Q \leftarrow$ UpdateFSM(Q, R')
23 **until** *timeout* T ***or*** *abort-signal*;
24 **return** $M = M_E + M_N$

2) Detailed Techniques of ObjFuzz: **Dynamic Optimization of Evaluation Bias.** To address the evaluation bias arising from dynamic variations in path and crash rewards during fuzz testing, this paper introduces a dynamic optimization algorithm based on a sliding window mechanism. This mechanism limits the influence of historical data by retaining only the most recent W data samples, enabling the rapid forgetting of obsolete information and ensuring that the model more accurately reflects the reward distribution characteristics of the current testing phase.

Mitigating Reward Variance in Fuzzing Evaluation. Due to the inherent stochasticity of fuzz testing, even high-value state-seed sequences may yield relatively low crash or path rewards, while low-value pairs could conversely produce higher rewards. This randomness can significantly compromise the accuracy of test evaluation. To mitigate this issue, this paper proposes a variance estimation mechanism σ^2 for state-action pairs, effectively reducing the interference caused by random fluctuations during testing.

The estimators for both the mean and variance are defined as follows:

$$\hat{\mu}(C_l, n_l) = \frac{1}{\min(W, n_l)} \sum_{p=n_l-W+1}^{n_l} R(C_l, p) \tag{17}$$

$$\hat{\sigma}^2(C_l, n_l) = \frac{1}{\min(W, n_l) - 1}$$

$$\times \sum_{p=n_l-W+1}^{n_l} (R(C_l, p) - \hat{\mu}(C_l, n_l))^2 \tag{18}$$

Adaptive Exploration-Exploitation Balance. To address the issue of "premature convergence" in fuzzing, where the deterministic selection strategy of traditional UCB algorithms causes early fixation on suboptimal paths, neglecting continued exploration of potentially high-value regions, this paper proposes an enhanced exploration mechanism based on the Boltzmann distribution. This mechanism transforms the sliding-window variance-aware UCB score (SWVA-UCB) into an exponentially weighted probability distribution, with a time-dependent temperature parameter $\eta(t)$ to dynamically modulate the exploration intensity. Specifically, higher initial temperature values facilitate broad exploration during the early testing phase, while decreasing temperatures gradually shift the focus toward higher-reward actions. This relative action-value-based softmax strategy effectively combines two advantages: (1) overcoming the local optimum limitation inherent in greedy algorithms, and (2) mitigating the indiscriminate exploration characteristic of ϵ-greedy methods, thereby achieving an optimal adaptive balance between exploration and exploitation throughout the testing process.

The improved upper confidence bound (UCB) based on our modification is expressed as:

$$\text{SWVA-UCB}(C_l, n_l) = \hat{\mu}(C_l, n_l) + \sqrt{\frac{2\hat{\sigma}^2(C_l, n_l) \ln\left(\sum_{C_l \in C} n_l\right)}{n_l}} \tag{19}$$

We propose a two-phase selection strategy to optimize the exploration process: First, we select the top $K = \lceil \log N \rceil$ candidate arms based on their SWVA-UCB scores to form set $\mathcal{K}$, followed by Boltzmann selection within $\mathcal{K}$. This mechanism achieves dynamic exploration through adaptive temperature regulation, significantly improving computational efficiency while maintaining selection quality. The Boltzmann distribution probability generation, adaptive temperature function, and final Boltzmann selection are expressed by the following formulas:

$$P(C_l, n_l) = \frac{\exp\left(\eta(C_l, n_l) \cdot \text{SWVA-UCB}(C_l, n_l)\right)}{\sum_{k=1}^{N} \exp\left(\eta(C_l, n_l) \cdot \text{SWVA-UCB}(C_k, n_k)\right)} \tag{20}$$

$$\eta(C_l, n_l) = \frac{1}{\sqrt{\frac{1}{N}\sum_{k=1}^{N} \hat{\sigma}^2(C_k, n_k) + \epsilon}} \tag{21}$$

$$P(C_l, n_l) = \begin{cases} \frac{\exp(\eta(C_l, n_l)\,\text{SWVA-UCB}(C_l, n_l))}{\sum_{k \in \mathcal{K}} \exp(\eta(C_l, n_l)\,\text{SWVA-UCB}(C_k, n_k))}, & l \in \mathcal{K}, \\ 0 \qquad\qquad\qquad\qquad\qquad , & \text{otherwise.} \end{cases} \tag{22}$$

Specifically, the sampling probability for each candidate action is first calculated according to Equation(20), where $\eta(C_l, n_l)$ presents the adaptive temperature parameter for the action. As defined in Equation(21), $\eta(C_l, n_l)$ is dynamically adjusted based on the mean variance of the current action set, such that a larger variance results in a higher temperature to encourage exploration, while a smaller variance leads to a lower temperature to promote exploitation. Finally, according to Equation(22), the probabilistic selection is performed only within the candidate set $\mathcal{K}$, with the sampling probability for non-candidate actions set to zero.

By dynamically narrowing the selection space and incorporating adaptive temperature regulation, this approach effectively enhances exploration diversity while maintaining the convergence and efficiency of the testing process, thus providing a more stable and efficient exploration mechanism for subsequent fuzzing stages.

Content-aware Module. To address the issue of non-compliant mutations caused by the randomness of fuzzing, we design and implement a content-aware module that filters out invalid mutated samples, thereby improving testing efficiency without compromising the ability to discover optimal solutions. We begin by executing the protocol implementation and conducting multiple rounds of interaction with test endpoints. During these interactions, we capture and analyze the exchanged network packets, with a particular focus on fields related to protocol commands, protocol states, and abnormal behaviors. Based on the RFC specification of the protocol, we construct a mapping between protocol state transitions and interaction commands. This mapping is incorporated into the fuzzing framework as prior knowledge, serving as the decision basis for the content-aware module.

Specifically, the fuzzer drives state transitions in the protocol implementation by providing protocol commands (events) to uncover potential vulnerabilities. According to the finite state machine modeling results, all parsable protocol command types can be abstracted into an input set I. When the fuzzer mutates command types within I to generate new instances, if a mutated instance cannot be correctly interpreted by the protocol implementation (i.e., it falls outside I), it is not added to the message queue Q for transmission to the target server but is instead placed into the crash queue M_E. This mechanism effectively reduces the waste of fuzzing resources and enhances overall testing efficiency.

State-guided Mutation module. We designed a state transition dictionary d based on prior fuzzer knowledge. This dictionary stores message sequences that transition the protocol server from the current state A to the target state B. Additionally, the current program states Q and its depth in the state chain D as parameters for the fuzzer. We then adjust the fuzzing strategy based on two scenarios: (1) if the depth of the selected fuzzing state Q in the loop is less than the minimum initialization state length, and (2) if a state repeatedly appears in the state chain without causing a crash (e.g., the same state information appears at different depths in the state chain). When the fuzzing iteration encounters these situations, ObjFuzz invokes our state-guided mutation module. The module reads a message sequence s_d from d based on the current state Q, which transitions the target server to a new state Q' , forming a new sequence , described as follows:

$$\exists\ \delta(Q, s_d) \rightarrow Q' \quad s_d \in d \tag{23}$$

By triggering state changes in the target server, we expand the exploration scope of the fuzzing iterations, thereby enhancing the tool's ability to test deeper program state paths.

IV. Evaluation

Our evaluations aim to answer the following questions:

RQ1. Bug Detection Capability of ObjFuzz: How does ObjFuzz's bug detection performance compare with previous grey-box fuzzing results for network protocols?

RQ2. State Space Exploration Capability of ObjFuzz: Can ObjFuzz explore more state transitions than other methods?

RQ3. Fuzzing Capability of ObjFuzz: Can ObjFuzz explore a greater number of program execution paths?

To answer these questions, we follow the recommended experimental design for fuzzing experiments.

A. Configuration Parameters

To reduce the variance introduced by the inherent randomness of fuzzing, we conducted 24-hour experiments for each configuration and repeated each setup 10 times to ensure statistical significance. During evaluation, we collected performance data from experiments. For our proposed approach, **ObjFuzz**, all parameters were carefully selected based on empirical observations. Specifically, the time window width used for temporal reward estimation was set to 5, and the size of the adaptive seed pool was dynamically adjusted to $\log(N)$, where N denotes the total number of reachable state-seed pairs in the system. For the time-weighted reward function β_p, we adopted a decaying schedule with a maximum reward value $\beta_{\max} = 50$, an inflection point at $T_0 = 600$ to trigger the decay, and a slope control parameter $\gamma = 0.15$ to moderate the rate of decline. These configurations were derived through extensive pilot experiments and are further validated in our subsequent ablation studies.

B. Benchmark and Baselines

Benchmark. Our benchmark testing includes three network protocol implementations that cover three widely used protocols: RTSP, FTP, and SIP. These protocols span a wide range of applications, including streaming media, file transfer, and session control. For each protocol, we select implementations that are commonly used in practice, as vulnerabilities in these implementations can have significant consequences.

Baseline. We chose AFLNET, ICS3Fuzzer, and AFLNETLEGION as our benchmark tools. AFLNET is the first open-source, state-of-the-art, and widely utilized protocol fuzzer. ICS3Fuzzer extends AFLNET by prioritizing the selection of deeper program states and states that trigger the execution of more basic blocks, thereby improving the overall efficiency of state space exploration. AFLNETLEGION further builds upon AFLNET by introducing a variant of the Monte Carlo Tree Search (MCTS) algorithm to guide state selection and mutation. This design aims to address the limitations of traditional heuristics by systematically balancing exploration and exploitation during fuzzing. Other fuzzing tools either have limited performance or are not open-source, making them unsuitable for comparison with our selected benchmark tools.

C. Variables and Measures

Given that certain protocols (e.g., FTP) are inherently difficult to trigger observable crashes during fuzzing, using the number of crashes alone as a performance metric may not fully capture the effectiveness of a fuzzing approach. To provide a more comprehensive and objective evaluation of the performance of ObjFuzz compared to baseline fuzzers, we introduce three experimental variables to serve as core evaluation measures in our study.

Paths Discovered. The number of newly discovered execution paths is a key metric that reflects a fuzzer's capability to explore the input space and exercise different code branches. This variable captures the total number of unique program paths that were reached for the first time during fuzzing. It is particularly important for protocols with complex input constraints or large codebases, where path diversity correlates closely with fuzzing effectiveness. A higher value indicates broader coverage and stronger exploration ability.

State Transitions. For stateful network protocols, the number of unique state transitions triggered by the fuzzer serves as an indicator of its semantic awareness and state-space exploration capability. This variable measures how many distinct state-transition pairs were exercised throughout the fuzzing process, as observed from protocol-level interactions or server responses. It provides insights into whether the fuzzer can reach deeper protocol logic and uncover hidden state behaviors. A higher count signifies better depth in state-space traversal.

Observed crashes. Observed crashes remain a traditional and critical metric for evaluating the vulnerability discovery capability of a fuzzer. In our experiments, this variable refers to the total number of unique crash-triggering inputs identified during fuzzing. While crash counts alone may not fully reflect exploration depth or path diversity, they directly demonstrate the fuzzer's effectiveness in exposing program exceptions and security flaws. A larger number of observed crashes generally corresponds to stronger practical impact in bug finding.

D. Experimental Infrastructure

All our experiments are conducted on a machine equipped with an Intel(R) Core(TM) i5-13600KF CPU running at 3.50GHz, 32GB of main memory, and Ubuntu 18.04 LTS.

E. Experiment Results

1) *Observed crashes* : To answer RQ1, we conducted 24-hour testing sessions with 10 repetitions for each experiment, and collected all crash data generated by AFLNET, ICS3Fuzzer, and AFLNETLEGION.

Table II presents the number of crashes discovered by each fuzzer within the same testing time frame.

OBjFuzz results in more crashes in the same period. Notably, none of the three fuzzing tools triggered crashes in the FTP protocol implementation, which may be attributed to the relatively simple program states of these protocols. This further underscores the importance of exploring deeper program states for effective protocol security testing.

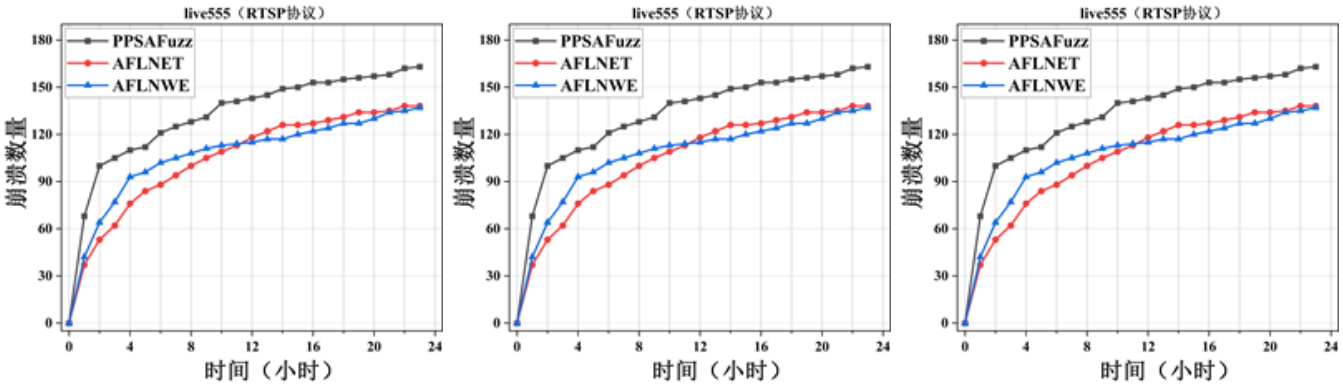

Image Image Image

Fig. 4: Visual comparisons of original models.

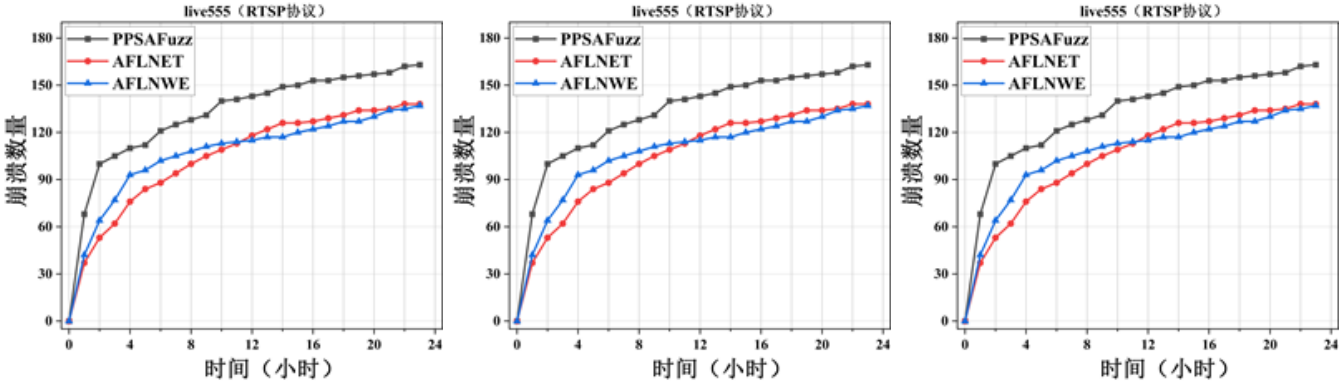

Image Image Image

Fig. 5: Visual comparisons of original models.

Table II
Average Number of Observed Crashes for Each Fuzzer

Subject	ObjFuzz	AFLNet	StateAFL	NSFuzz-V
Live555	167	140 (25.6%)	145 (17.3%)	152 (4.4%)
Kamailio	3	1 (12.1%)	2 (4.5%)	2 (0.7%)
AVG (IMP)	–	**16.3%**	**7.0%**	**0.7%**

Table III
Average Number of State Transitions for Each Fuzzer

Subject	ObjFuzz	AFLNet	StateAFL	NSFuzz-V
Live555	145	113 (25.6%)	118 (17.3%)	121 (4.4%)
LightFTP	343	316 (8.2%)	327 (1.8%)	336 (-2.9%)
Kamailio	138	123 (12.1%)	130 (4.5%)	132 (0.7%)
AVG (IMP)	–	**16.3%**	**7.0%**	**0.7%**

In Figure 5, subfigures (a) and (b) respectively illustrate the trends in the number of crashes triggered over time by the three fuzzers on the RTSP and DTLS protocol implementations under identical testing environments. The y-axis represents the cumulative number of crashes, while the x-axis denotes elapsed time. As shown in the figure, ObjFuzz exhibits a faster crash discovery rate on both protocols compared to the other two fuzzers. Moreover, it consistently discovers a significantly larger number of crashes within the 24-hour testing period. These results demonstrate the superior effectiveness of our approach in vulnerability detection.

2) *State Transition* :

Answer to RQ2: To answer RQ2, we conducted 24-hour testing sessions with 10 repetitions for each experiment, and collected all State Transitions generated by AFLNET, ICS3Fuzzer, and AFLNETLEGION.

Table III lists the number of state transitions detected by each fuzzer when testing the same three protocols. The results clearly demonstrate that SGMFuzz outperforms both AFLNET and NSFuzz and slightly exceeds Chatafl, with approximately 16% more state transitions detected than AFLNET, 7% more than NSFuzz, and 0.7% more than Chatafl. This finding aligns with SGMFuzz's ability to cover a broader range of program execution paths, indicating its superior capacity for exploring program state space.

Figures 6 (a), (b) and (c), respectively present the temporal evolution of state transition counts for three fuzzing tools when testing implementations of the RTSP, DTLS, and FTP protocols under identical experimental conditions. The vertical axis quantifies the cumulative number of state transitions, while

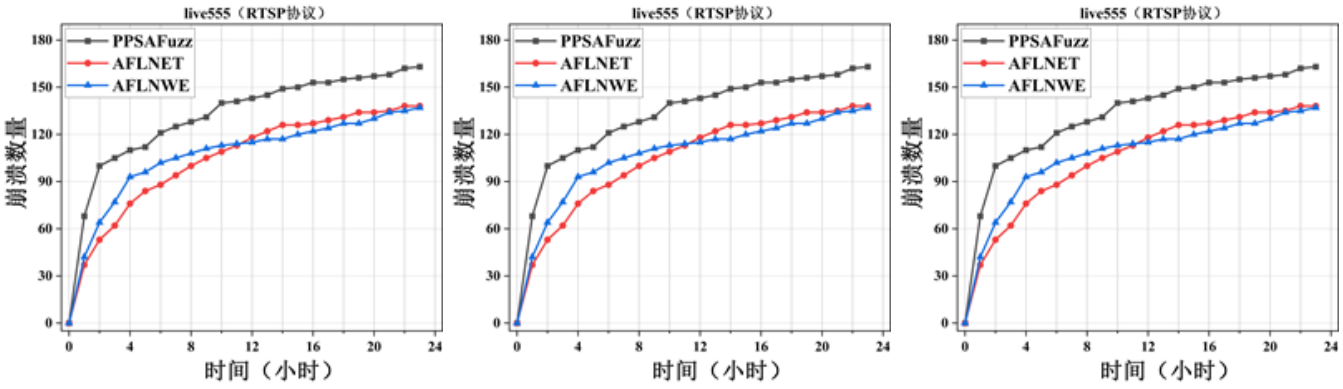

Image Image Image

Fig. 6: Visual comparisons of original models.

the horizontal axis represents testing duration. Experimental results demonstrate that ObjFuzz exhibits: (1) a 13.7% higher state transition discovery rate compared to baseline tools, and (2) an 11.2% increase in total transition path coverage (N = 10 experimental trials) during the 24-hour evaluation period. These findings statistically validate the superior vulnerability detection capability of our framework.

Table IV
Average Number of Execution Paths
for Each Fuzzer

Subject	ObjFuzz	AFLNet	StateAFL	NSFuzz-V
Live555	1460	1095 (33.3%)	1217 (14.1%)	1273 (14.1%)
LightFTP	781	635 (8.2%)	720 (1.8%)	763 (-2.9%)
Kamailio	4533	3978 (12.1%)	4263 (4.5%)	4378 (0.7%)
AVG (IMP)	–	**16.3%**	**7.0%**	**0.7%**

3) *Execution Path* :

Answer to RQ3: To answer RQ3, we conducted 24-hour testing sessions with 10 repetitions for each experiment, and collected all Execution Paths generated by AFLNET, ICS3Fuzzer, and AFLNETLEGION.

Table IV lists the number of state transitions detected by each fuzzer when testing the same three protocols. The results clearly demonstrate that SGMFuzz outperforms both AFLNET and NSFuzz and slightly exceeds Chatafl, with approximately 16% more state transitions detected than AFLNET, 7% more than NSFuzz, and 0.7% more than Chatafl. This finding aligns with SGMFuzz's ability to cover a broader range of program execution paths, indicating its superior capacity for exploring program state space.

Figures 7 (a), (b) and (c), respectively present the temporal evolution of execution paths for three fuzzing tools when testing implementations of the RTSP, DTLS, and FTP protocols under identical experimental conditions. The vertical axis quantifies the cumulative number of execution paths, while the horizontal axis represents testing duration. Experimental results demonstrate that ObjFuzz exhibits: (1) a 10.7% higher state transition discovery rate compared to baseline tools, and (2) an 10.2% increase in total transition path coverage (N = 10 experimental trials) during the 24-hour evaluation period. These findings statistically validate the superior vulnerability detection capability of our framework.

F. Key Parameter

Time window width. This study employs a controlled variable methodology to evaluate the impact of the time window parameter (w) on fuzzing effectiveness. The experimental protocol maintains constant environmental conditions while varying window sizes $w \in \{5, 10, 20\}$. We conducted 10 independent 24-hour test cycles on the RTSP protocol implementation, with crash count serving as the primary evaluation metric. Empirical results demonstrate that configuration w=10 achieves optimal testing performance, yielding 17.3% and 9.8% higher average crash discovery rates compared to w=5 and w=20 configurations respectively.

Table V
Average Number of Crashes
in Different Window Width

Subject	w = 5	w = 10	w = 20	w = 30
Live555	142	113 (25.6%)	121 (17.3%)	136 (4.4%)
LightFTP	342	316 (8.2%)	336 (1.8%)	347 (-2.9%)
Kamailio	142	123 (12.1%)	132 (4.5%)	140 (0.7%)
AVG (IMP)	–	**16.3%**	**7.0%**	**0.7%**

V. Conclusion

In this paper, we propose HEASSNet, a novel MTL framework designed to perform HE and SS using monocular optical RSI. Rather than prioritizing specific optimization strategies for HE and SS, we focus on simplifying the network structure and reducing design complexity. Specifically, HEASSNet adopts the pre-trained encoder of SAM, and we propose a novel LoRA fine-tuning strategy to make it more suitable for optical RSI. We design two independent branches to handle the distinct attributes of HE and SS tasks. To explore the potential correlation between HE and SS, we introduce the FEA-Gate, which incorporates long-range dependencies of SS into HE, effectively regularizing the HE process. Additionally, height maps are processed through a prompt encoder to obtain

height vectors, which guide the generation of the segmentation map in the SS branch. Experiments on multiple datasets demonstrate that HEASSNet achieves superior performance and compatibility. In future work, We will further explore dense and small object recognition to improve the performance of SS and HE tasks.

REFERENCES