SGMFuzz: State Guided Mutation Protocol Fuzzing

Zhenyu Wen, *Senior Member, IEEE*, Jianfeng Yu, Zening Huang, Yiming Wu, Zhen Hong, Rajiv Ranjan *Fellow, IEEE*

***Abstract*—Protocol implementations are fundamental components in network communication systems, and their security is crucial to the overall system. Fuzzing is one of the most popular techniques for detecting vulnerabilities and has been widely applied to the security evaluation of protocol implementations. However, due to the lack of machine-understandable prior knowledge and effective state-guided strategies, existing protocol fuzzing tools tailored for stateful network protocol implementations often suffer from shallow state coverage and generate numerous invalid test cases, thereby impacting the effectiveness of the testing process.**

In this paper, we introduce SGMFuzz, a grey-box fuzzing tool that combines a state-guided mutation mechanism to detect security vulnerabilities in protocol implementations. SGMFuzz uses the feedback collected during fuzzing to construct a finite-state machine, which aids in a deeper exploration of the program. Additionally, we design a message-aware module to enhance the tool's ability to generate valid test cases. Our evaluation demonstrates that, compared to the most advanced and widely used network protocol fuzzing tools, SGMFuzz increases the number of discovered execution paths by over 15% on average and improves state transition coverage by over 10%, providing a more comprehensive security assessment of protocol implementations.

***Index Terms*—Grey-box Fuzzing, Protocol Security, State-Guidance, Content-Awareness**

I. Introduction

Network communication protocols define how devices on the Internet interact with each other. However, flaws in the implementation of these protocols in software can lead to security vulnerabilities. For example, the Heartbleed bug in OpenSSL, a widely used implementation of the TLS protocol, allows attackers to steal sensitive information from servers [1]. Similarly, weaknesses in Microsoft's SMB protocol are exploited by the WannaCry ransomware attack that affected millions of computers around the world [2].

Grey-box fuzzing has become one of the most popular methods for vulnerability discovery, efficiently generating test cases through mutation-based techniques, characterized by rapid detection capabilities and low false positives. The tools developed based on this technique [3] have successfully uncovered numerous vulnerabilities in various software and hardware testing domains. However, traditional grey-box fuzzing tools face challenges when fuzzing protocol implementation programs, as these programs cannot directly read test cases from files and rely on network interfaces to receive client requests. Moreover, protocol implementation programs are stateful, requiring continuous updates to internal program states during client interaction. Consequently, traditional fuzzing tools that lack network interfaces and rely solely on code coverage metrics are not applicable for testing protocol implementation programs.

In recent years, several research efforts have emerged to develop grey-box fuzzing tools specifically designed for protocol implementations. AFLNET [4], the first grey-box protocol fuzzing tool, extracts status codes from protocol implementation response messages via network interface calls to track internal states and constructs a state model to guide the fuzzing process. However, due to the significant volume of network interactions during the fuzzing process, AFLNET's testing efficiency is greatly affected. Therefore, SNPSFuzzer [5], based on the AFLNET framework, attempts to introduce process-level snapshot technology to reduce the time spent handling network interaction messages during fuzzing tests. When conducting fuzzing tests on specific states, SNPSFuzzer directly restores the program to the corresponding state from the snapshot file, thereby enhancing fuzzing efficiency. NSFUZZ [6] detects server states through instrumentation statements of state variables and rebuilds network interfaces to optimize data processing, reducing ineffective waiting time during fuzzing and improving overall fuzzing efficiency.

Current research on grey-box protocol fuzzing tools primarily focuses on improving interaction speed and expanding state coverage. Yet several limitations remain (1) test case generation: recent fuzzing tools typically employ message mutation techniques to create new test cases. However, this approach often generates numerous instances that violate protocol structure specifications. The tested server cannot parse these instances, leading to a significant waste of fuzzing resources; (2) state transition coverage: most fuzzing tools aim to cover individual states rather than program state transitions. As a result, these tools prioritize simple test cases to expedite interaction speed, neglecting the interaction information embedded in runtime state transitions. Given that protocol implementations involve numerous state transitions, which may harbor errors, conducting a comprehensive security assessment of these transitions is crucial [7].

In this paper, we introduce SGMFuzz, a state-guided protocol fuzzing approach that utilizes a comprehensive understanding of protocol structures and their complex program state transitions to enhance the fuzzing process. First, we construct a state transition guidance dictionary based on RFC

This work was supported by the Key R&D Program of Zhejiang under Grant No.2024C03288, National Natural Science Foundation of China under Grants No.62302454 and 62302454, Zhejiang Provincial Science Fund for Distinguished Young Scholars under Grant LR24F020004, and Zhejiang Provincial Natural Science Foundation of Major Program (Youth Original Project) under Grant LDQ24F020001.

Zhenyu Wen, Jianfeng Yu, Zening Huang, Yiming Wu, Zhen Hong are with the Institute of Cyberspace Security, and College of Information Engineering, Zhejiang University of Technology, Hangzhou, Zhejiang, China (e-mail: wenluke427@gmail.com; {211122120007, 221122030311, wyiming, zhong1983}@zjut.edu.cn).

Rajiv Ranjan is with School of Computing, Newcastle University, NE4 5TG, UK (e-mail: raj.ranjan@newcastle.ac.uk).

documentation, specifically tailored to the relevant protocol. This dictionary serves as a knowledge base for the fuzzing tool. Second, we optimize existing fuzzing strategies and develop a state-guided mutation (SGM) module using this knowledge base, which improves the exploration of deep program states. Additionally, we introduce a message content awareness module to verify the legality of the mutation-generated test cases, minimizing unnecessary resource wastage during the fuzzing process. to enhance the exploration of deep program states. Finally, we refine the grey-box fuzzing algorithm and integrate these advanced modules into existing fuzzing tools to deliver our optimized solution—SGMFuzz. This integration results in a significantly enhance fuzzing tool capable of providing superior results through strategic, state-guided testing. *In summary*, this paper makes the following contributions:

- Our study takes a detailed look at the key characteristics of protocol implementations and explores the significant challenges they present for testing. At the same time, we examine the common weaknesses in current fuzzing methods and how these flaws negatively affect the overall efficiency of the fuzzing process.
- We have devised the enhanced SGMFuzz. This advanced tool includes both a message content-aware module and a state-guided mutation module, significantly improving the capability of traditional fuzzing tools to detect protocol vulnerabilities. SGMFuzz achieves this improvement by increasing the likelihood of generating test cases that adhere to protocol standards and by guiding the mutation process with great precision.
- We conducted a preliminary evaluation of SGMFuzz. The results indicate that, compared to other state-of-the-art grey-box network protocol fuzzers, SGMFuzz excels in path coverage, state space exploration, and vulnerability detection capabilities.

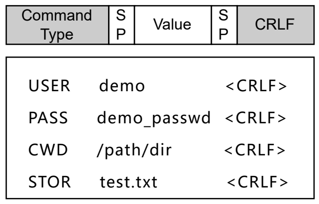
II. Motivation and Problem Definition

In this section, we introduce the primary technical concepts of protocol fuzzing and clarify the main challenges we aim to address in this paper. Subsequently, we model the system using a finite-state machine and define our optimization objectives.

A. Protocol Fuzzing

To ensure effective and reliable information sharing on the Internet, the Internet Engineering Task Force (IETF) and published as Requests for Comments (RFCs). For example, the File Transfer Protocol (FTP) is based on RFC 959. These protocols outline the general structure and sequence of message exchanges. As shown in Fig. 1, an FTP message consists of a message command type, key-value pairs, and carriage return and line feed characters (CRLF). The required sequence of FTP messages is depicted in Fig. 2: the protocol implementation begins in the INIT state to the AUTH state upon receiving USER and PASS-type messages. From the INIT state to the TRAN state, the protocol must receive at least one more specific type and structure of message besides the USER and PASS messages.

Fuzzing tools automatically generate message sequences and send them to the protocol implementation. Ideally, these message sequences should adhere to the required structure and order of the protocol. However, even the most advanced grey-box fuzzing tools face several challenges:



```
USER   demo          <CRLF>
PASS   demo_passwd   <CRLF>
CWD    /path/dir     <CRLF>
STOR   test.txt      <CRLF>
```

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

Challenge 1 (C1): Random mutations often fail to produce test cases that conform to the protocol message structure. Fuzzing tools generate new test cases by performing byte- or file-level mutations [3] on message sequences. Due to a lack of prior knowledge about the target protocol, these tools handle the contents of message sequences indiscriminately, mutating any part of the message command type, the command's key-value pairs, or the carriage return and line feed characters. For instance, consider the Lightftp interaction commands shown in Fig. 1. If a fuzzing tool mutates the command type of the third CWD command by reversing it to DWC and sending this mutated command to the protocol implementation, this mutation directly changes the command's semantics. Consequently, the protocol implementation will be unable to parse the command, disrupting subsequent protocol state transitions, impeding the fuzzing process, and wasting significant fuzzing resources.

Challenge 2 (C2): Short-sighted fuzzing strategies hinder the exploration of deep program state paths. The fundamental approach of fuzzing tools involves selecting a target protocol state for fuzzing and then choosing a message sequence from the queue that can reach this target state. In this scenario, multiple different test cases might lead the protocol implementation to the same program state. Specifically, as shown in Fig. 2, instances that bring the protocol implementation to the TRAN state can include different command groups such as [PASS, USER, STOR], [PASS, USER, LIST, STOR], or [PASS, USER, CWD, LIST, MKD, STOR]. Each additional command increases the fuzzing resources consumed by that command group. Consequently, to cover program states, most existing protocol fuzzing tools based on AFL will prefer the message sequence that reaches the target state faster, thereby reducing the number of message interactions to enhance fuzzing performance. However, protocol implementations are stateful software systems, and each valid command affects its state. In the example mentioned, the command group [PASS, USER, CWD, LIST, MKD, STOR] includes an extra MKD command, which creates an additional directory on the server. In subsequent fuzzing processes, when the fuzzing tool sends commands involving directory operations, it will obtain different results. For each type of message command, every directed edge from state A to state B increases the out-degree of state A and the in-degree of state B. As recorded in Fig. 3, the state transitions from AUTH to TRAN are much more extensive than those from INIT to AUTH, resulting in more state transition paths. Therefore, fuzzing strategies that only aim to cover states are suboptimal, as they discard complex instances and ultimately make it difficult to explore deep paths and uncover vulnerabilities.

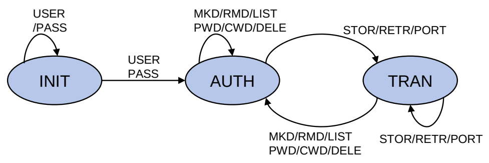
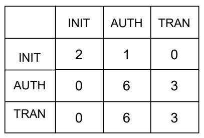


Fig. 2. FTP state model.



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

Fig. 3. FTP state transition out-degree and in-degree matrix diagram.

B. Problem Definition

During the fuzzing of protocols, each state transition may correspond to the occurrence of an event. Finite State Machines (FSMs) describe the potential behavior of a system through states and the transitions between them, making FSMs particularly suitable for modeling event-driven systems. By using FSMs to model protocol implementations, it is possible to construct mappings between inputs and protocol state transitions. This mapping aids fuzzing tools in generating inputs to check the security of deep program states and their transitions. The main parameters used in this section are listed in Table I.

A Mealy FSM is a six-tuple :

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

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

Meanwhile, let Σ^* (excluding ϵ) represents the set of finite-length sequences on Σ, 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 $s_k = \sigma_{i_1}\sigma_{i_2}\ldots\sigma_{i_k} \in \Sigma^*$ 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. Let F be the fuzzing tool that

TABLE I
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	Σ	$\sigma_1, \sigma_2, \ldots, \sigma_m$
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	$Q \times \Sigma \to \Lambda$
Generates instances	C	$c_1, c_2, \ldots, c_n$

generates instances $C = \{c_1, c_2, \ldots, c_n\}$. If the protocol implementation receives an instance that triggers a series of state transitions resulting in previously unobserved results O_j, it indicates that the instance has explored and covered a new input-output pair (Δ_i, O_j). We record instances that produce such results as $M = \{M_1, M_2, \ldots, M_N\}$. As mentioned above, the size of the set M can reflect the completeness of the test. To enhance the overall effectiveness of the fuzzing, we establish the following optimization objective:

$$\begin{aligned} \arg\max M_1, .., M_N \quad & \delta(q_0, M_n) \to O_n \quad \forall M_n \in \{M_1, .., M_N\} \\ \text{s.t.} \quad & \delta(q_0, M_n) \to O_n \neq \delta(q_0, M'_n) \to O'_n \\ & \forall M_n \in \mathbf{C} \end{aligned} \tag{2}$$

As described in Equation 2, during each round of fuzzing, each element in M represents a unique exploration path that leads the protocol implementation to produce previously unobserved results. Therefore, our objective is to maximize the set M. Additionally, the elements in M are generated by the fuzzing tool F based on the information collected at runtime.

III. Design and Implementation

In this section, we address the optimization problem posed in Section II. In the SGMFuzz fuzzing tool, we have designed a content-aware module that conserves fuzzing resources by performing validity checks on the mutated sequences, aiming to minimize invalid inputs without compromising the optimal solution. Furthermore, we have developed a state-guided mutation module that directs the fuzzing tool to employ different fuzzing strategies based on the state of the protocol's program and the current fuzzing depth to maximize the size of M described in Equation 2.

The Algorithm 1 outlines the SGMFuzz workflow. The inputs include the target protocol implementation P_t, an initial seed sequence q_0, a protocol knowledge base d, and the fuzzing depth D. The outputs comprise the final set of seed sequences Q and the set of seed sequences Q_c that cause the target server to crash. In each fuzzing iteration, SGMFuzz selects a target state s (line 3) and a sequence M from the seed sequences (line 4) to help the fuzzer explore the target server. The sequence M is divided into three subsequences: M_1, M_2, and M_3 (line 5). M_1 drives P_t to state s, M_2 is selected for mutation, and M_3 represents the remaining subsequence. Fuzzing resources are allocated for the state s and the sequence M (line 6). SGMFuzz employs different fuzzing strategies based on the protocol's program state and the current fuzzing

Algorithm 1: State Guided Protocol Fuzzing

```
Input: $P_t$: Protocol Implementation, $q_0$: Initial Seed
       Queue, $d$: Dictionary Corpus and $D$: Current
       Fuzzing Depth
Output: $Q$: Seed Queue and $Q_c$: Crashing Queue
1  Initialize FSM $S$ and Seed Queue $Q$ with $q_0$
2  repeat
3      State $s \leftarrow$ Choosestate($S$)
4      Messages $M$, Response $R \leftarrow$ ChooseQuece($Q, s$)
5      $\langle M_1, M_2', M_3 \rangle \leftarrow M'$;
6      for $i$ from 1 to AssignEnergy($M$) do
7          INIT $\leftarrow$ CalculateDepth($D, R$);
8          if NotINIT then
9              $M_2' \leftarrow$ StateGudiedMutate($M_2, d, s$);
10         else
11             $M_2' \leftarrow$ RandMutate($M_2, s$);
12         end
13         $M' \leftarrow \langle M_1, M_2', M_3 \rangle$;
14         if ContextAware($M_2', d$) then
15             $Q_c \leftarrow Q_c \cup \{M'\}$;
16         end
17         $R' \leftarrow$ SendToServer($P_t, M'$);
18         if IsCrashes($M', P_t$) then
19             $Q_c \leftarrow Q_c \cup \{M'\}$;
20         else if IsIntersting($M', P_t, S$) then
21             $Q \leftarrow Q \cup \{M'\}$;
22             $S \leftarrow$ UpdateFSM($S, R'$);
23         end
24     end
25 until timeout $T$ reached or abort-signal;
```

depth (line 7) to mutate M_2 (lines 9, 11). After mutation, SGMFuzz reassembles the subsequences into a new message sequence M', which undergoes validity checks by the content-aware module (line 14) and then sends it to P_t (line 17). The fuzzer retains the sequences M' that cause P_t to crash or increase code and state coverage. In the latter case, the fuzzer also updates the FSM. This process continues until the fuzzing resources for the current round are exhausted, at which point the next state s and sequence M are selected.

To address the challenge (C1), we executed the protocol implementation and conducted a series of interactions with test endpoints. We capture and analyze network packets exchanged during these interactions, focusing on fields related to protocol interaction commands, protocol states, and abnormal behaviors. Referring to the protocol's RFC documentation, we constructed mappings of protocol state transitions and interaction commands, using these mappings as input from prior knowledge for the fuzzing tool. Using these inputs, we developed a content-aware module. This module employs a filtering method to screen mutated samples. At a high level, this approach safely discards a large number of invalid samples without affecting the optimal solution. Specifically, the fuzzer drives the protocol implementation through state transitions by providing protocol commands (events) to detect potential vulnerabilities. Based on the FSM modeling results of the protocol, all interpretable protocol command types can be abstracted into an input set Σ. As described in Algorithm 2, when the fuzzer mutates command types of protocol within Σ to generate new instances, if the mutated instance cannot be interpreted by the protocol implementation (i.e., it does not belong to Σ), it will not be added to the message queue Q for transmission to the target server, but instead is placed in the crash queue Q_c, thus reducing the waste of fuzzing resources.

Algorithm 2: Content-Awareness Module

```
1  $\forall M_2 \in \Sigma$ from $d$: Dictionary Corpus;
2  If $M_2' \notin \Sigma$ then $Q_c \leftarrow M' = (M_1, M_2', M_3)$;
3  Return $Q \leftarrow M'$;
```

During each fuzzing iteration, the fuzzer receives feedback from the target server, which are the results of the server interpreting the message sequence contents. **To leverage this information to address the challenge (C2)**, 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 S 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 S 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, SGMFuzz invokes our state-guided mutation module. The module reads a message sequence σ_i from d based on the current state S, which transitions the target server to a new state S', forming a new message sequence M', described as follows:

$$\exists \delta(S, \sigma_i) \rightarrow S' \quad \sigma_i \in d \quad M' = (M_1, \sigma_i, M_3) \tag{3}$$

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

Metrics Selection. We design a prototype of SGMFuzz.We select three metrics to assess the performance of the fuzzing tools: new paths discovered, newly detected program state transitions, and observed crashes. To mitigate the impact of randomness, each experiment is conducted for 24 hours and repeated ten times, with the average taken as the final result.

Experimental Setup. 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.

We chose AFLNET [4], NSFUZZ-v [6], and Chatafl [8] as our benchmark tools. AFLNET is the first open-source, state-of-the-art, and widely utilized protocol fuzzer. NSFUZZ-v extends AFLNET by incorporating static analysis to identify state variables and using this information as feedback to maximize state space coverage. Chatafl is the first grey-box protocol fuzzer that integrates AI technology with AFLNET, modifying test instances through online interaction with large language models to generate the next instance. Other fuzzing tools either have limited performance or are not open-source, making them unsuitable for comparison with our selected benchmark tools.

All our experiments are conducted on a machine equipped with an Intel(R) Core(TM) i5-8300H CPU running at 2.30GHz, 32GB of main memory, and Ubuntu 18.04 LTS.

A. Research Questions

Through comparative experiments with other network protocol fuzzing tools, we address the following questions:

RQ1. Fuzzing Capability of SGMFuzz: Can SGMFuzz explore a greater number of program execution paths?

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

RQ3. Bug Detection Capability of SGMFuzz: How does SGMFuzz's bug detection performance compare with previous grey-box fuzzing results for network protocols?

TABLE II
AVERAGE NUMBER OF TOTAL PATHS COVERED BY EACH FUZZER

Subject	SGMFuzz	AFLNET	NSFUZZ-V	CHATAFL
Live555	1460	1095(33.3%)	1273(14.1%)	1394(4.5%)
Lightftp	784	635(23.4%)	720(8.9%)	790(-0.8%)
Kamailio	4540	3978(14.1%)	4360(4.1%)	4538(0%)
AVG(IMP)	/	19.3%	7.7%	1.2%

B. Results

SGMFuzz reduces the number of non-compliant test cases by examining mutated test cases and enhances the exploration of deeper protocol state spaces through its state-guided fuzzing strategy, thereby improving the effectiveness of fuzzing.

Answer to RQ1: Table II displays the path coverage results for each fuzzer when testing three different protocols and their implementations. The results indicate that SGMFuzz significantly outperforms AFLNET and NSFuzz, achieving approximately 19% more detected paths than AFLNET, 7% more than NSFuzz and 1.2% more than Chatafl. By introducing a state-guided fuzzing algorithm, SGMFuzz can cover more new paths than AFLNET and NSFuzz. Although Chatafl shows slightly lower overall performance, its efficiency is influenced by the prompts used during its interactions with large language models. The effectiveness of SGMFuzz varies across different protocols compared to Chatafl, but considering that Chatafl relies on online interactions and requires more resources, its overall gains remain within an acceptable range. This suggests that the state-guided fuzzing strategy is beneficial in covering a richer set of execution paths in the fuzzing of the network protocol.

Answer to RQ2: 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.

TABLE III
AVERAGE NUMBER OF STATE TRANSITIONS FOR EACH FUZZER

Subject	SGMFuzz	AFLNET	NSFUZZ-V	CHATAFL
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%	0.7%

TABLE IV
AVERAGE NUMBER OF UNIQUE CRASHS FOUND BY EACH FUZZER

Subject	SGMFuzz	AFLNET	NSFUZZ-V	CHATAFL
Live555	155	140	152	155
Kamailio	3	1	2	3

Answer to RQ3: Table IV presents the number of crashes discovered by each fuzzer within the same testing time frame. SGMFuzz 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.

V. CONCLUSION

In summary, our design for state-guided mutation involves collecting and dynamically inferring information during the fuzzing process, using events to drive state transitions in the program to explore a larger state space. Additionally, we check mutation-based fuzzing instances to minimize the generation of invalid seeds, thereby conserving fuzzing resources and enhancing overall system efficiency.

REFERENCES

[1] "The heartbleed bug," 2014. [Online]. Available: https://heartbleed.com/

[2] (2021) Wannacry ransomware attack. Accessed on 2022-04-25. [Online]. Available: https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

[3] Google, "American fuzzy lop - a security-oriented fuzzer," https://github.com/google/AFL, 2023.

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

[5] J. Li, S. Li, G. Sun, T. Chen, and H. Yu, "Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 2673–2687, 2022.

[6] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "Nsfuzz: Towards efficient and state-aware network service fuzzing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–26, 2023.

[7] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "{TCP-Fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 489–502.

[8] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," *Proceedings 2024 Network and Distributed System Security Symposium*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:265296188