applied sciences

MDPI

Article

SATFuzz: A Stateful Network Protocol Fuzzing Framework from a Novel Perspective

Zulie Pan [1,2], Liqun Zhang [1,2], Zhihao Hu [1,2,*], Yang Li [1,2] and Yuanchao Chen [1,2]

[1] College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; panzulie17@nudt.edu.cn (Z.P.); zhanglq@nudt.edu.cn (L.Z.); liyanghf@nudt.edu.cn (Y.L.); chenyuanchao@nudt.edu.cn (Y.C.)

[2] Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China

* Correspondence: huzhihao@nudt.edu.cn

Abstract: Stateful network protocol fuzzing is one of the essential means for ensuring network communication security. However, the existing methods have problems, including frequent auxiliary message interaction, no in-depth state-space exploration, and high shares of invalid interaction time. To this end, we propose SATFuzz, a stateful network protocol fuzzing framework. SATFuzz first prioritizes the states identified by the status codes in response messages, then randomly selects a state to test among the high-priority states, and determines its corresponding optimal test sequence, which is composed of the minimum pre-lead sequence, the test case, and the fittest post-end sequence. Finally, SATFuzz uses a quasi-recurrent neural network (QRNN) to filter the test cases before performing interaction, and only the optimal test sequence, including the valid test case, can be fed to the protocol entity. To verify the proposed framework, we conduct extensive experiments with the state-of-the-art fuzzer on two popular protocols. The results show that the vulnerability discovery efficiency of the proposed approach increases by at least 1.48 times (at most by 3.06 times), making it superior to the rival methods. This not only confirms the effectiveness of SATFuzz in terms of improving the vulnerability discovery efficiency but also shows that SATFuzz has significant advantages.

Keywords: stateful network protocol fuzzing; status code; auxiliary message; quasi-recurrent neural network (QRNN); deep learning; vulnerability discovery

Citation: Pan, Z.; Zhang, L.; Hu, Z.; Li, Y.; Chen, Y. SATFuzz: A Stateful Network Protocol Fuzzing Framework from a Novel Perspective. *Appl. Sci.* **2022**, *12*, 7459. https://doi.org/10.3390/app12157459

Academic Editors: Cheng Huang, Weina Niu and Wang Yang

Received: 26 June 2022
Accepted: 22 July 2022
Published: 25 July 2022

1. Introduction

Network protocols are the foundations of networks, and security vulnerabilities triggered by their incorrect implementations usually result in severe consequences. Therefore, network protocol vulnerability discovery has become a research hotspot in the field of network and information security in recent years. Currently, fuzzing is the most commonly used and most effective network protocol vulnerability discovery method. Its main principle is to construct test cases through generation or mutation, use them as the inputs of the protocol entity, and monitor the protocol entity to find vulnerabilities in the examined network protocol implementations [1].

Regarding the relationships between adjacent messages, network protocols can be divided into stateless and stateful network protocols [2]. Stateless network protocols have no context correlations between adjacent messages. For example, each request message of the Internet Control Message Protocol (ICMP) is independent. Stateful network protocols contain correlations between adjacent messages, and the protocol entity may transition to a new state after processing messages. For example, complete communication in the File Transfer Protocol (FTP) has a series of transitions in related states. Therefore, stateful network protocol fuzzing is more complicated than stateless network protocol fuzzing because the state space of the corresponding protocol entity is unknown and large [3].

Traditional stateful network protocol fuzzing methods mainly include coverage-based grey-box fuzzing [4,5] and stateful black-box fuzzing [6,7]. Coverage-based grey-box fuzzing cannot effectively traverse the state space of the protocol entity due to a lack of state information [8–10], and stateful black-box fuzzing cannot update a finite state machine because it does not save valid test cases, and thus the efficiencies of both types of methods are seriously affected. Stateful-coverage-based grey-box fuzzing appears to solve the problems of traditional methods [11].

This method does not need to understand protocol specifications and instead constructs test cases through mutation and uses state feedback to guide the fuzzing process. However, the existing methods have problems, such as frequent auxiliary message interaction, no in-depth state-space exploration, and high shares of invalid interaction time. These problems cause protocol entities to execute many invalid test cases, which waste fuzzing time and make it difficult to find vulnerabilities quickly and effectively. To improve the efficiency of network protocol vulnerability discovery, we propose SATFuzz to the above problems.

The main contributions of this paper are summarized as follows.

(1) We propose a fuzzing process optimization method to efficiently and deeply explore the state space of the protocol entity. According to the characteristics of status codes, we prioritize states. High-priority states mean that exploring them leads to higher state-space coverage and more potential bugs. Then, we randomly select a state to test among the high-priority states and screen out the corresponding optimal test sequence. The optimal test sequence is composed of the minimum pre-lead sequence, the test case, and the fittest post-end sequence.
(2) We propose a test case filtration method to avoid invalid interactions. We use a quasi-recurrent neural network (QRNN) to learn the internal relationships between the mutation modes and validity levels to judge the validities of test cases before performing interaction. Then, the filtration method quickly filters invalid test cases that cannot be received by the protocol entity, i.e., those that do not meet the protocol specification or match the state of the protocol entity.
(3) We propose a stateful network protocol fuzzing framework, named SATFuzz, and conduct extensive experiments to evaluate its performance. The results show that optimizing the fuzzing process in terms of state selection and test sequence selection can improve the vulnerability discovery efficiency of SATFuzz. In addition, reducing the invalid interaction time can further improve the efficiency.

In the next section, we review some representative related studies to outline the motivation of our research. Section 3 introduces SATFuzz in detail, including the state selector, test sequence selector, and test case filter. In Section 4, we systematically evaluate SATFuzz. Finally, we summarize our work in Section 5.

2. Related Work

2.1. Network Protocol Fuzzing

Fuzzing is a popular testing technique to detect bugs in software systems. Many fuzzing approaches have been proposed to test the implementations of network protocols in academia [7,12] and industry [6,13]. Sulley [12] provides users with a large number of data formats to describe the protocol. Before fuzzing, users need to utilize these formats to define all necessary blocks. Currently, Sulley has ceased maintenance, and the BOOFUZZ [7] project is the successor of Sulley.

Peach [6] generates test cases based on DataModel and performs fuzzing based on StateModel. Peach can expand the Peach Pit file to support different network protocols without modifying the source code. beSTORM [13] uses automated analysis technology to obtain protocol knowledge, which is then converted into an automated test set. Most fuzzers take a black-box fuzzing approach and a generation-based approach, which means that new message sequences are generated from scratch based on manually constructed protocol specifications.

Manually constructing a model of the protocol is tedious and error-prone. A better approach is to automatically reverse engineer the protocol either for or during fuzzing. Some black-box approaches [14,15] learn the message structure from a given corpus of messages. Ref. [14] presented TreeFuzz, a generic approach for generating structured data without an a priori known model. The key idea is to exploit a given corpus of example data to automatically infer probabilistic, generative models that create new data with properties similar to the corpus.

Ref. [15] proposed a method to automatically generate test cases for the black-box fuzzing of proprietary network protocols. The method uses neural-network-based machine-learning techniques to learn a generative input model of proprietary network protocols by processing their traffic and to generate new messages using the learned model. Some white-box approaches [16,17] that actively explore the protocol implementation to uncover message structure. Polyglot [16] uses dynamic analysis techniques, such as tainting and symbolic execution to extract the message format from the protocol implementation. Ref. [17] presented Tupni, a tool that can reverse engineer an input format with a rich set of information, including record sequences, record types, and input constraints.

In contrast to these above approaches, several recent approaches [11,18,19] perform stateful protocol fuzzing. AFLNET [11] adopts a mutation-based method without understanding the protocol specifications and identifies the state of the given server based on its status codes and uses state feedback to guide fuzzing. Ref. [18] proposed a practical tool by extending TLS-Attacker to learn comprehensive state models of multiple DTLS implementations. By comparing these learned state models, the user can infer vulnerabilities in DTLS implementations. Ref. [19] proposed a transition-guided fuzzing approach that uses a new cover age metric named branch transition as program feedback to improve the coverage of state transitions.

There are other works. Roberto Natella [20] presented a new benchmark ProFuzzBench for stateful fuzzing of network protocols. The benchmark includes a suite of representative open-source network servers for popular protocols and tools to automate experimentation.

2.2. Deficiencies in Existing Methods

Generally, the following three deficiencies are exhibited by existing stateful network protocol fuzzing methods.

(1) Frequent auxiliary message interaction

In stateful network protocol fuzzing, messages consume time waiting and processing at the node and transmission in the network, which is a problem that cannot be ignored. Existing methods usually only focus on improving the validities of test cases without considering the optimization of the fuzzing process. As a result, only a tiny portion of the messages in the test sequence belong to the test cases, while others called auxiliary messages belong to the pre-lead sequence and post-end sequence. These auxiliary messages generate additional time overhead [21], which results in a reduction in the number of test cases participating in fuzzing per unit time and then a reduction in the probability of raising exceptions in the protocol entity, thereby, affecting the vulnerability discovery efficiency of such approaches.

(2) No in-depth state space exploration

In stateful network protocol fuzzing, in addition to code coverage, state-space coverage should also be considered. When the code coverage is roughly constant, the higher the state-space coverage is, the deeper the state-space exploration, and the more comprehensive the fuzzing process [22]. However, the existing methods can only explore a small part of the state space of the protocol entity, which reduces the probability of raising exceptions in the protocol entity and affects the efficiency of vulnerability discovery.

(3) A high share of invalid interaction time

A test case raises exceptions in the protocol entity only if the protocol entity can receive it [23]. The existing methods exhibit blindness; it is uncertain whether a test case can be received by the protocol entity before fuzzing. Most test cases are directly discarded by the protocol entity due to their invalidity; they do not meet the protocol specification or match the state of the protocol entity. The above reason results in a high share of invalid interaction time, which reduces the probability of raising exceptions in the protocol entity, thus affecting the vulnerability discovery efficiency of these methods.

Based on the above analysis, this paper is dedicated to compensating for the above shortcomings to improve the efficiency of vulnerability discovery.

3. Architecture

The structure of SATFuzz is shown in Figure 1 and mainly includes three parts: a state selector, test sequence selector, and test case filter.

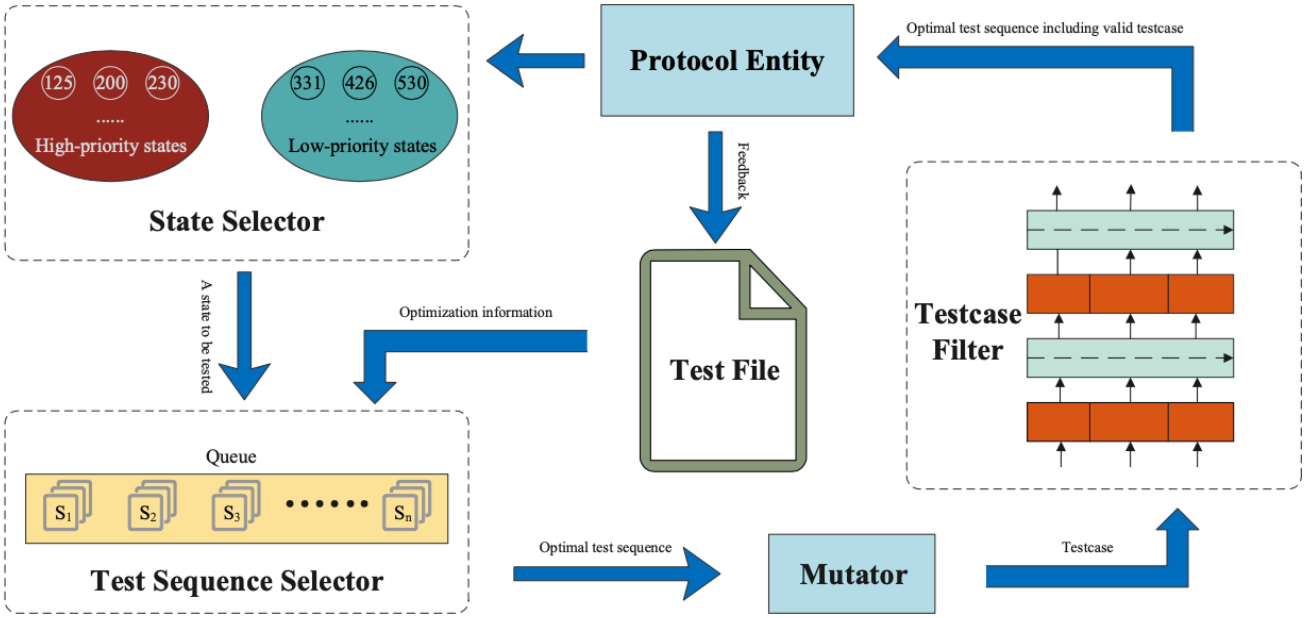

Figure 1. The structure of SATFuzz.

First, the state selector randomly selects one state to test among the high-priority states. Second, the test sequence selector selects the optimal test sequence that can reach the state from the queue according to the test file. Then, the old test case in the optimal test sequence is mutated by Mutator, while the minimum pre-lead sequence and the fittest post-end sequence remain unchanged. Next, the test case filter determines the validity of the new test case and only inputs the optimal test sequence, including the valid test case, to the protocol entity. Finally, SATFuzz updates the test file with feedback and prepares for the next testing step.

3.1. Simplified Finite-State Machine

We use a simplified finite state machine (SFSM) as a formal description model for stateful network protocol interactions. The SFSM can be defined as a triple M = (S,I,T). $S = \{s_0, s_1, \ldots, s_n\}$ is a finite set of state symbols, and s_0 represents the initial state of M. At any time, M can only be in a specific state s_i, and M begins to receive inputs from s_0; $I = \{i_1, i_2, \ldots, i_m\}$ is a finite set of input symbols; T: S × I→S is a state transition function, which is a one-to-one mapping. When the SFSM is applied to the server, S represents the set of status codes in the response messages, and I represents the set of request message types that can be received and processed normally.

Taking the SFSM of the FTP as an example [24], the specific values corresponding to its state symbols and input symbols are shown in Table 1. M, shown in Figure 2, only contains part of the state transition relationships, where the transition from s_0 to s_1 indicates that, when the server is in state 220 (abstracted by s_0), if a request message of the USER type (abstracted by i_1) is received and processed normally, then the server transitions to state 331 (abstracted by s_1).

Table 1. Specific values corresponding to different state symbols and input symbols.

State Symbol	Status Code	Input Symbol	Request Message Type
s_0	220	i_1	USER
s_1	331	i_2	PASS
s_2	230	i_3	PWD
s_3	257	i_4	TYPE
s_4	200	i_5	STOR
s_5	150	i_6	MKD
s_6	250	i_7	RETR
s_7	221	i_8	RMD
		i_9	CWD
		i_{10}	DELE
		i_{11}	CDUP
		i_{12}	QUIT

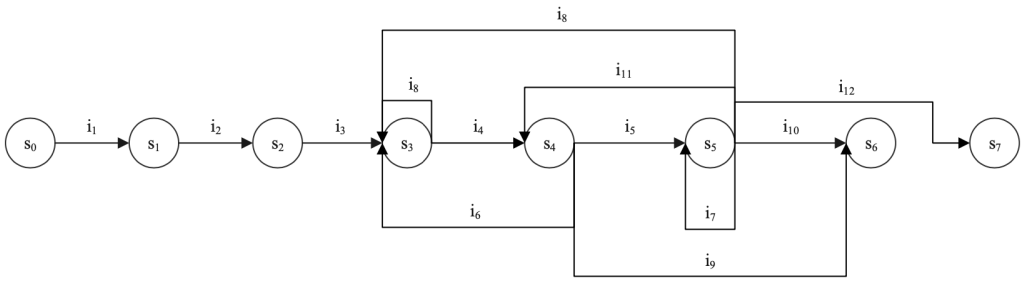

Figure 2. The SFSM of the FTP.

3.2. State Selector

Existing methods treat states equally and use polling to select states for testing. Different states have different capabilities for exploring the state space of the protocol entity, and this directly affects the efficiency of vulnerability discovery. A state in this paper is identified by xyz, which is a three-digit number, where each number has a specific meaning [25]. The first digit defines the type of the state, which is 1yz (the initial state), 2yz (the complete state), 3yz (the intermediate state), 4yz (the temporary rejection state), or 5yz (the permanent rejection state); the second digit defines the meaning of the state, which is x0z (syntax), x1z (request), x2z (connection), x3z (authentication), or x5z (file); and the third digit gives the state a richer meaning. Therefore, we rank states in order of importance by referring to the meaning of the first two digits, and the classification results are shown in Table 2.

Table 2. State classification results.

x/y	0	1	2	3	5
1	high *	high	high	high	high
2	high	high	high	high	high
3	high	high	low #	low	high
4	high	high	low	low	high
5	high	high	low	low	high

* high-priority states. # low-priority states.

One contains low-priority states, including 32z, 33z, 42z, 43z, 52z, and 53z. A protocol entity in these states remains unchanged with a high probability after processing test cases. In contrast, the remaining cases belong to high-priority states. A protocol entity in these states transitions to the new state with a high probability after processing test cases.

In the beginning, the state selector randomly selects states to test among the high-priority states for certain cycles. Then, it chooses the states that achieved outstanding performance in contributing to increased code or state-space coverage when previously selected. Now, we illustrate how the selected states affect state-space exploration. Suppose we perform fuzzing on LightFTP, a server that implements the FTP and one of the subjects in our evaluation, with the test case ("MKD test") and the post-end sequence ("CWD test+STOR test .txt+LIST+QUIT").

As shown in Figure 3a, the number of covered unique state transitions is 6 when one of the high-priority states (230) is tested. In contrast, as shown in Figure 3b, the number of covered unique state transitions is only 2 when one of the low-priority states (530) is tested, and the state-space coverage rate is reduced by 66.7%. For the FTP, 230 means that the user logs in successfully, and subsequent request messages can play their original role only under this condition. Otherwise, all these requests are not allowed, resulting in a loop in the single state.

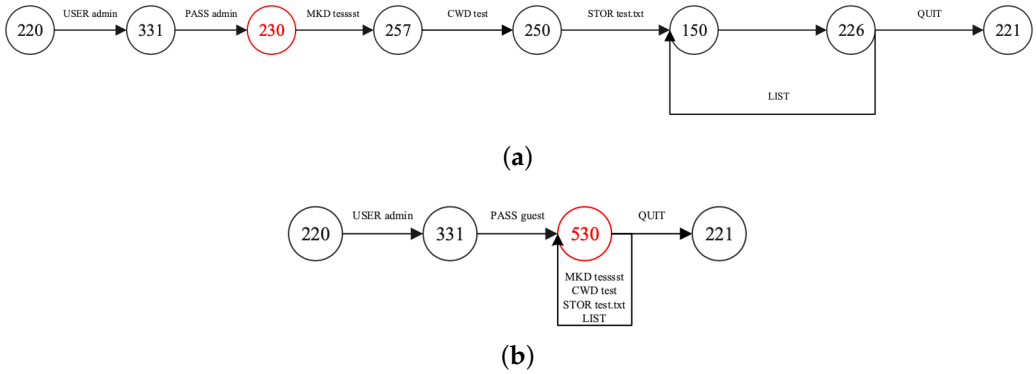

Figure 3. State transition diagrams produced when different states are selected for testing. (**a**) 230 as a state to be tested. (**b**) 530 as a state to be tested.

3.3. Test Sequence Selector

The existing stateful network protocol fuzzing methods mainly include three steps [26–28]. First, the protocol entity is guided to a specific state to be tested through the interaction of normal messages. The sequence formed by these messages is called the pre-lead sequence. A test case composed of one or more mutated messages is then fed into the protocol entity.

If the protocol entity crashes or does not respond for a long time after processing the test case, it is necessary to save errors for further analysis. Finally, if no abnormality is found, the communication needs to be terminated through the interaction of normal messages, and the next testing step is prepared. The sequence formed by these messages is called the post-end sequence. The test sequence is composed of the pre-lead sequence,

test case, and post-end sequence, and the messages composing the pre-lead sequence and post-end sequence are called auxiliary messages.

Once the state to be tested is selected, the existing methods select any test sequence that can reach this state from the queue, and no further screening is performed. The main components of the test sequence are auxiliary messages, which cause additional time overhead during waiting, processing, and transmission, thereby, affecting the efficiency of vulnerability discovery. Simply reducing the lengths of auxiliary messages can save time; however, this approach influences the post-end sequence when exploring the state space, which also affects the efficiency of vulnerability discovery.

Therefore, by reasonably optimizing the auxiliary messages, we screen out the optimal test sequence (including the minimum pre-lead sequence), which is responsible for quickly leading the protocol entity to the state to be tested, and the fittest post-end sequence, which is responsible for exploring the state space in depth after no abnormalities are detected. The above information is recorded in the test file for the test sequence selector to read. The test file is written in XML, including eight different elements, as shown in Table 3. Among them, GuidingMesNum controls the minimum pre-lead sequence, which, as the name implies, is the pre-lead sequence with the least number of request messages. BackMesNum and HiPriMesPro jointly control the fittest post-end sequence to ensure that the proportion of request messages that can trigger high-priority states within a certain length remains maximized.

In addition to ending the communication process, another function of the fittest post-end sequence is to explore the state space in depth, which requires more request messages that can be received and processed normally by the protocol entity. The request messages that meet this condition correspond to high-priority states. When a certain state to be tested simultaneously meets the minimum pre-lead sequence and fittest post-end sequence, the corresponding test sequence is recorded in OptimSeqName for the test sequence selector to select from the queue.

Table 3. Basic elements of the test file.

Elements	Meaning
ProcOptim	Main element of the test file, representing all the information needed to optimize the stateful network protocol fuzzing process
TotalStates	Child element of ProcOptim, representing the total number of states that have been explored
StateSeqInfo	Child element of ProcOptim, representing the information of the optimal test sequence corresponding to a certain state
StatusCode	Child element of StateSeqInfo, representing the state identified by the status code
GuidingMesNum	Child element of StateSeqInfo, representing the number of request messages in the pre-lead sequence
BackMesNum	Child element of StateSeqInfo, representing the number of request messages in the post-end sequence
HiPriMesPro	Child element of StateSeqInfo, representing the proportion of request messages that can trigger high-priority states in the post-end sequence
OptimSeqName	Child element of StateSeqInfo, representing the name of the optimal test sequence

3.4. Test Case Filter

The existing methods directly input test cases from Mutator into the protocol entity, without knowing whether the protocol entity can receive them, and most test cases are discarded due to invalidity—namely, not meeting the protocol specification or matching the

state of the protocol entity; this affects the efficiency of vulnerability discovery. Therefore, we use deep learning to filter the test cases, determine the validity of test cases before interaction, and avoid invalid interactions.

A test case is similar to long time-series data, which means that the beginning data in a long sequence can affect the ending data; therefore, we mainly consider two aspects when designing the test case filter. One is how to improve the data processing capabilities of the filter; the other is how to improve its parallel computing capabilities. By comparing and analyzing common deep learning models, we finally choose a QRNN [29] to implement the test case filter. In order to eliminate the differences between different stateful network protocols, we design a general input representation method, and its algorithm is shown in Algorithm 1. We divide the old test case M in the optimal test sequence by bytes to form a byte stream vector $X = \langle x_1 + 1, x_2 + 1, \ldots, x_m + 1 \rangle$, where x_i is the i-th byte in M, $x_i \in \{0, 1, \ldots, 255\}$, and m is the total number of bytes in M.

We also divide the new test case M' mutated by M in the same way and form a byte stream vector $X' = \langle x'_1 + 1, x'_2 + 1, \ldots, x'_n + 1 \rangle$, where x'_i is the i-th byte in M', $x'_i \in \{0, 1, \ldots, 255\}$, and n is the total number of bytes in M'. We perform tail zeroization on the byte stream vector with a smaller length (X or X') to make both lengths consistent and take the byte stream vector $P = X \oplus X' = \langle p_1, p_2, \ldots, p_k \rangle$ generated by the XOR operation between the two as the input of the QRNN, where p_i is the i-th byte in P, $p_i \in \{0, 1\}$, and k=max{m,n} is the total number of bytes in P. At the same time, 1 is used as the output value of a valid test case, and 0 is used as the output value of an invalid test case.

Algorithm 1 General representation of the QRNN input.

Input: old test case M and new test case M'
Output: byte stream vector P

1: Extract the old test case M from the optimal test sequence and divide M by bytes to form a byte stream vector X.
2: Divide the new test case M' mutated by M in the same way to form a byte stream vector X'.
3: Compare the lengths of X and X', add zero to the tail of the shorter vector, and XOR the two vectors to form a byte stream vector P.
4: Return P.

Most test cases constructed by the existing methods via mutation are invalid cases that cannot be received by the protocol entity, resulting in a proportional imbalance between the invalid samples and valid samples in the original training set [30], which seriously affects the classification performance of the test case filter. Therefore, we use a custom semirandom undersampling method to preprocess the original training set. By setting the proportions of invalid samples and valid samples in advance and taking the negative impact of the sample byte length on the implementation speed into account, we retain valid samples and randomly select invalid samples to form a class-balanced training set in terms of the distribution ratio of invalid samples with different byte lengths.

Then, the test case filter is implemented after training the QRNN for a certain number of epochs with the class-balanced training set. After the test case filter enters a valid test case into the protocol entity for execution, the resulting feedback is used to update the test file. At the same time, the next testing phase starts.

4. Experimental Analysis

4.1. Experimental Setup

The FTP is one of the most commonly used stateful network protocols and is widely used for file transfer. The design concept of the TCP/IP protocol family has led to the malicious use of the vulnerability of the FTP. Therefore, the security of the FTP has become a research focus in the field of vulnerability discovery.

According to statistics, nearly 4 million video surveillance equipment packages worldwide support a stateful network protocol, the Real Time Streaming Protocol (RTSP). However, the RTSP has been proven to have many medium- and high-risk buffer overflow vulnerabilities in its implementation. Hence, research on the security of the RTSP has theoretical significance and application value [31].

This paper selected LightFTP [32], a lightweight server that supports the FTP, and Live555 [33], a streaming media server that supports the RTSP, as the test objects; their versions are 5980ea1 and ceeb4f4, respectively. The hardware configuration includes an Intel(R) Xeon(R) Gold 6139 CPU @ 2.30 GHz, 376 of GB RAM, and the Ubuntu 18.04 server.

For the above test objects, we compare SATFuzz with a baseline approach called AFLNET, a stateful-coverage-based fuzzer, and use the number of triggered crashes within the same time period as an indicator to evaluate the vulnerability discovery efficiency of each method. The higher the number of triggered crashes is, the higher the vulnerability discovery efficiency of the corresponding approach.

We recorded message sequences for the most common usage scenarios of LightFTP and Live555, such as uploading a file and starting to stream a media source. These message sequences serve as the initial seed corpus for SATFuzz to use. The same is true for AFLNET and AFLNWE in the experiment. However, BOOFUZZ started with a detailed model of the protocol, including the message templates and the state machine.

4.2. State Selection

In Section 3.2, we split the states into two according to their importance. One type includes high-priority states, and the other includes low-priority states. In each cycle, the state selector only focuses on high-priority states and ignores low-priority states. To verify the effectiveness of this method, we conducted a comparative experiment with three state selection criteria. To reduce the impact of randomness, we ran 10 isolated instances of each group for each subject. We set the test time to 12 h, and the experimental results of the three groups are shown in Figure 4.

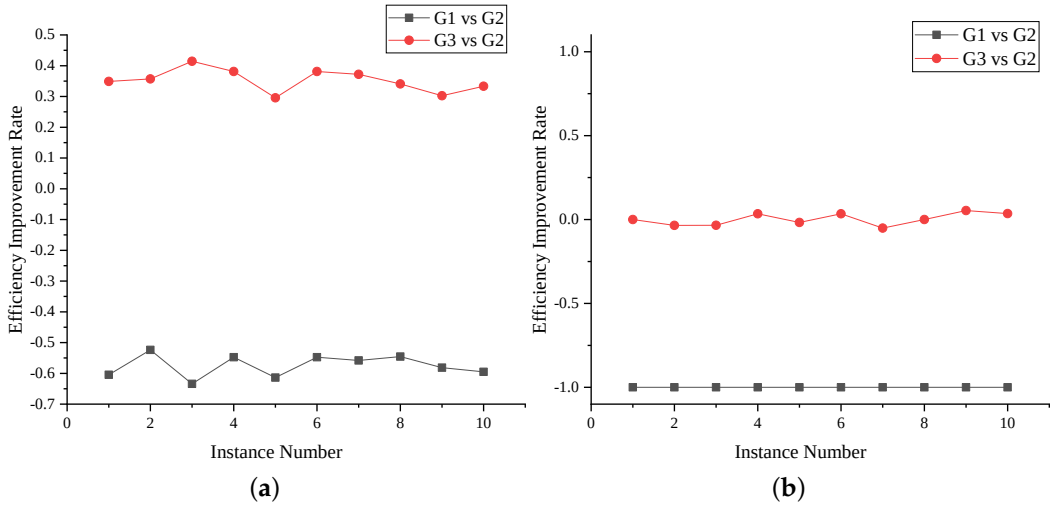

Figure 4. Comparative experimental results obtained with different state selection criteria. (**a**) LightFTP. (**b**) Live555.

The state selection criterion for group one (G1 for short) includes low-priority states only. Group two (G2 for short) chooses round-robin scheduling, which is the same technique as that used in AFLNET. Group three (G3 for short) realizes our method. It can be seen from Figure 4a that, for LightFTP, in terms of vulnerability discovery efficiency, G1 decreases by 57.51% on average compared to G2, while G3 increases by 34.27% on average compared to G2. Figure 4b shows that, for Live555, in terms of vulnerability discovery efficiency, G1 is zero, while G3 and G2 have the same value.

By analyzing the experimental results, we find that high-priority states performed much better in exploring the state space than low-priority states, which verifies the effectiveness of our state selection method. In addition, the performance of our method on Live555 was worse than that on LightFTP because, when the SFSM is applied to Live555, the status code set does not contain low-priority states, a condition for which G3 and G2 have the same state selection criterion; in contrast, the state selection criterion of G1 fails to take effect.

4.3. Test Sequence Selection

In Section 3.3, defines the optimal test sequence as including the minimum pre-lead sequence and the fittest post-end sequence, which is determined for a certain state to be tested by the test sequence selector. To make the best use of the post-end sequence in state-space exploration, we increased the proportion of request messages that can trigger high-priority states under the premise of a certain length.

To verify the effectiveness of this method, we conducted comparative experiments in terms of three test sequence selection criteria. To reduce the impact of randomness, we ran 10 isolated instances of each group for each subject. We set the test time to 12 h, and the experimental results of the three groups are shown in Figure 5. The common point regarding the test sequence selection criteria of the three groups is that they all include the minimum pre-lead sequence. When selecting the post-end sequence, group one (G1 for short) prefers the sequence with the highest proportion of request messages that can trigger low-priority states within a certain length.

Group two (G2 for short) chooses randomness, which is the same choice as that of AFLNET, and group three (G3 for short) realizes our method. It can be seen from Figure 5a that, for LightFTP, in terms of the vulnerability discovery efficiency, G1 decreased by 72.15% on average compared with G2, while G3 increased by 20.11% on average compared with G2. Figure 5b shows that, for Live555, in terms of the vulnerability discovery efficiency, G1, G2, and G3 are almost the same.

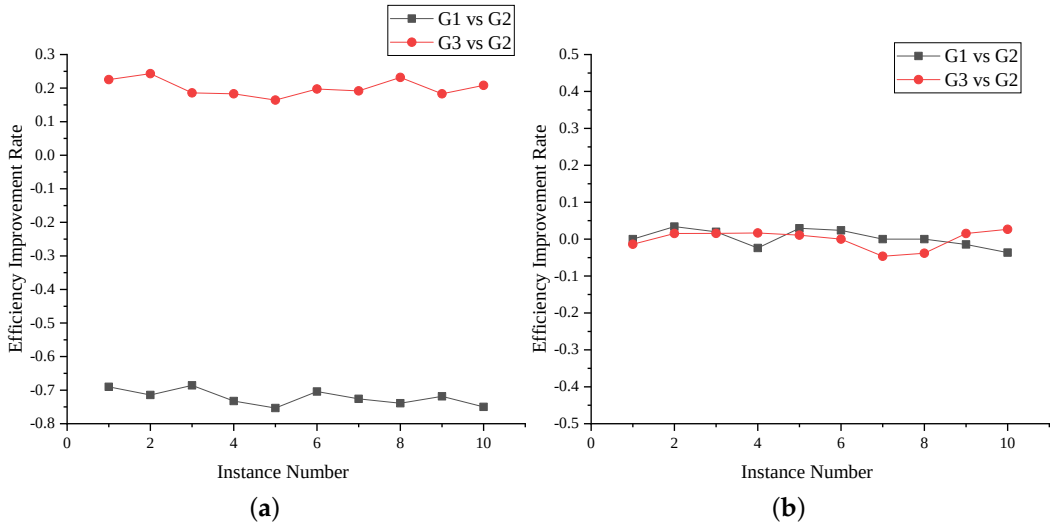

Figure 5. Comparative experimental results obtained with different test sequence selection criteria. (**a**) LightFTP. (**b**) Live555.

By analyzing the experimental results, we found that it was easier to request messages that can trigger high-priority states in the post-end sequence to be received and processed normally by the protocol entity, which remains normal after processing the test case, indicating its stronger ability to explore the state space deeply. In addition, our method had no advantage on Live555 because, when the SFSM was applied to Live555, the status code set only contained high-priority states, a condition for which the test sequence selection criteria of the three groups are exactly the same.

4.4. Test Case Filtration

In Section 3.4, we generalized the input and output of the test case filter, and the parameter names and their corresponding values are shown in Table 4. Each byte component of P has only two values, 0 and 1; thus, the input size was set to 2. The classification process has only two results, valid and invalid; thus, the output size was set to 2. The default values of the number of hidden layers, bidirectional flag, dropout rate, and learning rate were 2, True, 0.4, and 0.001, respectively. Moreover, the cross-entropy loss function was used to improve the classification performance of the test case filter.

Table 4. The parameter settings of the test case filter.

Parameter Name	Parameter Value
Input Size	2
Output Size	2
Hidden Layer Size	4(LightFTP) 2(Live555)
Hidden Layer Number	2
Bidirectional Flag	True
Epoch	500
Dropout	0.4
Learning Rate	0.001
Optimization Algorithm	Adam
Loss Function	Cross-Entropy

By fixing the other parameters, we compared the influences of different hidden layer sizes on the classification performance of our approach. The experimental results are shown in Table 5. As classification performance requires specific evaluation indicators and a single precision rate or recall rate cannot meet the actual evaluation needs, we took the weight-equivalent harmonic mean of the two, the F1 value, as the measurement standard, and its definition is given as follows:

$$F1 = 2\frac{Precision \times Recall}{Precision + Recall} = \frac{2TP}{2TP + FP + FN} \quad (1)$$

In the formula: Precision denotes the precision rate; Recall denotes the recall rate; P denotes invalid test cases; N denotes valid test cases; TP denotes true-positive cases; FP denotes false-positive cases; and FN denotes false-negative cases.

Table 5. Classification performance of the test case filter under different hidden layer sizes.

Hidden Layer Size	Test Object	TN	FN	TP	FP	F1 Value
2	LightFTP	73	1604	7026	319	87.96%
	Live555	37	833	6710	707	89.71%
4	LightFTP	49	921	6861	403	91.20%
	Live555	62	1207	6095	532	87.52%
8	LightFTP	67	1511	7316	358	88.67%
	Live555	92	1839	7067	370	86.74%

The experimental results show that the corresponding relationship between the classification performance and hidden layer size varies slightly for different test objects. For LightFTP, when the hidden layer size is 4, the F1 score reaches its highest value, and the test case filter has the best classification performance. In contrast, for Live555, the F1 score reaches its highest value, and the test case filter has the best classification performance when the hidden layer size is 2. Figure 6 visually shows the difference between the experimental results of the two test case filters.

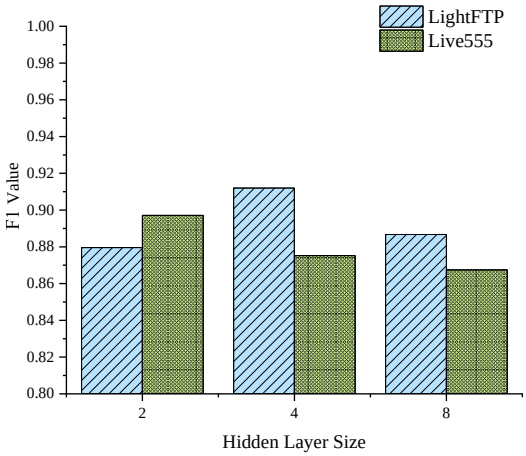

Figure 6. Comparative experimental results produced by the two test case filters.

4.5. Comparasion with AFLNET

We evaluate the effectiveness of SATFuzz in comparison with a stateful grey-box fuzzer called AFLNET. Specifically, we compare the average invalid interaction rates and numbers of crashes exposed in a 12-h fuzzing campaign on LightFTP and Live555. To reduce the impact of randomness, we ran 10 isolated instances of AFLNET and SATFuzz for each subject. The results are shown in Figure 7.

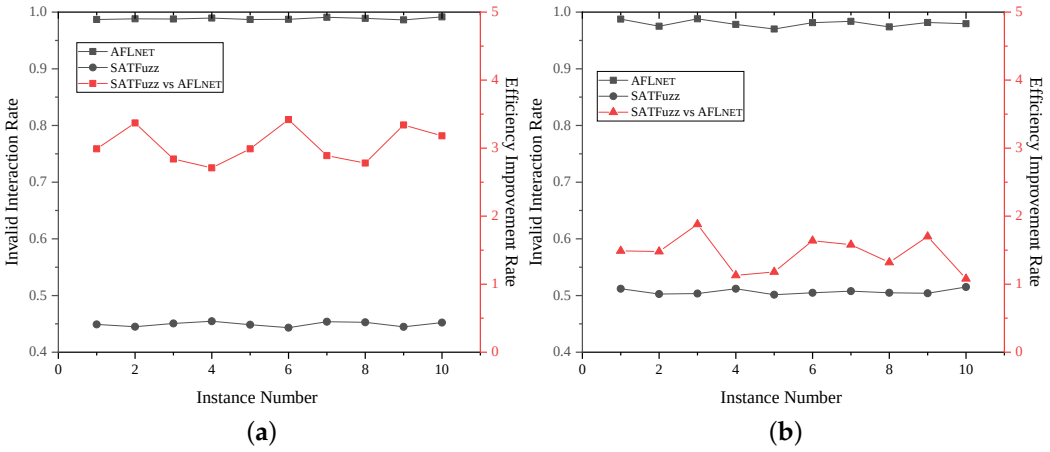

Figure 7. Comparison of the experimental results produced by AFLNET and SATFuzz. (**a**) LightFTP. (**b**) Live555.

Figure 7a shows that, for LightFTP, the invalid interaction rate of AFLNET was 98.84% on average, while the invalid interaction rate of SATFuzz was only 44.95% on average, showing that the proposed approach avoided more than half of the invalid interactions. Figure 7b shows that, for Live555, the invalid interaction rate of AFLNET was 97.99% on average, while the invalid interaction rate of SATFuzz was only 50.68% on average, showing that the proposed approach avoided nearly half of the invalid interactions.

The interaction processes of the FTP and RTSP include connection establishment. Only after the client establishes a stable and reliable connection with the server can the subsequent reasonable client requests be received and processed normally by the server. In contrast, establishing an FTP connection requires additional authentication, and the interaction process is slightly more complicated. The test cases constructed by mutation usually have difficulty satisfying the interactive conditions. Therefore, the invalid interaction rates of AFLNET for LightFTP and Live555 were close to 100%, and the invalid interaction rate obtained on LightFTP was slightly higher than that obtained on Live555.

Moreover, the classification performance of the test case filter was better for LightFTP as the average request message length of the FTP was less than that of the RTSP, which means that the context correlation of the test case was closer, making it easier for the QRNN to learn. Figure 7a,b show that, for LightFTP and Live555, the vulnerability discovery efficiency levels of SATFuzz were 3.05-times and 1.45-times higher than those of AFLNET, respectively. The experimental results show that, compared with competitors of the same type, the method in this paper had clear advantages in improving the efficiency of vulnerability discovery and achieved the expected results.

4.6. Vulnerability Discovery

In order to verify whether our proposed method can effectively improve the efficiency of vulnerability discovery and can be applied in reality, we compared SATFuzz with BOOFUZZ, AFLNWE, and AFLNET. BOOFUZZ is a stateful blackbox fuzzer, AFLNWE is a stateless coverage-guided fuzzer, and AFLNET is a stateful-coverage-based fuzzer. They are all open source and widely used in practice. For all fuzzers, we counted the numbers of vulnerability found and measured the time they took to expose these vulnerabilities. To reduce the impact of randomness, we ran 10 isolated instances of each group for each subject. We set the test time to 12 h, and the experimental results are shown in Table 6.

CVE-2018-4013 vulnerability exists in the HTTP packet-parsing functionality of the LIVE555 RTSP server library, which can cause a stack-based buffer overflow, resulting in code execution. CVE-2019-7733 vulnerability is a buffer overflow in Live555 because handleRequestBytes has an unrestricted memmove. According to the experimental results, it can be seen that SATFuzz outperformed both fuzzers on all vulnerabilities. Specifically, SATFuzz found two vulnerabilities and improved the time efficiency. Compared to the state-of-the-art fuzzer AFLNET, SATFuzz found vulnerabilities in a shorter time. In the discovery of the vulnerability CVE-2019-7733, SATFuzz took less than two thirds of the time used by AFLNET.

Table 6. The vulnerabilities found and average time to error comparison.

CVE-ID	LEVEL	Time to Error			
		BOOFUZZ	AFLNWE	AFLNET	SATFuzz
CVE-2018-4013	super critical	>12 h	1 h 21 m 37 s	1 h 18 m 10 s	1 h 10 m 25 s
CVE-2019-7733	high critical	>12 h	2 h 29 m 36 s	1 h 45 m 42 s	1 h 2 m 53 s

5. Discussion

5.1. Theoretical and Practical Implications

In contrast to generation-based fuzzers, SATFuzz takes a mutational approach and uses state-feedback to guide the fuzzing process. SATFuzz is seeded with a corpus of recorded message exchanges between the server and an actual client. No protocol specification or message grammars are required.

In contrast to existing stateful network protocol fuzzing methods, we made improvements from three perspectives. To explore the state space in depth, we prioritized states according to their characteristics and the state selector randomly selected states to test among the high-priority states. To reduce the extra time overhead, we considered the minimum pre-lead sequence and the fittest post-end sequence together, and the Test Sequence Selector screened out corresponding optimal test sequences for the states to be tested. To avoid invalid interactions, we chose a QRNN to implement a test case filter that can determine the validity of the test cases before interaction.

We summarize three problems in existing network protocol fuzzing methods: frequent auxiliary message interaction, no in-depth state-space exploration, and high shares of invalid interaction time. To this, we proposed SATFuzz to improve these three aspects. Compared with other methods, we found that SATFuzz triggered more crashes in the same time, which can effectively improve the efficiency of network protocol vulnerability

discovery. In addition, SATFuzz can be applied in practice to find high-risk vulnerabilities of real software in a shorter time.

5.2. Limitations and Future Works

SATFuzz can be strengthened in certain aspects.

First is the effective test case classification false positive rate problem. The disadvantage of the method in this paper is the high false positive rate of effective test case classification. In the future, we will study how to optimize the structure of the test case filtering model to improve its ability to correctly classify effective test cases and further improve the efficiency of network protocol fuzzing.

Second is state prioritization issues. This article prioritizes the states with reference to the meaning of the first two digits of the status code and does not fully consider the inherent meaning of the status code. In the future, the state prioritization will be refined to further improve the efficiency of stateful protocol fuzzing.

Third is no status code issues. This paper requires that the stateful protocol entity must meet the condition that the response message contains the status code; thus, for the stateful protocol entity that does not contain the status code in the response message, the method in this paper is no longer applicable. In the future, we will study how to fuzz test stateful protocol entities whose response messages do not contain status codes to expand the scope of application of this method.

6. Conclusions

In this paper, we proposed a stateful network protocol fuzzing framework called SATFuzz. The key idea is to improve the vulnerability discovery efficiency from three perspectives: reducing the extra time overhead, exploring the state space in depth, and avoiding invalid interactions. Specifically, we prioritized states according to their characteristics, screened out corresponding optimal test sequences for the states to be tested (i.e., high-priority states), and directly filtered invalid test cases that were destined to be unable to successfully participate in the fuzzing process.

To systematically evaluate SATFuzz, we selected stateful network protocols with hidden security risks as the research object and conducted comparative experiments with the most representative method. The experimental results show that the vulnerability discovery efficiency of the proposed approach increased by at least 1.48 times (at most by 3.06 times). The experimental results verified that the vulnerability discovery efficiency of SATFuzz was far superior to that of its competitors of the same type. We proved that SATFuzz can effectively compensate for some shortcomings of the existing methods and achieved significantly improved vulnerability discovery efficiency.

Author Contributions: Conceptualization, Z.P. and L.Z.; methodology, Z.P. and Z.H.; software, L.Z. and Z.H.; validation, Z.P., L.Z. and Z.H.; investigation, Y.L. and Y.C.; resources, Y.L. and Y.C.; writing—original draft preparation, Z.P.; writing—review and editing, Z.P., L.Z., Z.H., Y.L. and Y.C.; supervision, Z.P.; project administration, Z.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key R&D Program "Cyberspace Security" grant number 2017YFB0802900.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: We built a QRNN template used for processing long time-series data. We uploaded it to a GitHub project to support related research on test case filtering in stateful network protocol fuzzing; the template can be downloaded from https://github.com/Whoolocked/QRNN-For-LTSData (accessed on 20 July 2022).

Acknowledgments: We are very thankful to Bingyang Guo and Zhijie Xie for their help with the preparation of the experiment and reviewing the paper. We thank the anonymous reviewers for their detailed comments, which helped to improve the quality of the paper.

Conflicts of Interest: The authors declare that they have no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

QRNN	Quasi-recurrent neural network
SIP	Session Initiation Protocol
IP	Internet Protocol
SFSM	Simplified Finite State Machine

References

1. Li, Z.J.; Zhang, J.X.; Liao, X.K.; Ma, J. Survey of Software Vulnerability Detection Techniques. *Chin. J. Comput.* **2015**, *38*, 717–732.
2. Zhang, B.F.; Zhang, C.B.; Xu, Y. Network Protocol Vulnerability Discovery Based on Fuzzy Testing. *J. Tsinghua Univ. (Sci. Technol.)* **2009**, *2*, 2113–2118.
3. Munea, T.L.; Lim, H.; Shon, T. Network protocol fuzz testing for information systems and applications: A survey and taxonomy. *Multimed. Tools Appl.* **2015**, *75*, 14745–14757. [CrossRef]
4. Technical "Whitepaper" for afl-fuzz. Available online: https://lcamtuf.coredump.cx/afl/technical_details.txt (accessed on 14 July 2022).
5. libFuzzer—A Library for Coverage-Guided Fuzz Testing. Available online: https://llvm.org/docs/LibFuzzer.html (accessed on 14 July 2022).
6. Peach Fuzzer Platform. Available online: http://www.peachfuzzer.com/products/peach-platform (accessed on 14 July 2022).
7. A Fork and Successor of the Sulley Fuzzing Framework. Available online: https://github.com/jtpereyda/boofuzz (accessed on 14 July 2022).
8. Böhme, M.; Pham, V. T.; Roychoudhury, A. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Trans. Softw. Eng.* **2017**, *45*, 489–506. [CrossRef]
9. Böhme, M.; Pham, V.T.; Nguyen, M.D.; Roychoudhury, A. Directed Greybox Fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017.
10. Pham, V.T.; Böhme, M.; Santosa, A.E.; Căciulescu, A.R.; Roychoudhury, A. Smart Greybox Fuzzing. *IEEE Trans. Softw. Eng.* **2019**, *47*, 1980–1997. [CrossRef]
11. 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.
12. A Pure-Python Fully Automated and Unattended Fuzzing Framework. Available online: https://github.com/OpenRCE/sulley (accessed on 14 July 2022).
13. Dynamic Application Security Testing. Available online: https://beyondsecurity.com/solutions/bestorm.html (accessed on 14 July 2022).
14. Patra, J.; Michael, P. *Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data*; Tech. Rep. TUD-CS-2016-14664 TU Darmstadt, Department of Computer Science: Darmstadt, Germany, 2016.
15. Fan, R.; Chang, Y. Machine learning for black-box fuzzing of network protocols. In *International Conference on Information and Communications Security*; Springer: Cham, Switzerland, 2017.
16. Caballero, J.;Yin, H.; Liang, Z.; Song, D. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In Proceedings of the 14th ACM conference on Computer and Communications Security, CCS '07, Alexandria, VA, USA, 28–31 October 2007 .
17. Cui, W.; Peinado, M.; Chen, K.; Wang, H.J.; Irun-Briz, L. Tupni: Automatic reverse engineering of input formats. In Proceedings of the 15th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 27–31 October 2008.
18. Fiterau-Brostean, P.; Jonsson, B.; Merget, R.; De Ruiter, J.; Sagonas, K.; Somorovsky, J. Analysis of DTLS implementations using protocol state fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020.
19. Zou, Y.H.; Bai, J.J.; Zhou, J.; Tan, J.; Qin, C.; Hu, S.M. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21), Virtual, 14–16 July 2021.
20. 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.
21. Sutton, M.; Greene, A. *Fuzzing: Brute Force Vulnerability Discovery*; Chinese Machine Press: Beijing, China, 2009.
22. Su, P.; Ying, L.; Yang, Y. *Software Security Analysis and Application*; Tsinghua University Press: Beijing, China, 2017.

23. Wang, J.; Chen, B.; Wei, L; Liu, Y. Skyfire: Data-Driven Seed Generation for Fuzzing. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May.
24. Gascon, H.; Wressnegger, C.; Yamaguchi, F.; Arp, D.; Rieck, K. Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols. In *International Conference on Security and Privacy in Communication Systems*; Springer: Cham, Switzerland, 2015.
25. RFC Editor. Available online: https://www.rfc-editor.org/info/rfc959 (accessed on 14 July 2022).
26. Zhao, J.; Chen, S.; Liang, S.; Cui, B.; Song, X. RFSM-Fuzzing a Smart Fuzzing Algorithm Based on Regression FSM. In Proceedings of the 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Compiegne, France, 28–30 October 2013; pp. 380–386.
27. Cui, B.; Liang, S.; Chen, S.; Zhao, B.; Liang, X. A Novel Fuzzing Method for Zigbee Based on Finite State Machine. *Int. J. Distrib. Sens. Netw.* **2014**, *10*, 762891. [CrossRef]
28. Kang, H.; Wu, L.; Hong, Z.; Zhuang, H.; Zhang, Y. Fuzzing method for BGP-4 protocol based on FSM. *Comput. Eng. Appl.* **2017**, *53*, 111–117.
29. Bradbury, J.; Merity, S.; Xiong, C.; Socher, R. Quasi-Recurrent Neural Networks. *arXiv* **2016**, arXiv:1611.01576.
30. Zhou, Y. Research on Network Protocol Vulnerability Mining Method Based on Deep Learning. Master's Dissertation, University of Electronic Science and Technology of China, Chengdu, China.
31. Li, J.L.; Chen, Y.L.; Li, Z. Mining RTSP Protocol Vulnerabilities Based on Traversal of Protocol State Graph. *Comput. Sci.* **2018**, *45*, 178–183.
32. Small x86-32/x64 FTP Server. Available online: https://github.com/hfiref0x/LightFTP (accessed on 14 July 2022).
33. The LIVE555^{TM} Media Server. Available online: http://www.live555.com/mediaServer (accessed on 14 July 2022).