Hindawi
Security and Communication Networks
Volume 2022, Article ID 6880677, 13 pages
https://doi.org/10.1155/2022/6880677

Research Article

Model-Based Grey-Box Fuzzing of Network Protocols

Yan Pan, Wei Lin, Liang Jiao, and Yuefei Zhu

State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China

Correspondence should be addressed to Yuefei Zhu; yfzhu17@sina.com

Received 19 February 2022; Accepted 11 April 2022; Published 5 May 2022

Academic Editor: Irshad Azeem

The widely used network protocols play a crucial role in various systems. However, the protocol vulnerabilities caused by the design of the network protocol or its implementation by programmers lead to multiple security incidents and substantial losses. Hence, it is important to study the protocol fuzzing in order to ensure its correctness. However, the challenges of protocol fuzzing are the mutation of protocol messages and the deep interactivity of the protocol implementation. This paper proposes a model-based grey-box fuzzing approach for protocol implementations, including the server-side and client-side. The proposed method is divided into two phases: automata learning based on the minimally adequate teacher (MAT) framework and grey-box fuzzing guided by the learned model and code coverage. The StateFuzzer tool used for evaluation is presented to demonstrate the validity and feasibility of the proposed approach. The server-side fuzzing can achieve similar or higher code coverage and vulnerability discovery capability than those of AFLNET and StateAFL. Considering the client, the results show that it achieves 1.5X branch coverage (on average) compared with the default AFL, and 1.3X branch coverage compared with AFLNET and StateAFL, using the typical implementations such as OpenSSL, LibreSSL, and Live555. The StateFuzzer identifies a new memory corruption bug in Live555 (2021-08-25) and 14 distinct discrepancies based on differential testing.

1. Introduction

With the rapid development of computer networks, more applications are integrated into the network applications. Protocols play an essential role in cyberspace, as the carrier of various network transmissions. The logic errors in the design process and implementation bugs lead to vulnerabilities, which result in a significant harm, that is, the heartbleed vulnerability of OpenSSL [1], CCS injection vulnerability [2], and Server Message Block protocol vulnerability (CVE-2020-0796).

Several automated software testing techniques have been proposed to find vulnerabilities. Compared with the symbolic execution and code auditing, fuzzing is one of the most efficient techniques for detecting security vulnerabilities in real-world software due to the fact that it is user-friendly and efficient. However, several challenges exist for fuzzing on servers, that is, protocol implementations.

In contrast to regular programs, protocol implementations process inputs according to the basic state model, which determines the processing logic of all the interactive messages. The American Fuzzing Loop (AFL) [3] and LibFuzzer [4] are popular tools belonging to the coverage-based grey-box fuzzing (CGF). They tackle the stateless programs and concrete functions without an in-depth interaction. The previously described single-input fuzzing is considered as stateless fuzzing. The stateful black-box fuzzing tools (SBF), such as Peach [5] and Boofuzz [6], are other well-known techniques. In the approaches, the effectiveness depends on the given state machine, which is obtained from the RFC specification or the captured traffic data.

The protocol state fuzzing [7] is another branch of protocol testing. It first learns the state machine of the protocol implementation by black-box testing and then finds the suspicious logic by comparing the learned model and the specification or analyzing the differences between several different versions. However, it can only find the logical errors by manual comparison, while lacking the ability to discover crashes such as buffer overflow vulnerabilities.

Hence, it is important to combine the state machine and grey-box fuzzing. The existing approach consists in

dynamically constructing the state machine while fuzzing, which helps generate the complete state model, that is, AFLNET [8]. However, the state model is inaccurate, which leads to interesting tests loss and redundant tests. Hence, the grey-box fuzzing based on the learned model from active automata learning is performed.

Most of the protocol fuzzing tools only tackle the server programs, while ignoring the client testing. Memory bugs and semantic errors exist in the client. Fiterau-Brostean [9] applies protocol state fuzzing to the sliding window behavior of TCP, while the learned model through active learning is an abstract model, which may not cover the abnormal behavior. The Secure Copy Protocol (SCP) client cannot verify that the object returned by the SCP server corresponds to what was requested, which results in the malicious manipulation of the server or man-in-the-middle attacker (CVE-2019-6110). Hence, it is important to fuzz the client-side programs of the application layer protocols.

The contributions of this paper are summarized as follows:

(1) A model-based grey-box fuzzing framework, which consists of automata learning and state-aware grey-box fuzzing, is proposed.

(2) StateFuzzer (https://gitee.com/z11panyan/state fuzzer.git), which can fuzz the server-side and client-side implementations of the application layer protocols, is implemented.

(3) The experimental results of the open-source SSL library (OpenSSL and LibreSSL) show that the tool StateFuzzer can achieve 1.5X code coverage (on average) compared with the default AFLNWE and 1.3X code coverage compared with AFLNET. In addition, the effects based on SMTP and RTSP are compared, and a new memory bug and an undisclosed vulnerability are found.

The remainder of this paper is organized as follows. The existing technologies are classified in Section 2. The motivation is introduced in Section 3. The proposed method is detailed in Section 4. The experimentation and evaluation are presented in Section 5. The related studies are discussed in Section 6. Finally, the conclusion and perspective are drawn in Section 7.

2. Background

In the absence of the naming convention and classification of the existing technologies, this paper attempts to classify them according to the different emphases and targets of the current technologies.

2.1. Software Testing and Fuzzing. Software testing, which is the technique of verifying the correctness and errors determination of the program according to rules, can be divided into specification-based and code-based testing [10]. According to Tretmans [11], if the implementation $\mathcal{I}$ conforms to the rules $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$, the relationship between the rules and the implementation can be written as $\mathcal{I}\ imp\ \mathcal{R}$. Otherwise, the implementation violates the rules, and it is denoted by $\mathcal{I}\ \overline{imp}\ \mathcal{R}$. The rules can be divided into specification-based and code-based rules, denoted by $\mathcal{R} = \mathcal{S} \bigcup \mathcal{C}$. The code-based rules are basic program rules. For instance, the use-after-free, buffer overflow, and double-free are strictly forbidden. The specification-based rules are extracted from request for comments (RFC).

Fuzzing is one of the most efficient techniques of software testing. A test suite T that consists of test cases generated from the rules $\mathcal{R}$ is tested as the input of implementation in order to discover abnormal behaviors. A test case t is a pair input, output, where t_{in} = input and t_{out} = output. According to the rules, the expected output of the input is $\mathcal{R}(\text{input})$, while the output on the implementation is $\mathcal{I}(\text{input})$. The test cases conforming to the rules can be written as $T_{\mathcal{R}} = \{t | t_{\text{out}} = \mathcal{R}(t_{in})\}$. Hence, fuzzing can be described as $\exists t \in T_{\mathcal{R}}, \mathcal{I}(t_{in}) \neq t_{\text{out}}$, then $\mathcal{I}\ \overline{imp}\ \mathcal{R}$. Similarly, fuzzing can be divided into specification-based and code-based. As for the code-based fuzzing, the output of the rules can be defined as no crash, namely $T_{\mathcal{C}} = \{t | t_{\text{out}} = \text{no crash}\}$. The generation of the input is critical. For protocol fuzzing, some analysts focus on the generation of the single message, and the others pay more attention to stateful fuzzing.

In the specification-based fuzzing, the specification $\mathcal{S}$ is extracted from RFC. In addition, it is nontrivial to obtain the output of t from RFC. Hence, protocol state fuzzing and differential testing are proposed to reduce the number of test cases.

Protocol state fuzzing is also referred to as learning-based testing or model-based testing. The protocol state fuzzing consists in first inferring a state machine from the protocol implementation based on active automata learning. The state machine is then checked against the specification. The details are provided in Section 2.2.

Due to the fact that it is difficult to check if $t \in T_{\mathcal{S}}$, differential testing is one of the approaches used to find the inconsistencies. Given two implementations $\mathcal{I}$ and $\mathcal{I}'$, if $\mathcal{I}(t_{in}) \neq \mathcal{I}'(t_{in})$, the test case $t_{in}, \mathcal{I}(\text{input})$ should be analyzed.

The learning-based fuzzing [27] can be considered as the combination of two methods. It first learns a hypothesis model by active automata learning and then tries to find the inputs that reveal nonconformance between another implementation $\mathcal{I}'$ and the hypothesis. Hence, the learning-based fuzzing is considered as differential testing, in which the generator of test cases is based on the state machine.

In summary, the existing approaches are divided into specification-based and code-based fuzzing (cf. Table 1). Note that the related work in Section 5 is reviewed according to this classification.

2.2. Active Automata Learning. Active automata learning is an active method of model learning. It is one of the most efficient algorithms for inferring the model of the black-box system. The MAT [40] is a widely used active learning framework. It includes a learner, who only knows the input and output symbols of the system under learning (SUL), and a teacher, who knows all the information of the target

Table 1: Classification of the existing protocol fuzzing techniques.

Specification-based fuzzing	Protocol state fuzzing	C&C [12], TLS [7], TCP [9], SSH [13], OPENVPN [14, 15], QUIC [16], IPSec [17], DTLS [18], SFADiff [19], MPInspector [20]	Learning-based-fuzzing [27]
	Differential testing	Certificate verification [19–23], HVLearn [24], TLS-diff [25], NEZHA [26]	
Code-based fuzzing	Focus on input	TLS-attacker [28], Jero [29], Miff [30], MQTT [31], GANFuzz [32], Hfuzz [33], SeqFuzzer [34], Snipuzz [35]	
	Focus on state	Chen (TLS) [36], AFLNET [8], StateAFL [37], Profuzzbench [38], TCP-fuzz [39], proposed method	

system. The learner learns the unknown model by querying the teacher. More precisely, the learner proposes a membership query (MQ) by sending a message sequence to SUL. If SUL accepts it, the teacher returns "yes". Otherwise, it returns "no". The learner then tries to construct an automaton (a.k.a. hypothesis) based on the learning algorithm and submits it to the teacher. The teacher can judge whether the behavior of the automaton matches the target system. Otherwise, the teacher gives a counterexample, which is referred to as an equivalence query (EQ).

The mealy machine is one of the most common models. It can be defined as a 4-tuple $M = (S, s_0, \delta, \lambda)$ based on I and O, where $I = \{i_1, i_2, \cdots, i_n\}$ is the finite set of input symbols, $O = o_1, o_2, \cdots, o_n$ represents the finite set of output symbols, S denotes the finite set of states, $s_0 \in S$ is the initial state, δ: $S \times I \longrightarrow S$ represents the state transition, and λ: $S \times I \longrightarrow O$ denotes the output function. They can be written as $\delta(s_0, i_1) = s_1$ and $\lambda(s_0, i_1) = o_1$. In the initial state, the state transition and output function can be written as $\delta(i_1) = s_1$ and $\lambda(i_1) = o_1$, respectively. Protocol implementations can be abstracted into mealy machines, where the messages sent by the client and server can be simplified as the input and output symbols.

In order to apply this technology to the realistic system with a large number of inputs and outputs, Aarts [41] added a component "mapper" into the MAT framework. The mapper is located between the learner and SUL and plays the role of abstraction and concretion. More precisely, the learner sends an abstract symbol to the mapper which converts it into a specific message based on the input alphabet and sends it to SUL. Simultaneously, the mapper converts the response back to an abstract symbol and returns it to the learner. Finally, a hypothesis model $\mathscr{H}$ is equivalent to the implementation.

Given the previously mentioned statements, protocol state fuzzing can be divided into two stages. The first stage is the generation of $T_{\mathscr{H}} = \{t | t = t_{in}, \mathscr{S}(t_{in})\}$ based on active automata learning, where t_{in} is a sequence of symbols. In the second stage, if $\exists t \in T_{\mathscr{H}}, s.t. \mathscr{S}(t_{in}) \neq t_{\text{out}}$, the test case triggers a semantic bug.

3. Motivation

Based on the previous analysis, this paper pays more attention to stateful fuzzing. The program is an exhaustive state machine with a large state space and a big input/output alphabet. Thus, it is challenging to explore the whole state space. A fuzzer tries to sample from "interesting" regions of the state space as efficiently as possible. The analysts can abstract the large state set into a smaller set which is supposedly separated by certain operations, as shown in Figure 1(a). The fuzzer can explore edge tuples on this state machine [42]. The feature is more prominent in the protocol implementations. Therefore, the analysts attempt to fuzz protocol implementations based on the state machine.

The state-of-the-art state-guided protocol fuzzing approach, namely AFLNET, constructs the state machine based only on the server responses, which is not the case with the mealy machine. If a new status code exists in the server response, a new state is added. Figure 1(b) presents the state machine of Live555 obtained from AFLNET, where a graph node represents a new state, marking the states with the status code, and the state with the label "0" is the initial state.

AFLNET relies on status codes from messages, leading to interesting tests loss and redundant tests. Considering the sequence "200-404-454" as an example, it means that an input sequence, of which the response sequence is "200-404-454", exists. If the response sequence is "454–405", which does not exist in Figure 1(b), it is considered as an interesting transition, and the relative input sequence is added to the seed pool. Because the sequence "200-200" exists, the response sequence "200-200-200" is not interesting. However, the input sequence of "200-200" may be "DESCRIBE-SETUP" and that of "200-200-200" may be "DESCRIBE-SETUP-PLAY." The sequence "200-200-200" should be regarded as an interesting sequence. In addition, the input sequence of the response sequence "200-200-200" may be "DESCRIBE-SETUP-PLAY" or "DESCRIBE-SETUP-SETUP", which is not distinguished in AFLNET.

The state machine in Figure 1(c) is the mealy machine learned by active automata learning. The model considers the inputs and outputs, which is coherent with the programmer's logic. Moreover, the issues in AFLNET can be avoided when considering the inputs. As previously discussed, "DESCRIBE-SETUP-PLAY" is considered an interesting transition. Furthermore, the same output "200" of different inputs "SETUP" and "PLAY" has different meanings, which can be distinguished by the mealy machine. Other protocols, such as SMTP, have the same problem.

In summary, the expressiveness of the mealy machine is better than that of AFLNET. This paper takes advantage of the coverage and state transition in order to guide the fuzzing on top of active automata learning.

4. Methodology

Based on the previous analysis, a model-based grey-box fuzzing framework is proposed (cf. Figure 2). A high-level

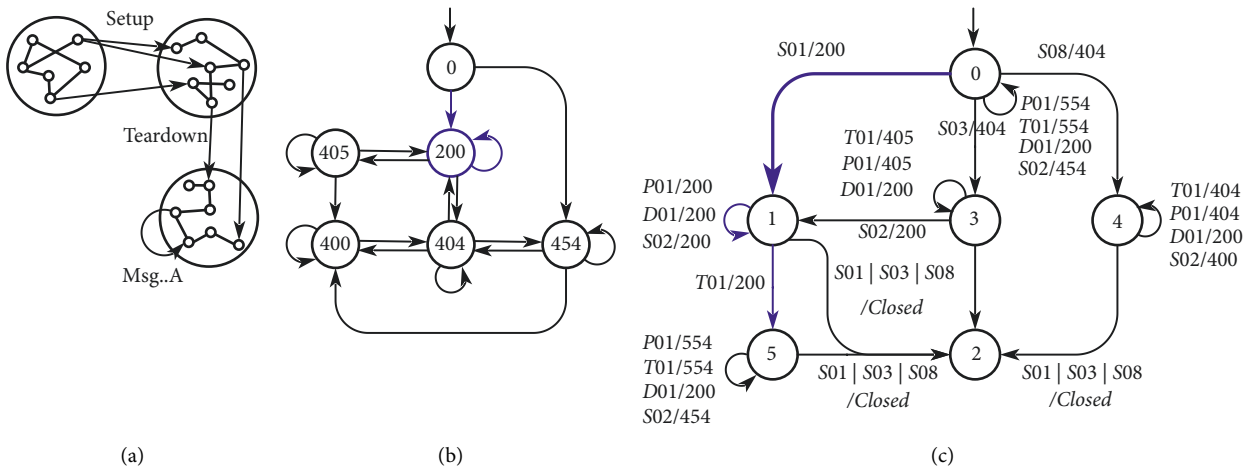

Figure 1: The different models based on AFLNET and active automata learning.

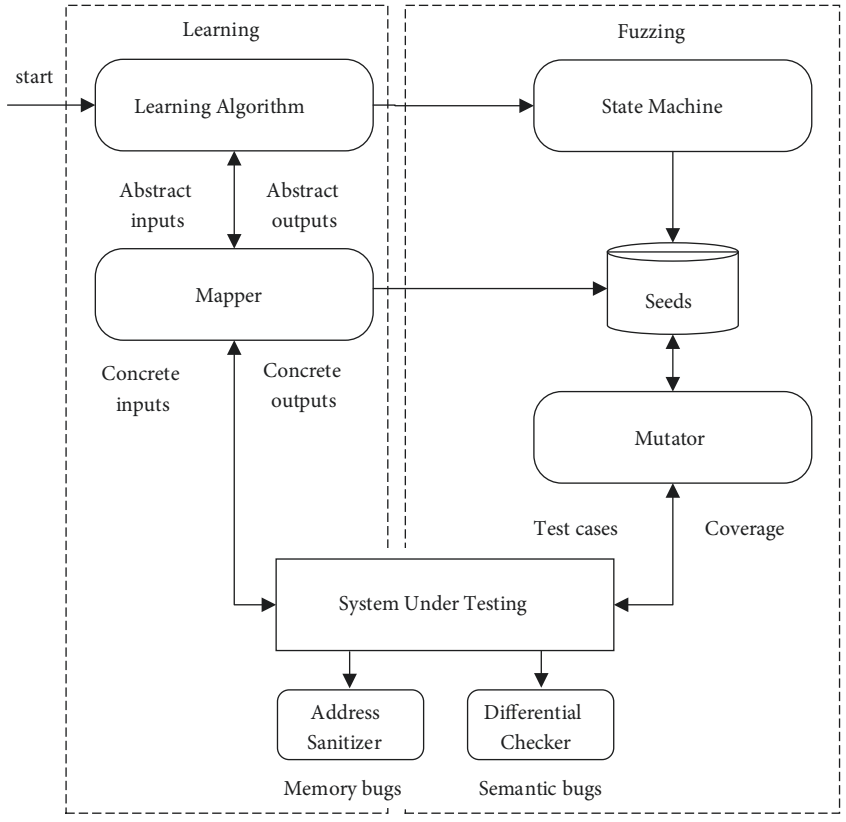

Figure 2: Model-based grey-box fuzzing framework.

overview of the proposed method is first provided, followed by a detailed description. The approach is divided into two stages: learning and fuzzing. The target of the learning process is the state machine based on given alphabets. Due to protocol systems feature, it is necessary to construct a mapper between the learner and SUL, also referred to as system under testing (SUT).

At the stage of fuzzing, the test cases are generated based on the state machine and mutation, and the weights of seeds are adjusted by the code coverage. The test cases causing crashes and coverage increases are exported to a specified structure. In addition, a differential checker is provided to identify semantic bugs.

4.1. Learning Phase. The learning component reuses the StateLearner tool. It learns the state machine by calling the interface of LearnLib, which provides different learning algorithms, such as LStar [43] and TTT [44], and the equivalence query algorithms, such as w-method and

modified w-method. The mapper, also referred to as the test harness, is the core of the StateLearner. The test harness plays a crucial role as a stateless client, which converts symbols of the given alphabet to concrete messages, and sends them in order.

Different methods exist for building the mapper for encryption protocols and plaintext protocols. For encryption protocols, the encryption and decryption components should be manually adjusted in order to realize the mapper [7]. As for plaintext protocols, the keywords representing the protocol state can be extracted from the existing data packets, based on protocol reverse analysis. The mapper can be constructed according to the keywords and their corresponding messages, where the keywords are abstract symbols and messages are concrete inputs.

Afterwards, considering the RTSP protocol as an example, the mapper construction is explained. The keyword is the first field of the message separated according to the space character. The alphabet is composed of all the unique keywords, and the concrete inputs are related messages, as shown in Table 2. The first 3 bytes of the response data are extracted as abstract outputs. Especially, if the response times out, the output is set to "empty". Once a connection is closed, all outputs returned afterwards will be the same (referred to as connection closed).

The state machine of the RTSP implementation can be inferred based on this mapper. The state machine is represented by a structure array, which contains three elements: a state identification, the outputs of all the symbols in the alphabet, and the target state of all the symbols. Based on the state machine, the output and the target state of each symbol in each state can be obtained.

For different protocols, the Request Sequence Parser component in AFLNET uses protocol-specific information of the message structure to extract status codes. Similarly, we can construct the mapper as described previously, which has the same expansibility for other protocols as AFLNET.

4.2. Grey-Box Fuzzing. Due to the complexity of the protocol message structure, the critical points of grey-box fuzzing are the mutation and the scheduling algorithm based on the coverage and the state machine. The primary problem consists in how to use the state machine. Not only states but also the state transitions highly affect the execution of protocol implementations, as stated by Zou et al. [39]. The interesting transitions are obtained from the state machine based on the breadth-first search, as shown in Algorithm 1. The inputs of the algorithm are the model, its initial state *initS*, and the sink state *sinkS*. The output and target state of each symbol on each state can be obtained based on the model. The output of any symbol on the sink state is "connectionclosed". The lists of states, input symbols, and output symbols, recorded as state$L$, *in*L, and out$L$, form a structure to represent the path. The target state and output of each symbol are obtained on the current path. If the target state is already in the state list or the target state is the sink state, then the path is added into the set. Otherwise, it is pushed into the queue. In addition, the output is the set of interesting paths, in which every path contains the lists of states, input symbols, and output symbols. Simultaneously, the mapper stores the correspondence between the symbol and the message. For each path, a standard message sequence denoted by $M = m_1 \cdot m_2 \cdot m_3$ can be obtained.

Table 2: The alphabet and mapper of RTSP.

Alphabet (symbols)	Concrete inputs (messages)
Describe	DESCRIBE XX aacAudioTest...
Setup	SETUP XX aacAudioTest/track1...
Play	PLAY XX aacAudioTest...
Pause	PAUSE XX aacAudioTest
Teardown	TEARDOWN XX aacAudioTest...

```
(1) Input: Model, initS, sinkS
(2) stateL, inL, outL = ∅; currS = initS
(3) Queue.add (stateL, inL, outL)as
(4) curr ⟵ Queue.poll
(5) While (true)
(6)    for each a: symbols do
(7)       tarS, output = Model (curr, a)
(8)       if output == connect closed
(9)       continue
(10)      end if
(11)      in L+ = a, out L+ = output
(12)      if tar S ∈ state L ‖tar S = sink S
(13)         sL+ = tarS
(14)         paths+ = stateL, inL, outL
(15)      else
(16)         state L+ = tar S
(17)         Queue.add (stateL, inL, outL)
(18)      end if
(19)   end for
(20)   curr ⟵ Queue.poll
(21)   if curr == null then break
(22) return paths
```

Algorithm 1: Get path.

The scheduling strategy of fuzzing is based on two assumptions: (i) the "deeper" the network communication, the more likely an error exists; (ii) the more code edges are covered, the more likely an error exists. It is important to mention that different paths reaching the same state may execute different codes. Hence, a hierarchical scheduling strategy is designed. The seeds are classified according to the paths, as shown in Figure 3. A seed pool is constructed for each path.

In order to determine the initial weight of every path and every seed, every standard message sequence is mutated as a seed. After a certain number of mutations, the initial score and seeds can be obtained. This is referred to as the initialization phase, as stated in Algorithm 2. The random phase based on the hierarchical scheduling strategy is then executed. When selecting the seed, it first determines the path according to their weights and then chooses the seed from the pool of the path. In addition, the interesting sequence is added to the seed pool with the related weight, and the weight of the seed pool is increased.

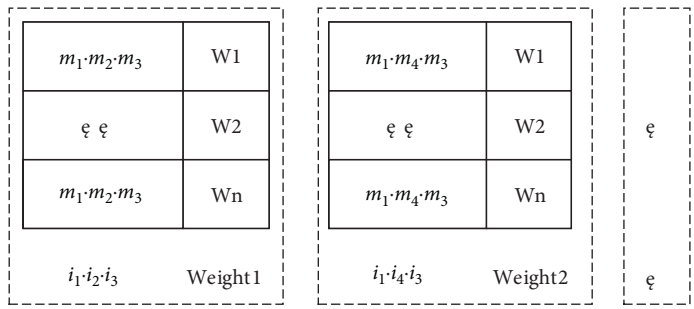

Figure 3: The hierarchical scheduling strategy. The dotted box represents a seed pool determined by the path, such as $i_1 \cdot i_2 \cdot i_3$. The seed pool consists of interesting messages that are assigned weights based on the code coverage.

```
(1) Input: paths
(2) for each path ∈ paths do
(3)     m1 · m2mn · · · ⟵ path.inList
(4)     for each i in (1, n)do
(5)         M = m1 · m2 · ·mutate (mi) · ·mn
(6)         coverage, outList = exec (M)
(7)         if interesting then
(8)             update (path.score)
(9)             path.messageList.add (M)
(10)        end if
(11) end
```

Algorithm 2: Initialization phase.

In contrast to fuzzing with a single input, protocol fuzzing requires deep interaction. In other words, the test cases are sequences of multiple messages. Given a seed $M = m_1 \cdot m_2 \cdot m_3$, its index is first randomly selected, such as the second message m_2. The mutated message sequence is $M' = m_1 \cdot mutate(m_2) \cdot m_3$. Most of the mutation strategies, such as bitflip, shuffling, erasing, swapping, inserting, and splicing, are implemented. The ordered message sequence is then regarded as a single seed, and a message is treated as a byte. The mutation strategies include splicing between sequences and disordering within a single sequence.

4.3. Client Side. Several open-source implementations of protocols include both the server-side and client-side functions. The client-side vulnerabilities are often overlooked, which leads to hazards. The client-side is tested in a similar way to the server-side. The tool needs to interact with the client by simulating the server. It can construct different types of response packets, rather than implementing the complete logic of the server-side.

The difference between fuzzing on the server-side and client-side is that the client to be tested should connect to the simulated server actively. The tested code heavily relies on the client's request. Considering RTSP as an example, different codes are executed when requesting to play different types of files. Hence, as many types of the request as possible should be tested.

At the learning stage, the input and output symbols are the opposite of the servers. The same three-byte status codes may have different meanings. For instance, "200" in RTSP may be the status code of DESCRIBE or SETUP. When fuzzing the client, the status codes are the input symbols. Hence, they should be marked with the meaning, as shown in Section 5.1.

At the fuzzing stage, the process is similar to that of the server-side, without the more specific process to change.

4.4. Differential Checker. Except for memory bugs, it is worth focusing on semantic bugs in protocol implementations. Semantic bugs refer to the conflicts between the implementation and RFC specification. Most of the fuzzers (such as AFLNET) detect memory bugs based on sanitizers that are powerless to detect semantic vulnerabilities. Differential testing is used to detect the differences among the protocol implementations, due to their diversity. The semantic errors are further detected by manual analysis.

Based on the idea of differential testing, a differential checker component is designed to help discover inconsistencies in different implementations. In fact, it is meaningless to compare the responses of the implementations because they contain timestamps and random fields. Hence, a response is abstracted as a symbol based on the mapper. Inevitably, some subtle inconsistencies may be lost. However, this avoids the large-scale analysis of the RFC documents and reduces the difficulty and cost of the manual analysis.

Timestamps, random, and some nondeterministic fields exist in the responses. It is crucial to define a reasonable metric that can assess whether the responses are discrepant or in agreement. The TLS-diff defines the reduction function that maps a TLS implementation's response [25]. The responses of ClientHello are divided into Handshake and Alert based on the reduction function. In this paper, a response is abstracted as a symbol based on the mapper.

The detailed algorithm of the testing phase is similar to the fuzzing stage. However, their output standards are not the same. The outputs of the differential checker are the test cases leading to the different responses.

In order to avoid the excessive duplication of differences due to the same root cause, the deduplication strategy is used. That is, if the sets of basic blocks triggered by two test cases are the same, the two test cases are considered as duplicates [45].

5. Experimentation and Evaluation

The StateFuzzer is developed based on Java, while the fuzzing component is common for different protocols. For different types of protocols, the learning component requires a customized development. The code coverage is obtained based on LLVM. According to LibFuzzer, which has a simple code coverage instrumentation built-in (SanitizerCoverage), the source code of protocol implementations is slightly modified slightly.

The experimental environment and results are detailed in the sequel. All the experiments are performed on an Ubuntu server (16.04 LTS) with 4 CPUs and 8 GB RAM.

5.1. Experimental Setup. We performed experiments on ProFuzzBench, a public benchmark for network fuzzers [38]. The target programs are slightly modified in the previously described manner and compiled based on CLANG. The latter will feedback on the code coverage to guide the selection and mutation of seeds. After fuzzing, the recorded test cases are sent to the standard program compiled by GCC with the "-ftest-coverage" parameter. The results of the lines and branches coverage are simultaneously generated. In particular, the corresponding relationships between the acronyms and meanings are l_abs (lines absolute), l_per (lines percentage), b_abs (branches absolute), and b_per (branches percentage), and abbreviations are used in the following text.

The efficiency is evaluated by comparing three baseline approaches, a stateless coverage-guided fuzzer (AFLNWE (https://github.com/profuzzbench/aflnwe Pham ported the)) and two grey-box stateful fuzzers (AFLNET and StateAFL). The TLS encryption protocol as well as the SMTP and RSTP plaintext protocols are used. The target implementations are OpenSSL3.0.0(0437435a), OpenSSL1.0.1f (0d877634), LibreSSL3.2.1, Exim (Version 4.98), and Live555 (0.92). For the sake of reducing the influence of initial conditions as much as possible, the same seeds as AFLNET are used. The used parameters of automata learning are "LStar" and "Modified W-method".

In order to learn the model of the three protocols, the alphabets are defined as follows:

(1) TLS server: ClientHello (RSA and DHE), Certificate (RSA and empty), ClientKeyExchange, ClientCertificateVerify, ChangeCipherSpec, Finished, and ApplicationData (regular and empty)

(2) TLS client: ServerHello (RSA and DHE), Certificate (RSA and empty), CertificateRequest, ServerKey Exchange, ServerHelloDone, ChangeCipherSpec, Finished, and ApplicationData (regular and empty)

(3) RTSP server: OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, and PAUSE

(4) RTSP client: DESCRIBE (200), SETUP (200), TEARDOWN (200), PLAY (200), Session_NotFound (454), Stream_NotFound (404), Method_NotAllowed (405), and BadRequest (400)

(5) SMTP server: HELO, RSET, MAIL, RCPT, DATA, and QUIT

Due to the fact that the three baseline approaches do not support the fuzzing of client programs, the experimental results are divided into two parts for analysis. The effects of fuzzing only on the server-side programs are compared, and the gains from fuzzing on the client-side programs are analyzed, respectively. For most of the implementations, the growth in code coverage leveled off after 4 hours of fuzzing. Each fuzzing tool on each target program can be reached for 4 hours, and the experiment is repeated 4 times to establish the statistical significance of the results. The follow-up figures and tables are based on the average values using the above experimental method.

When analyzing the gains from client-side fuzzing, the server and client are tested for 2 hours. In order to obtain the coverage information, the recorded inputs of fuzzing the server and the client are replayed. Since OpenSSL, LibreSSL, and Live555 are also available as clients, the coverage of the server and client can be put together.

5.2. Fuzzing Performance. Figure 4 presents the average percentage of branches and lines covered by AFLNWE, AFLNET, StateAFL, and StateFuzzer within four hours for four repetitions on OpenSSL3.0.0 (0437435a). It can be deduced from the obtained results that three stateful network fuzzers show an evident increase in branch and line coverage, while the stateless fuzzer has a moderate efficiency. When fuzzing only the server, the effect of our method is slightly better than AFLNET and StateAFL.

Figure 5 shows the lines percentage for each implementation and each fuzzer after 4 hours. As for the TLS (OpenSSL and LibreSSL) and SMTP (Exim), a significant improvement in code coverage exists with the three stateful fuzzers, while the four fuzzers cover similar lines and branches for the RTSP (Live555). The reason is that when all the messages are sent in one packet, the server of Live555 can handle it normally. StateAFL eliminates the need for protocol-specific parsing, while its effect is the worst of the three stateful fuzzers. Especially, as for the encryption protocol, StateFuzzer has a better effect than the other two stateful tools, due to the encryption and decryption processes being realized in order to interact with the target program more deeply.

Moreover, the lines covered by StateFuzzer and AFLNET are compared in detail. It can be seen from Figure 6 that for OpenSSL3.0.0, Exim4.98, and Live555 (0.92), most of the lines covered by the two methods are the same. Considering OpenSSL3.0.0 as an example, there are 9503 lines covered together, while StateFuzzer covers 471 additional lines and AFLNET covers 87.

When introducing the client fuzzing, more covered paths that the server fuzzing cannot cover exist. In Figure 5, the grey color above the red color represents the increased lines covered by the client fuzzing. The code coverage is highly increased by fuzzing on the client-side program. Since the implementation "Exim" does not have a client-side functionality, the grey color does not exist. Table 3 illustrates

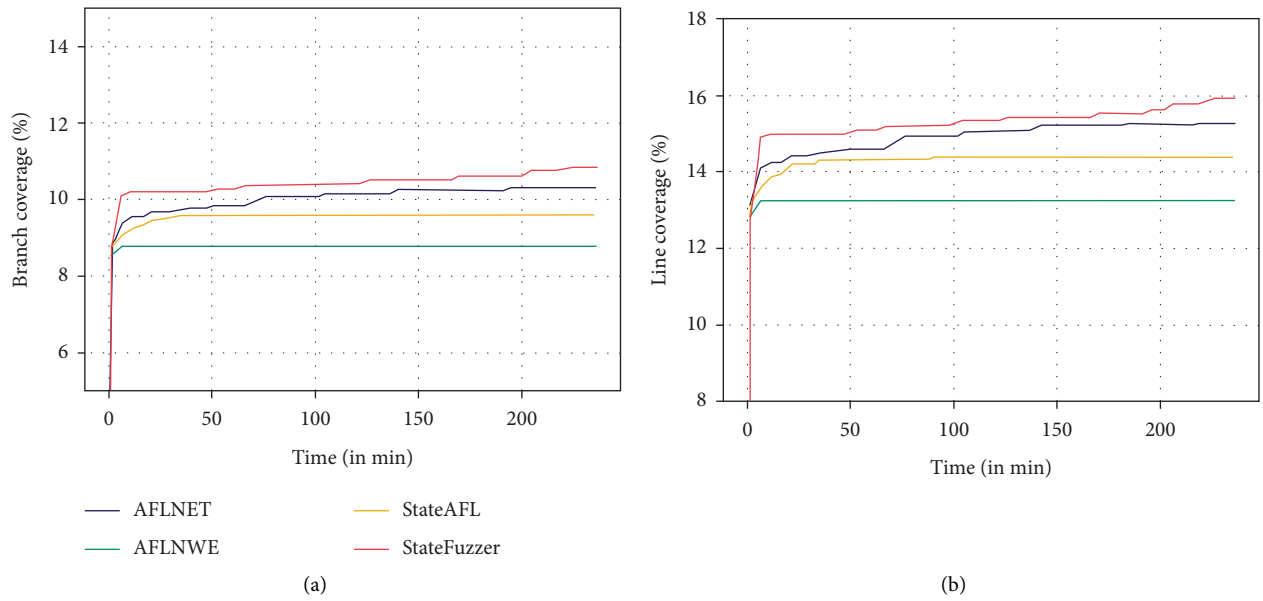

Figure 4: Average percentage of branches and lines covered by AFLNET, AFLNWE, StateAFL, and StateFuzzer within 4 hours for 4 repetitions on OpenSSL3.0.0 (colored). (a) Branch coverage overtime (%). (b) Line coverage overtime (%).

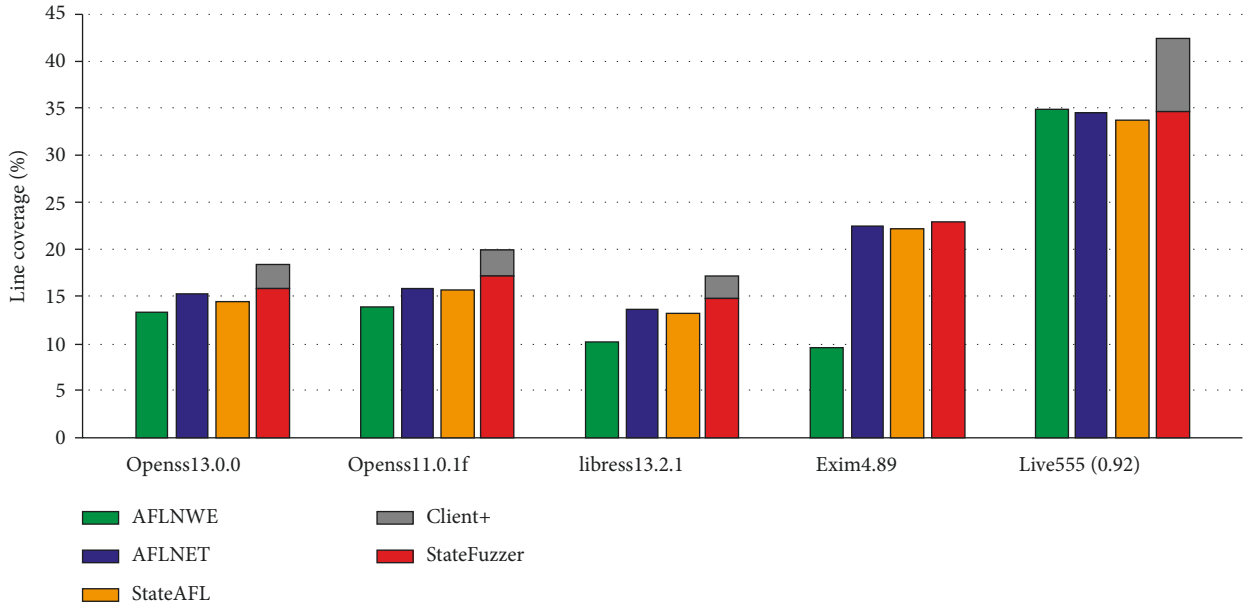

Figure 5: Line coverage after 4 hours of fuzzing on different implementations (colored).

the average growth rate of four implementations. For OpenSSL3.0.0, 1.49X branch coverage (on average) is achieved compared with the default AFLNWE, while 1.27X code coverage is achieved compared with AFLNET. The branch coverage increases by 85% and 30% compared with AFLNWE and AFLNET on Libressl3.2.1, respectively. For Live555 (ceeb4f46), a 34% increase in branch coverage exists.

Besides, a comparison between the black-box and grey-box is performed. Table 4 illustrates that the effect of grey-box fuzzing is better than that of the black-box which lacks coverage guidance.

5.3. Memory Bug Discovery. In order to compare the vulnerability discovery capacities between the four tools, this paper focuses on Live555 (ceeb4f46) and collects the vulnerabilities triggered by sending exception packets. Four known vulnerabilities meeting the requirements exist as follows: CVE-2020-24027, CVE-2021-38381, CVE-2021-39282, and CVE-2021-38383. The results show that all the fuzzers discover four known vulnerabilities. Simultaneously, a new crash is detected by StateFuzzer. After manually analyzing it, the vulnerability also exists in its latest version, and it is submitted to the vendor for patching. The root cause of the

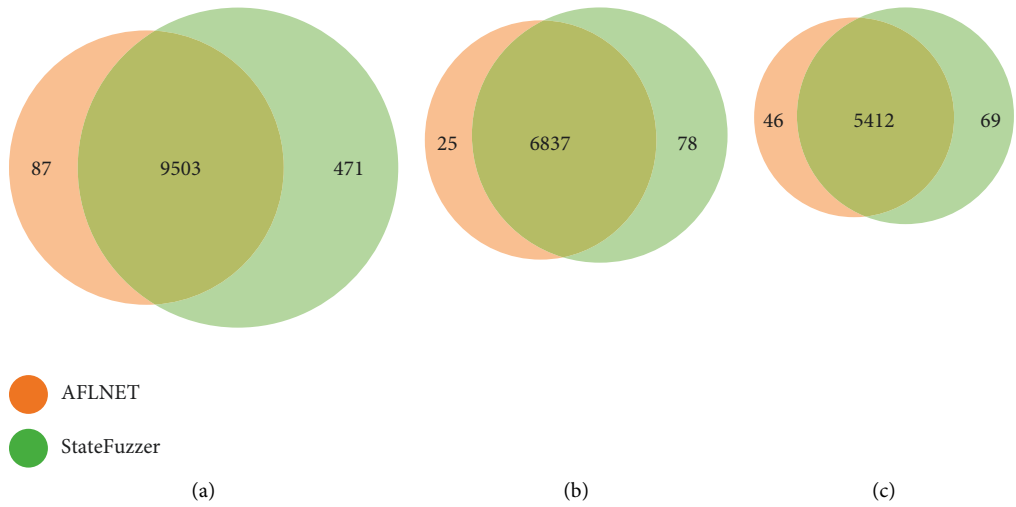

Figure 6: The lines covered by StateFuzzer and AFLNET (colored). (a) OpenSSL3. 0.0. (b) Exim4.98. (c) Live555(0.92).

Table 3: The average growth rate of coverage compared with AFLNET and AFLNWE within 4 hours.

Improvement	vs AFLNWE		vs AFLNET	
	Branch (%)	Line (%)	Branch (%)	Line (%)
OpenSSL3.0.0	49	38	27	20
OpenSSL1.0.1f	46	43	27	26
LibreSSL3.2.1	85	70	30	27
Live555–0.92	33	21	34	22

Table 4: The coverage of the black-box and grey-box within 4 hours.

Percent (%)	Black-box		Grey-box	
	b_per	L_per	b_per	L_per
OpenSSL3.0.0	10.35	15.25	10.85	15.95
OpenSSL1.0.1f	10.75	16.3	11.2	17.0
LibreSSL3.2.1	10.15	14.0	10.5	14.6
Exim4.89	15.35	22.65	15.75	23.0
Live555–0.92	20.7	33.8	21.85	34.575

vulnerability is analyzed. It is deduced that the order of its packets is specific. If the client sends a package including PLAY and SETUP after sending SETUP and PLAY several times, a heap-use-after-free bug exists when processing the Matroska file. In addition, there should be a specific time interval between the packets, which is difficult to trigger. Due to the random nature of the mutation, not every fuzz will trigger the vulnerability. It takes almost about three hours to activate it, while it only takes 15 minutes to trigger CVE-2021-38381 and CVE-2021-39282. Moreover, an undisclosed stack buffer overflow, which was patched in 2018.10.17, is found.

Furthermore, for Exim4.89 (SMTP), StateFuzzer is the only fuzzer able to discover CVE-2017-16943 and to provide a new PoC test case.

5.4. Inconsistencies. Large differences exist between the different versions. Considering Exim4.93 and Exim4.94 as an example, HELO is required at the beginning of the normal functional logic in Exim4.94, while it is unnecessary in Exim4.93, which results in different state machines. Essentially, Exim4.93 violates the rule "The first command in a session must be the HELO command" in RFC821. Hence, several relatively recent versions with the same state machine are selected, such as OpenSSL3.0.0 vs. OpenSSL1.1.1 h (f123043) vs. LibreSSL3.2.1, Exim4.93 vs. Exim4.89, and Live555 (0.92) vs. Live555 (1.02). Finally, some meaningful test cases are manually analyzed.

5.4.1. OpenSSL3.0.0 vs. OpenSSL1.1.1 h (f123043) vs. LibreSSL3.2.1. There are four discrepancies between OpenSSL and LibreSSL when processing the ClientHello. They make inconsistent judgments of the content fields in parsing some extensions. For instance, for parsing renegotiation extensions, OpenSSL only parses the length field, while LibreSSL makes further judgments about subsequent bytes. In addition, StateFuzzer can discover deep inconsistencies that require interactions. For example, OpenSSL3.0.0 will immediately alert when receiving an invalid Diffie-Hellman client key exchange, while LibreSSL3.2.1 and OpenSSL1.1.1 h do not immediately respond, which is not clearly defined in RFC5246. The latter only explicitly stipulates that "in any case, a TLS server MUST NOT generate an alert if processing an RSA-encrypted premaster secret message fails", for the RSA-encrypted premaster secret message.

Simultaneously, the client-side programs are analyzed based on differential testing. For instance, there are two version fields in ServerHello. The first invalid version field is ignored in OpenSSL, while it results in an alert in LibreSSL. Similarly, deeper inconsistencies can be detected in the client-side programs. The processing of LibreSSL is different from that of OpenSSL (https://github.com/openssl/openssl/issues/4320), for the negative serial number in the certificate. It is considered as the illegal extra padding in OpenSSL, while it passes the verification in LibreSSL.

5.4.2. Exim4.93 vs. Exim4.89. An inconsistency exists in Exim4.89. Any BDAT command sent after the BDAT LAST is illegal and MUST be replied to with a 503 "Bad sequence of commands" reply code, as described in rfc3030. Exim4.89 ignores the specification, which results in the CVE-2017-16943.

5.4.3. Live555 (0.92) vs. Live555 (1.02). Because the known vulnerabilities cause the program to crash, they also cause differences between the two implementations.

To sum up, 14 distinct discrepancies are discovered in the experiments (cf. Table 5). Some discrepancies are caused by implementation errors, and some are caused by the implementers' different understanding of RFC. The test cases that caused crashes or discrepancies are provided in the supplementary material.

5.5. Discussion. The proposed method combines the advantages of active automata learning and grey-box fuzzing. The experiment results demonstrate that the model-based grey-box fuzzing is valuable. It highly contributes to the exploration of more paths within a limited period, owing to the more accurate abstract model based on active automata learning and fuzzing for the client. Similar tools such as Boofuzz and Peach exist. They require a lot of effort to manually construct the input and code the state machine. Compared with them, the automation level is improved by using active automata learning.

This section focuses on the three studies that are also critical reference objects of the presented work. The first method is the learning-based fuzzing which combines automata learning and fuzz testing. Its target is the black-box system, and it focuses on inconsistencies among multiple implementations. In addition, the instrumentation feedback is not leveraged, which allows the fuzzer to distinguish the inputs that execute new branches from the inputs that do not reach a new code.

The other two methods are the grey-box fuzzing tools for network protocols on the top of AFL. AFLNET, StateAFL, and the proposed method belong to two different technical roadmaps. Compared with the traditional fuzzing tools, AFLNET uses state feedback and coverage feedback to guide the mutation of seeds, while treating the message sequences as the fuzzing input to enable deep interaction with protocol implementations. Moreover, StateAFL [37] analyzes the states through the memory characteristics by inserting probes on memory allocations and network I/O operations. It increases the degree of automation by eliminating the need for relying on detailed manual analysis of protocols. However, it is necessary to make specific modifications for special memory operations in the source code.

Compared with AFLNET and StateAFL, StateFuzzer requires more understanding and analysis to construct the test harness for different protocols. If the test harness for the protocol is developed, different implementations of the given protocol can be learned and fuzzed, which is the focal point of our future work. In addition, due to the lack of research on the client-side programs, the client programs are fuzzed based on the presented framework, and a differential checker is designed to improve the efficiency of finding semantic bugs.

Table 5: Distinct discrepancies of three protocols.

Discrepancies	TLS1.2	SMTP	RTSP
Server	5	2	5
Client	2	—	0

Besides, the proposed method only focuses on the state machine of the protocol. Because the protocol messages are highly structured inputs, it is crucial to pay more attention to how to mutate a single message, which is analogous to format-aware fuzzing. Fioraldi et al. [46] divide the techniques of format-aware fuzzing into grammar-based, where the inputs comply with a language grammar, and chunk-based, where the inputs are represented by a tree hierarchy with C structure-like data chunks to form individual nodes. The fuzzing of text protocol is suitable for the grammar-based technique, while the binary protocol corresponds to chunk-based ones. In future work, we aim at extending the format-aware fuzzing to protocol fuzzing [47].

6. Related Work

6.1. Fuzzing. Fuzzing is one of the research hotspots in vulnerability mining, which automatically constructs invalid inputs and sends them to the target software. The generation of test cases and the guidance strategy of feedback are two key points. According to the feedback provided by the runtime program, it is usually divided into black-box, grey-box, and white-box. The black-box fuzzing does not require any feedback data from the target program. It pays more attention to the mutation methods, such as input structure-based mutation [6] and generation strategy based on deep learning [47]. The white-box fuzzing consists in constructing test cases based on the internal logic of the programs and maximizing the code coverage by using dynamic symbol execution [48, 49] and taint analysis [50]. SAGE [51] is one of the representative tools. However, the preconditions and complexity of the white-box fuzzing are relatively high. The grey-box fuzzing mainly focuses on code coverage (basic blocks, paths, functions, etc.) and data flow. Popular tools, such as AFL and LibFuzzer, obtain the code coverage at runtime through code instrumentation (i.e., source code and binary).

Researchers have summarized the open challenges and opportunities for fuzzing [52, 53]. The related works for testing on protocols are presented in the sequel.

6.2. Code-Based Fuzzing. Due to the features of protocols, the methods of seed construction and stateful fuzzing are vital points of protocol fuzzing. As for the test cases, Walz et al. [25] propose a tree structure, referred to as the general message tree (GMT), in order to describe the specific TLS messages (ClientHello). The GMT and the message can be converted to each other to mutate the TLS messages efficiently. In addition, researchers try to generate test cases

based on deep learning, such as Seq2Seq [34] and GAN [32]. For the sake of TLS protocol, Somorovsky [27] implements an open-source and extensible TLS protocol fuzzing framework by designing all the protocol fields as the variables. This method can dynamically construct TLS messages and TLS records. In addition, due to the wide use and insecurity of IoT protocols, more researchers focus on the fuzzing of IoT. For instance, Chen et al. [30] obtain the protocol fields of interest by analyzing the code of the IoT application and mutating the relative fields to fuzz the IoT device. Similarly, Snipuzz [35] is a black-box fuzzing tool for IoT firmware by inferring and mutating message snippets.

Format awareness can boost CGF. In fact, it is worthwhile to draw lessons from the techniques of format-aware fuzzing and apply them to the mutation of protocol messages. Aschermann et al. [54] take advantage of the description-based input generation and feedback-based fuzzing to improve the efficiency of generation. Blazytko et al. [55] propose the GRIMOIRE synthesizing structure while fuzzing. GRIMOIRE reduces the interesting input to the fragment which causes a new coverage and generates the new inputs by recursive replacement and fragment splicing. Pham et al. [56] construct a virtual structure to represent the input file. The files are decomposed into fragments based on the specification. The fragments are internal nodes of the virtual structure. Moreover, they define innovative mutation operators that work on the virtual structure and convert them into files after mutation. Gopinath et al. [57] infer the syntax of strongly formatted input using dynamic control flow analysis. The obtained grammars are well-structured and very readable.

With respect to stateful fuzzing, Chen et al. [36] set the forkserver point at the state switching point of the test program. Simultaneously, a queue array of state test cases is maintained, and the corresponding test case data packets are sent when testing different states. The test program automatically forks the program according to the data package and simultaneously calculates the related parameters such as the code coverage. However, this method requires a very detailed analysis of the source code. TCP-Fuzz [39] uses a transition-guided fuzzing approach that exploits a novel coverage metric as program feedback, referred to as branch transition.

6.3. Specification-Based Fuzzing

6.3.1. Protocol State Fuzzing. Cho [12] first applied automata inference to control and command protocols. The researchers then began to test with various protocols as the target. Aiming at bridging the gap between active learning and real-world systems, Aarts implements TOMTE on the basis of LearnLib. It formally describes the abstract and concrete behavior of the mapper and implements it in TOMTE. Consequently, Ruiter et al. [7] model a state machine of the TLS protocol implementation based on the active learning method and manually analyze the generated state machine to find logical vulnerabilities. More protocols, such as TCP [9], SSH [13], OpenVPN [14, 15], QUIC [16], IPSec [17], and DTLS [18], are analyzed to compare the state machines with the protocol specification.

6.3.2. Differential Testing. The idea of differential testing is applied in different phases. For instance, Brubaker et al. [21, 22, 23] analyze the certificate verification algorithm in SSL/TLS implementations based on differential testing. HVLearn focuses on hostname validation [24], while TLS-diff pays more attention to the TLS handshake process [25]. TCP-Fuzz compares the outputs of multiple TCP stacks for the same inputs, in order to dig semantic bugs based on a differential checker [39].

7. Conclusion

This paper proposed a novel strategy for stateful protocol fuzzing, termed model-based grey-box fuzzing. A state machine is inferred based on active automata learning, and test cases are generated according to the state machine and seed pool. StateFuzzer is implemented on top of StateLearner, and the method is applied for fuzzing on implementations of three protocols, such as OpenSSL, LibreSSL, Exim, and Live555. Compared with AFLNET and StateAFL, the proposed method achieves higher lines and branches coverage in the same span of time, especially with the introduction of client-side fuzzing. In addition, differential testing is introduced to detect inconsistencies between implementations. Both the server-side and client-side programs can be analyzed based on the differential checking.

In future work, we aim at extending the test harnesses for different protocols, such as FTP and DTLS, in order to further verify the effectiveness of the proposed method and expose previously unknown bugs. For the TLS protocols, this paper only studied the code about tls1.2. An intensive study of tls1.3 and tls1.1 is then of our interest. Finally, we also expect to perform the interaction with encrypted data.

Data Availability

The source code data and experimental results used to support the findings of this study have been deposited in the Git repository, https://gitee.com/z11panyan/state-fuzzer.git.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Authors' Contributions

Pan Yan developed the theory and methodology, provided the software, and wrote the original draft. Zhu Yuefei supervised the study. Liang Jiao performed validation. Lin Wei performed investigation.

Acknowledgments

This work was supported by the National Key Research and Development Project of China (2019QY1300). The authors would like to express their gratitude to EditSprings (https://www.editsprings.cn) for the expert linguistic services provided.

References

[1] *OpenSSL Heartbleed Vulnerability*, CVE-2014-0160, 2013.

[2] *OpenSSL CCS Injection Vulnerability*, CVE-2014-0224, 2013.

[3] in*American Fuzzy Lop*, Jul. 12, 2016, http://lcamtuf.coredump.cx/afl/.

[4] in*libFuzzer a library for coverage-guided fuzz testing*, Jul. 13, 2016, https://github.com/enovella/libfuzzer-workshop.

[5] *Peach fuzzing platform*, 2021, https://www.peach.tech.

[6] Boofuzz, *A fork and Successor of the Sulley Fuzzing Framework*, 2017, https://github.com/jtpereyda/boofuzz.

[7] J. d. Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *Proceedings of the 24th USENIX Security Symposium*, pp. 193–206, USENIX Association, Washington, August 2015.

[8] V. T. Pham, Marcel, and A. R. AFLNet, "A greybox fuzzer for network protocols," in *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation*, Testing Tools Track, Porto, Portugal, October 2020.

[9] P. Fiterău-Broştean and F. Howar, "Learning-based Testing the Sliding Window Behavior of TCP Implementations," in *Proceedings of the International Workshop on Automated Verification of Critical Systems*, Turin, Italy, September 2017.

[10] P. Jorgensen, *Software Testing: A Craftsman"s Approach*, CRC Press, Florida, United States, 1995.

[11] J. Tretmans, *Model Based Testing with Labelled Transition Systems*, Springer-Verlag, Berlin, 2008.

[12] C. Y. Cho, D. Babic, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pp. 426–439, ACM, Chicago Illinois USA, October 2010.

[13] P. Fiterau-Brostean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of SSH implementations," in *Proceedings of the 24th Acm Sigsoft International Spin Symposium on Model Checking of Software*, pp. 142–151, Assoc Computing Machinery, New York, 2017.

[14] L. A. Daniel, E. Poll, and J. Ruiter, "Inferring OpenVPN State Machines Using Protocol State Fuzzing," in *Proceedings of the 2018 IEEE European Symposium on Security and Privacy Workshops*, pp. 11–19, IEEE, London, UK, April 2018.

[15] Y. Z. Shen, C. X. Gu, X. Chen, X. L. Zhang, and Z. Y. Lu, "Vulnerability analysis of OpenVPN system based on model learning," *Ruan Jian Xue Bao/Journal of Software*, vol. 30, no. 12, pp. 3750–3764, 2019, (in Chinese).

[16] A. Rasool, G. Alpár, and J. d. Ruiter, *State Machine Inference of QUIC*, 2019.

[17] J. Guo, C. Gu, X. Chen, and F. Wei, "Model Learning and Model Checking of IPSec Implementations for Internet of Things," *IEEE Access*, vol. 7, p. 1, 11/26 2019.

[18] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. d. Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS Implementations Using Protocol State Fuzzing," in *Proceedings of the presented at the 29th USENIX Security Symposium*, Boston, MA, 2020.

[19] G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. K. SFADiff, "Automated Evasion Attacks and Fingerprinting Using Black-Box Differential Automata Learning," in *Proceedings of the in the 2016 ACM SIGSAC Conference*, Vienna Austria, October 2016.

[20] Q. S. J. Wang, T. Tian, X. Zhang, B. Zhao, Y. Kan, and Z. Lin, "MPInspector: a systematic and automatic approach for evaluating the security of IoT messaging protocols," in *Proceedings of the 30th USENIX Conference on Security Symposium*, 2021.

[21] C. Brubaker, "Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, May 2014.

[22] Y. T. Chen and Z. D. Su, "Guided differential testing of certificate validation in SSL/TLS implementations," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the Acm Sigsoft Symposium on the Foundations of Software Engineering*, pp. 793–804, Assoc Computing Machinery, New York, August 2015.

[23] C. Tian, C. Chen, Z. Duan, and L. Zhao, "Differential testing of certificate validation in SSL/TLS implementations," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 4, pp. 1–37, Oct 2019.

[24] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. J. HVLearn, "Automated black-box Analysis of hostname verification in SSL/TLS implementations," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, May 2017.

[25] A. Walz and A. Sikora, "Exploiting dissent: towards fuzzing-based differential black-box testing of TLS implementations," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 2, pp. 278–291, 2020.

[26] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. J. Nezha, "Efficient domain-independent differential testing," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, pp. 615–632, New York, May 2017.

[27] B. K. Aichernig, E. Muskardin, and A. Pferscher, "Learning-Based Fuzzing of IoT Message Brokers," in *Proceedings of the ICST*, Porto de Galinhas, Brazil, April 2021.

[28] J. Somorovsky, "Systematic fuzzing and testing of TLS libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1492–1504, Vienna Austria, October 2016.

[29] S. Jero, M. L. Pacheco, D. Goldwasser, and C. Nita-Rotaru, "Leveraging textual specifications for grammar-based fuzzing of network protocols," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 9478–9483, 2019.

[30] K. Chen, C. Song, L. M. Wang, and Z. Xu, "Using memory propagation tree to improve performance of protocol fuzzer when testing ICS," *Computers & Security*, vol. 87, p. 13, Nov 2019.

[31] S. Hernández Ramos, M. T. Villalba, and R. Lacuesta, "MQTT security: a novel fuzzing approach," *Wireless Communications and Mobile Computing*, vol. 2018, pp. 1–11, 2018.

[32] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "GANFuzz: a GAN-based industrial network protocol fuzzing framework," in *Proceedings of the 15th ACM International Conference*, Ischia Italy, May 2018.

[33] X. Liu, B. Cui, J. Fu, and J. Ma. HFuzz, "Towards automatic fuzzing testing of NB-IoT core network protocols implementations," *Future Generation Computer Systems*, vol. 108, 2019.

[34] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. H. SeqFuzzer, "An industrial protocol fuzzing framework from a deep learning perspective," in *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 59–67, IEEE, Xi'an, China, April 2019.

[35] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, and D. Liu, "Snipuzz: black-box fuzzing of IoT firmware via message snippet inference," in *Proceedings of the 28th ACM Conference on*

Computer and Communications Security (CCS), Virtual Event Republic of Korea, November 2021.

[36] Y. Chen, T. Lan, and G. Venkataramani, "Exploring effective fuzzing strategies to analyze communication protocols," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, pp. 17–23, London United Kingdom, November 2019.

[37] R. Natella, *StateAFL: Greybox Fuzzing for Stateful Network Servers*, 2021, https://arxiv.org/abs/2110.06253.

[38] R. Natella and V.-T. Pham, "ProFuzzBench: a benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual, Denmark, November 2021.

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

[40] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987/11/01/1987.

[41] F. Aarts, *Tomte: Bridging the gap between Active Learning and Real-World Systems*, 2014, https://www.semanticscholar.org/paper/Tomte-%3A-bridging-the-gap-between-active-learning-Aarts/be280ee2af4be98d547f86adf43538416130c411.

[42] C. Aschermann, S. Schumilo, A. Abbasi, and T. H. Ijon, "Exploring deep state spaces via fuzzing," in *Proceedings of the IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2020.

[43] M. Isberner, B. Steffen, and F. Howar, "LearnLib tutorial, Runtime Verification," in *Lecture Notes in Computer Science*, E. Bartocci and R. Majumdar, Eds., vol. 9333, pp. 358–377, Springer International Publishing Ag, Cham, 2015.

[44] M. Isberner, F. Howar, and B. Steffen, "The TTT algorithm: a redundancy-free approach to active automata learning," in *Proceedings of the Runtime Verification*, Toronto, ON, Canada, September 2014.

[45] Y. Pan, W. Lin, Y. He, and Y. Zhu, "Coverage-guided differential testing of TLS implementations based on syntax mutation," *PLOS ONE*, vol. 17, no. 1, ArticleID e0262176, 2022, in press.

[46] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: automatic grey-box fuzzing for structured binary formats," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis,* Virtual Event, USA, July 2020, Available.

[47] M. Zakeri Nasrabadi, S. Parsa, and A. Kalaee, "Format-aware learn&fuzz: deep test data generation for efficient fuzzing," *Neural Computing and Applications*, vol. 33, 06/13 2020.

[48] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzing Symbolic Expressions," in *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 711–722, IEEE, Madrid, ES, May 2021.

[49] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," *Proceedings of the 2019 Network and Distributed System Security Symposium*, vol. 19, pp. 1–15, 2019.

[50] P. Chen and H. C. Angora, "Efficient fuzzing by principled search," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, May 2018.

[51] P. Godefroid, M. Y. Levin, and D. Molnar, "Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[52] M. Boehme, C. Cadar, and A. Roychoudhury, "Fuzzing: challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021.

[53] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys*, CSUR, 2022.

[54] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. T. Nautilus, "Fishing for Deep Bugs with Grammars," in *Proceedings of the NDSS*, San diago, California, April 2019.

[55] T. Blazytko, "GRIMOIRE: synthesizing structure while fuzzing," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, pp. 1985–2002, CA, USA, August 2019.

[56] V.-T. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, p. 1, 2020.

[57] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 172–183, Virtual Event USA, November 2020.