Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2024) 1–19

Procedia Computer Science

Protocol Fuzzing Corpus Composition with Universal Automata Model

anonymous submission

Abstract

Discovering vulnerabilities in network service is of great significance. Currently, coverage-guided fuzzing(CGF) is widely regarded as the most effective method. The initial corpus is a set of valid input examples used to initiate the fuzzing process. However, the efficiency of CGF depends on the quality of initial corpus. Constructing high-quality initial corpus is challenging, as it typically requires manual efforts to understand the implementation details and corresponding protocol specifications, making it difficult to generalize across different protocol implementations.

To generate high-quality corpus tailored to the service under test(SUT), this paper proposes a protocol-independent automatic generation method. The paper models protocol input based on the state variables of SUT, introduces the protocol-generic SVRM state machine model, and utilizes active learning algorithms to automatically construct this model. By analyzing the minimum spanning tree of this model, we achieve the automatic generation of high-quality corpus that adapts to the SUT.

We present SVRM-Learner for modeling and SVRM-Composer for generating high quality corpus. These tools are used to generate adaptive corpus for six targets of six different protocols in ProFuzzBench. Compared to the corpus provided by ProFuzzBench, the corpus generated by our system improve the state coverage of modern protocol fuzzers by 37.1% and discover known real protocol vulnerabilities at a speed 2.47x faster.

© 2011 Published by Elsevier Ltd.

Keywords: Grey box Fuzzing, State variable, Fuzzing Corpus

1. Introduction

Network protocols define the rules for information exchange between network entities. Such software has a simple and direct attack surface—attackers can disrupt the reliability, stability, and availability of network service simply by sending malicious packets. For example, the "Heartbleed" vulnerability in early versions of OpenSSL can leak sensitive information from memory[1], and vulnerabilities in Microsoft's SMB protocol can lead to remote code execution[2]. These vulnerabilities pose a threat to the privacy and property security of billions of users on the Internet. The easiness of carrying out attacks on network protocols, coupled with their widespread use, makes vulnerabilities in such software highly damaging.

Fuzzing is an effective method for discovering vulnerabilities in network service(i.e. protocol implementations). In the early days, researchers propose protocol fuzzers based on protocol specifications [3, 4]. These tools generate malformed test cases based on designed templates and send them to the service under test(SUT). They automatically discover vulnerability by monitoring the program's abnormal behavior during execution. Recently, researchers start to introduce coverage information to improve protocol fuzzers[5–7].

Compared to black-box fuzzers, coverage-guided fuzzing (CGF) does not require designed templates to genarate test cases. It only needs the instrumented target and network traffic as initial test cases. By analyzing the coverage

information of the target, it retains test cases that reach new code after mutation, thereby achieving automatic exploration of the unknown code space. It is generally believed that the higher the state coverage and code coverage achieved by fuzzers, the greater the probability of discovering vulnerabilities in SUTs.

CGF tools' state and code coverage depend on the quality of the initial corpus [8, 9]. The initial corpus serves as the starting point for fuzzing and guides the testing path of fuzzers. A low-quality corpus can lead to blind mutation, generating useless test cases, while a high-quality corpus should enable fuzzers to reach more code paths, thus exploring the different execution scenarios of the target more comprehensively. Several factors of corpus can affect the efficiency of CGF on network service, including ❶ lack of organization according to protocol specification, ❷ high redundancy of test cases in the corpus, ❸ excessively large test cases, and ❹ incomplete coverage of message types. To build a high-quality corpus, testers need to be familiar with the protocol specifications and implementation details of the SUT. They should design message sequences in a suitable order or select representative test cases from the SUT's network traffic. However, the specifications and implementations of different protocols vary widely, requiring testers to spend a considerable amount of time learning them. Therefore, testers need an automated method to build high-quality corpus for the SUT.

Driven by the above-mentioned issues, this paper aims to design a protocol-independent method for automatically constructing high-quality corpus for protocol fuzzers. To address issues ❶, on the one hand, we need to find common programming features of network service and model the input pattern of the SUT based on these features (Challenge 1). To address issues ❷ and ❸, on the other hand, guided by the characteristics of high-quality corpus, we aim to automatically generate a tailored corpus based on the input model of the SUT (Challenge 2).

For Challenge 1, we propose a **UNIVERSAL state machine model** for protocol implementation — the SVRM model (SVRM means programs' State Variable and Representative Message) — and implements a prototype modeler called SVRM-Learner. It is based on the insight that most protocol implementations use state variables (Special variables globally defined in program) to control protocol states. SVRM-Learner transforms the SUT into the minimal adequate teacher(MAT) of the SVRM model by turning state variable value information and message instances into input and output symbols. Then it uses active learning algorithms to construct the SVRM model of the SUT.

For Challenge 2, we propose a **corpus generation algorithm** based on the directed graph analysis of SVRM model and implement an automatic high-quality corpus generation tool named SVRM-Composer. By transforming the features of a high-quality corpus into graph features, we transform the problem of generating high-quality corpus into the problem of traversal on directed cyclic graph. By generating the minimum spanning tree of the directed graph, we use a greedy algorithm to find the shortest edge sequences that can traverse all state transitions and message types, thereby constructing a high-quality corpus adapted to the SUT.

In summary, we make the following contributions:

1. We propose SVRM model for protocol implementation and develop a modeling tool SVRM-Learner, which can construct protocol-independent models for multiple protocol implementations with little human-effort.

2. Based on the characteristics of high-quality corpus and the SVRM model, we propose a corpus generation algorithm and implement a tool called SVRM-Composer based on the analysis of the minimum spanning tree of the model, which automatically generates high-quality corpus adapted to the SUT.

3. We evaluate SVRM-Learner and SVRM-Composer using ProFuzzBench. The experimental results show that our method can automatically construct high-quality corpus adapted to six different protocol implementations. Compared to the corpus provided by ProFuzzBench, the corpus generated by our method can improve the state coverage of the CGF fuzzer AFLNET by 37.1% and discover known real protocol vulnerabilities at a speed 2.47x faster.

2. Background

2.1. Protocol Modeling

Constructing and analyzing the state machine model of the SUT can help testers understand the dynamic behavior of the protocol and predict system responses. In computation theory, a deterministic finite automaton (DFA) is a computational model that determines the next state based on the current state and input. Since network service needs to

provide correct responses according to current client interaction state and requested messages, following the protocol specifications (such as RFC documents), it is naturally suitable to model input using DFA. The methods for constructing DFA of the target can be divided into passive learning methods and active learning methods. Passive learning methods directly analyze the network traffic of the target to construct the DFA. For example, PULSAR[10] and other works [11] use Markov models or passive learning algorithms to analyze the network traffic of the target and use the obtained model for black-box fuzzing. However, the accuracy of the model largely depends on the quality of the network traffic. Active learning methods turn the input and response of the target through into defined symbols, construct the corresponding MAT, and use an iterative process of equivalence queries [1] and membership queries [2] to construct the DFA of the target. For example, STATELEARNER[12] implements the MAT of the TLS protocol and uses active learning algorithms to construct the Mealy automaton model of multiple TLS protocol implementations, discovering semantic vulnerabilities that violate the TLS protocol specifications. However, these methods are protocol-dependent, and testing new protocol implementations requires deep customization of the tools. Therefore, this paper proposes a protocol-independent state machine model to address the limitations of these methods.

2.2. Protocol Coverage-guided Fuzzing

Coverage-guided fuzzing uses runtime code coverage or state coverage information to guide the selection and scheduling of test cases in fuzzing, giving fuzzers the ability to automatically explore untested code. AFLNET[5] is the first fuzzer to apply code coverage and state coverage information to fuzzing on protocol implementations. The current progress in protocol coverage-guided fuzzing is as follows. Some CGF tools improve fuzzers throughput by modifying the way test cases are passed, such as message synchronization mechanisms[13, 14], snapshot mechanisms[15, 16], and desocket methods[17]. Other tools introduce state feedback mechanisms to supplement insufficient utilization of code coverage information by parsing protocol response codes[5, 6], long-lived memory[7], and state variables[13, 18]. AFLNET-LEGION[19] further investigates using Monte Carlo tree search methods to improve AFLNET's state selection algorithm. To generate more effective mutations, some tools introduce sequence mutation[5], state key-field mutation[18], or mutation templates generated based on large language models (LLM)[20]. However, there is still a lack of research on how to automatically generate high-quality corpus conducive to CGF.

2.3. Protocol Fuzzing Corpus Characteristics

In the corpus used for CGF, test cases consist of unilateral messages sent by clients to SUTs. The length of the message sequence, the number of test cases, and the types and arrangement of messages in the message sequence all affect the efficiency of fuzzing. If the message sequence is too long, it will significantly prolong the testing time and reduce the probability of producing favorable mutations. If there are too many message sequences, it will bring a large overhead to the corpus selection process of fuzzers. **Our comparative experiments on corpus** (see Appendix Appendix A) show that whether the message types involved in the test cases are comprehensive and whether they are arranged according to the protocol specifications also affect the code coverage and state coverage of fuzzers. The results also indicate that fuzzing is not suitable for uncovering new states of SUT. These insights into corpus characteristics will serve as our guidance for automatically generating higher-quality initial corpus.

3. Design

To construct the protocol-independent input model based on programming features and generate a high-quality corpus to improve code coverage and state coverage, this paper **(1)** proposes the SVRM[3] model based on state variables and representative messages. We **(2)** implement the method for constructing SVRM model's MAT (Minimum Adequate Teacher) for active learning algorithms. Based on SVRM model and characteristics of high-quality corpus proposed in Section 2.3, we **(3)** design a target-adaptive corpus generation algorithm. The entire automatic corpus generation system consists of the following parts (As shown in Fig. 1)

[1] i.e. Given a sequence, what is the corresponding response sequence

[2] i.e. Whether the current constructed hypothesis model and the actual model of the target are equivalent

[3] i.e. SVRM means State Variable - Representative Message

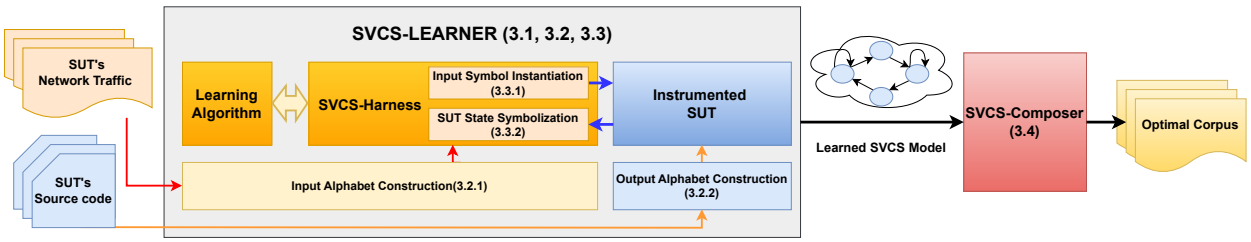


Figure 1: Workflow of SVRM-Model-Based Auto Network Service Corpus Composing System

- Generalized SVRM Model Learner (SVRM-Learner): This paper introduces the mathematical definition and representation of the SVRM model for protocol implementation (Section 3.1), the method for constructing input & output alphabet (Section 3.2), and the construction of the minimal adequate teacher(MAT) for the SVRM model of the SUT (Section 3.3).
- Auto Corpus Generation Engine (SVRM-Composer): SVRM-Composer transforms the characteristics of high-quality corpus into directed graph features of the SVRM model, and constructs a high-quality corpus by parsing and expanding the minimum spanning tree based on these features (Section 3.4).

3.1. Definition of Protocol Implementations' SVRM Model

According to whether the response is determined by the current input, the state machine can be divided into Mealy machine and Moore machine. Thus the SVRM model of the protocol implementation can be represented by the six-tuple model of the Mealy machine $\mathcal{M}$:

$$\mathcal{M} = (S, s_0, \Sigma, \Lambda, \mathcal{T}, O) \tag{1}$$

Where, S represents **the finite set of states** of the SUT. s_0 represents the initial state of the SUT, corresponding to the state when communication with the client is just established. Σ represents the **finite set of input symbols** of the SVRM model (also known as the input alphabet), which satisfies the following properties:

$$\Sigma = \{\sigma \mid \forall m \in \mathrm{M}, \exists f \in \mathcal{F}, f : \sigma \mapsto m, f^{-1} : m \mapsto \sigma\} \tag{2}$$

Where, M represents the set of **representative message instances** of the SUT. A representative message instance is a typical message instance representing the same type of message in the protocol, which should cover the similar function of SUT, and can be distinguished according to control segment or other characteristics. The definition means that each input symbol σ in the input alphabet Σ is a one-to-one mapping of each message instance m in the set of representative message instances M.

Λ represents the finite set of output symbols of the SVRM model (also known as the output alphabet), which satisfies the following properties:

$$\Lambda = \{\lambda \mid n_i \in \mathrm{N}, \lambda \in \{\mathrm{val}(n_1) \times \mathrm{val}(n_2) \times \cdots \times \mathrm{val}(n_i)\} \cup \{\lambda_0\}\} \tag{3}$$

Where N represents the set of **state variables** of the SUT. State variables are program variables used to represent protocol interaction states with clients. Val(n) represents the set of values of state variable n. This definition means that each output symbol λ in the output alphabet Λ is an element of the union of the Cartesian product of the set of values Val(n) for each state variable n of the target, and a special output symbol λ_0. **The special output symbol** λ_0 is used to reflect the target being in a special state, the function of which will be explained in detail in section 3.3.

$\mathcal{T}$ is the **set of state transition functions** of the SVRM model, where each function is defined as $\mathcal{T} : S \times \Sigma \to S$, which means the next state transition of the SVRM model is determined by the current state and the current input symbol. O represents **the set of response functions** of the SVRM model, where each function is defined as $O : S \times \Sigma \to \Lambda$, which means the current response of the SVRM model is determined by the current state and the current input symbol. The goal of automaton learning is to infer S with $\mathcal{T}$ and O.

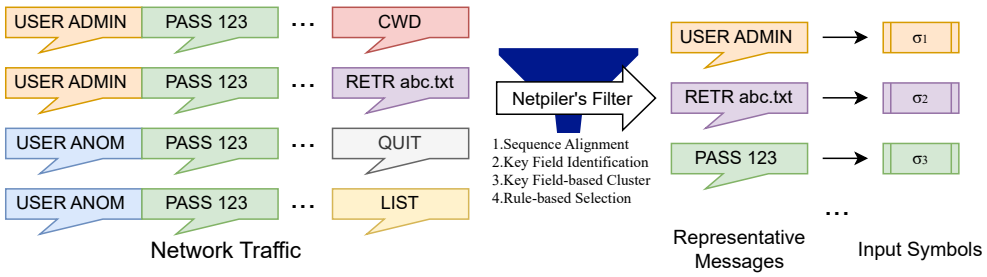


Figure 2: Input Alphabet Construction workflow. We cluster network traffic by Netpiler and select representative message from clusters as SVRM model's input symbol.

3.2. Alphabet Construction of SVRM Model

To construct the input alphabet Σ and output alphabet Λ for the SVRM model of the SUT, we first obtain the set of representative message instances M and the set of state variables N according to the definition in section 3.1. Then, the input and output alphabets are constructed through the process in section 3.2.1 and section 3.2.2. A construction example is provided in Appendix C.1

3.2.1. Input Alphabet Construction with Representative Message

To construct a representative message set, we first need to cluster existing network traffic and select suitable messages from the clustered samples to serve as representative messages. The process is shown in Fig.2.

Under the condition of having sufficient traffic samples (approximately 500-1000 entries), we use methods proposed by Netplier [21], which use sequence alignment and key field identification, to cluster the network traffic. In cases where the traffic samples are extremely limited (fewer than 10 entries), clustering can be performed manually (Based on field semantics and duplicated fields). [4]

For the clustered message samples, a representative message is chosen based on the following criteria:

1. Each cluster should select one or two representative message.
2. The representative message for each cluster originates from the same network traffic entry.

Taking the example of constructing representative message set M for Live555 based on network traffic provided by ProFuzzBench, we can use the control fields of these messages to classify messages, such as SETUP, PLAY, TEARDOWN, etc. It is worth noting that the SETUP message in Live555 has two types: one type includes the session field, while the other does not. We consider these two types of SETUP messages as different message instances and add them to set M.

According to equa.2 , we construct a one-to-one mapping from each element in the set M to the input alphabet Σ, thus obtaining the input alphabet Σ for the SVRM model of SUT.

3.2.2. Output Alphabet Construction with State Variable

To construct the output alphabet Λ for the SUT, we need to obtain the set of state variables N and the runtime values Val(n) of these variables. The process is shown in Fig.3.

To obtain state variables of the SUT, we use automatic static analysis methods[13, 18] to recognize them. In detail, We treat enumerated variables directly as state variables. For non-enumerated variables, we treat variables satisfying the following rule as state variables:[5]

1. the variable's assignment point's operand should be constant.
2. the variable should be involved in conditional expressions.

[4] If we only have one traffic sample, we just regard each message pack iteraction with SUT as representative messages

[5] Some variables can be pruned by semantic according to their name

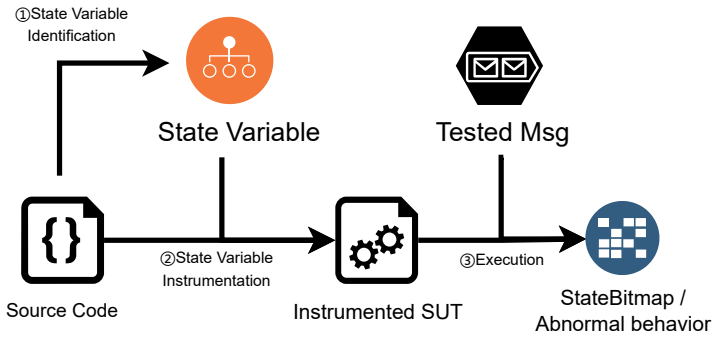


Figure 3: Workflow of Output Alphabet Construction. the output alphabet comes from SUT's state variable's runtime information (State bitmap / abnormal behavior)

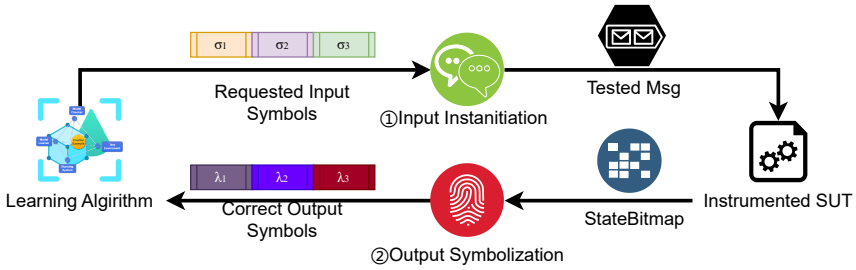


Figure 4: Workflow of Minimum Adequate Teacher for SVRM model. Input Symbol Instantiation transform input symbols into tested network traffic. Output Symbolization fetch correct Statebitmap's hash or abnormal symbol λ_0 as output symbol

To get the runtime values Val(n) of state variable n, we insert instrumentation `__SVRM_hash()` before each assignment point of state variables, as shown in Listing 1 from Appendix C.

In detail, The main function of `__SVRM_hash()` is to collect state variables' unique ID and their current values. Referring to AFL's [22] bitmaps, we implement the "StateBitmap" to record changes in the values of state variables, which is used to generate a global summary of the runtime values of state variable. The instrumentation calculates and uses the hash value of StateBitmap to represent the current values of all state variables, and passes this value to shared memory used for communication with SVRM-Learner. The shared memory also records the number of triggered instrumentation of the SUT, assisting SVRM-Learner in determining a stable hash value(details in section 3.3). Assuming no hash collisions occur, this hash value uniquely corresponds to each element of the Cartesian product of the state variable value sets.

According to equa.3, all possible values of the hash value generated by the instrumentation code constitute the output alphabet λ of the SUT.

3.3. MAT Construction for SVRM Model

We use the L* learning algorithm[23] to construct the SVRM model of the SUT. Thus, we need to implement a MAT (Minimum Adequate Teacher) to provide correct information needed by learning algorithm. The process is shown in Fig.4.

Unfortunately, the hash value of the StateBitmap obtained in section 3.2.2 cannot be directly used as output symbols. On one hand, when the SUT is processing messages, the StateBitmap may become unstable in period. On the other hand, the SUT may also encounter special states such as communication interruption or abnormal termination, where current hash value is meaningless.[6] Therefore, we need to design a converter to obtain the hash value of

[6]the example of such phenomenon is illustrated in Appendix C.2

Algorithm 1: Minimum Adequate Teacher for SVRM model

Input: Instrumented program $\mathcal{P}$, input alphabet Σ, input symbol σ, minimum stable count MIN_STABLE, Maximum unstable count MAX_UNSTABLE

Output: output symbol λ

```
1  Trigger_pre ← 0;
2  unstable ← 0;
3  stable ← 0;
4  msg_instance ← SymbolInstantiation(σ, Σ);
5  SendMessage(P, msg_instance);
6  if SUTCrash(P) ⋁ CommunicateFail(P) then
7      λ ← λ0;
8      return λ ;
9  else
10     repeat
11         Trigger_cur ← GetCurTriggerTime(P);
12         if unstable > MAX_UNSTABLE then
13             error_exit() ;
14         else if Trigger_cur = Trigger_pre then
15             stable ← stable + 1;
16         else if Trigger_cur ≠ Trigger_pre then
17             Trigger_pre ← Trigger_cur;
18             λ ← GetStateHash(P);
19             unstable ← unstable + 1;
20             stable ← 0;
21     until stable > MIN_STABLE;
22     return λ ;
```

the target at a appropriate time and pass the hash value or the abnormal output symbol λ_0 to the learning algorithm according to the running status.

We implement Algorithm 1 to solve this problem, describing how to convert the input symbol requested by the learning algorithm into a message instance, and automatically obtain the correct hash value as output symbol based on the runtime status and instrumentation triggering times.

3.3.1. Input Symbol Instantiation

The process of Symbol Instantiation is converting input symbols sequence requested by learning algorithm into message instances and sending them to SUT.

After obtaining the input symbol σ requested by the active learning algorithm, Algorithm 1 converts the input symbol into a message instance and sends it to the SUT via socket. In the definition of the SVRM model, since each input symbol uniquely maps to a message instance, it only needs to send the corresponding message instance to the SUT using the inverse mapping according to equation 2 (lines 1-4 of Algorithm 1).

3.3.2. SUT State Symbolization

The process of State Symbolization is providing correct output symbol from SUT to learning algorithm.

When processing the current message instance, if the SUT experiences a communication interruption or crashes, the hash value information in the shared memory cannot represent the output symbol of the SUT. we use the abnormal output symbol λ_0 to indicate such state (lines 6-8 of Algorithm 1).

To address the issue of hash values may changing when the SUT processes messages, algorithm 1 sets a fixed threshold (MAX_STABLE and MAX_UNSTABLE) and observes the instrumentation counter. Specifically, after de-

termining that the target SUT has received all response messages from the socket, algorithm 1 will check the instrument triggering times(lines 11 of Algorithm 1), if the instrumentation counter value remains stable until STABLE exceeding the threshold MAX_STABLE, the algorithm considers current hash value as the current output symbol of the SUT (lines 14-15 and 21 of Algorithm 1). If the hash value cannot remain stable for a period of time (when UNSTABLE exceeding the threshold MAX_UNSTABLE), the algorithm determines that the SVRM model constructed is invalid. It then provides an error message, and exits (lines 11-13 and 16-20 of Algorithm 1).

After transforming the SUT into SVRM model's MAT, SVRM-Learner uses the L* algorithm to progressively construct a hypothesis SVRM model for the SUT. In detail, SVRM-Learner directly returns the corresponding output symbol sequence to answer the equivalence queries of the L* algorithm; and the membership queries use the improved W-method[12] to accelerate the query process. When the the learning algorithm have tested all test cases provided by W-method, the algorithm will stop.

In this paper, the hypothesis SVRM model generated in the k-th iteration of the L* algorithm is named k_{th} **iteration hypothesis SVRM model**, and the final SVRM model obtained when the learning algorithm terminates is called the **final SVRM model**. It is observed that some targets may not obtain the final SVRM model of the SUT within the limited time due to the complexity of the target. In such cases, we choose the hypothesis model generated in last iteration as the base model for constructing a high-quality corpus in section 3.4.

3.4. Corpus Composition based on SVRM Model

To generate a high-quality corpus for the SUT, based on the experimental results and fuzzing experience from section 2.3, we formally proposes three key features that the high-quality corpus for protocol fuzzing should have:

1. The corpus should cover all message types supported by the SUT.
2. Message sequences in the corpus should cover all state transitions of the SUT's state machine.
3. While maintaining conditions above, each message sequence in the corpus should be as short as possible, and the total number of message sequences should be minimized.

To satisfy condition 1, the message instances in the representative message instance set M should cover all message types, which relies on the granularity of message clustering method.

To satisfy conditions 2 and 3, we formalize the problem of constructing a high-quality corpus as a traversal problem of the directed graph $\mathcal{G}$ of the SVRM model:

Given the directed graph $\mathcal{G} = (V, E)$ corresponding to the SVRM model $\mathcal{M}$ of the SUT, let the weight of each edge e in $\mathcal{G}$ be 1. From node v_i to node v_j, there exists a set of directed edges sequences denoted as $\Pi^{ij} = \{\pi_1^{ij}, \pi_2^{ij}, ..., \pi_n^{ij}\}$, where each sequence represents a path from node v_i to node v_j. In particular, for any directed edge sequence from the start node v_0 to any non-start node v_j, we denote it as π^j. The weight of an edge sequence π_k^{ij} is denoted as $W(\pi_k^{ij})$, and $W(\pi_k^{ij}) = \left|\pi_k^{ij}\right|$. Here we denote weight of an edge sequences set Π as $W(\Pi)$,where :

$$W(\Pi) = \sum_n W(\pi_n^{ij}) \tag{4}$$

Then the optimal corpus generation problem is equivalent to finding a set of directed edge sequences Π_{min}, which should satisfy:

1. $\forall \pi_k^{ij} \in \Pi_{min}, \pi^{ij} = \pi^j$, the start point of each sequence in the set must be v_0.
2. $\forall e \in E, \exists \pi_k^{ij} \in \Pi_{min}, e \in \pi_k^{ij}$, directed edge sequences set Π_{min} should cover all the directed edges in $\mathcal{G}$.
3. $\forall \pi_k^{ij} \in \Pi_{min}, \forall e \in \pi_k^{ij}, f(e) \in \Sigma$, the set of input symbols mapped by the directed edges, which is involved in all the directed edge sequences in Π_{min}, should be equal to the input alphabet Σ of the model.
4. $\forall \Pi_p \subset \Pi, W(\Pi_p) \geq W(\Pi_{min})$, the weight of Π_{min} is minimal, which means the message sequence of a single seed should be as short as possible, and the total number of seeds in the corpus should be as few as possible.

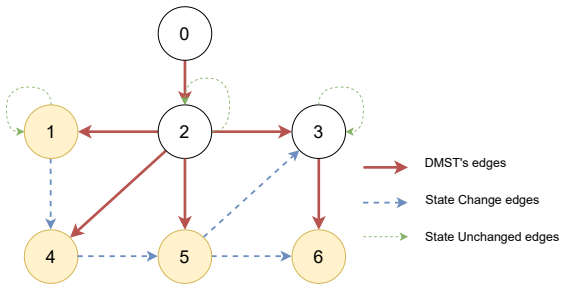


Figure 5: Types of edges in SVRM model. Algorithm 2's goal is to go through all useful edges (including DMST's edges, State Change edges, and some State Unchanged edges) with the lowest cost

To construct the minimum set of directed edge sequences Π_{min} that satisfy the above constraints, we can first find the shortest sequences from the start point v_0 to other nodes in the directed graph $\mathcal{G}$. Then, we can supplement the remaining directed edges with the shortest edge sequences according to the constraints above. Based on this greedy idea, SVRM-Composer uses Algorithm 2 to process the directed graph of the SVRM model of the target, achieving the generation of a high-quality corpus for the target. The traversal of SVRM model is graphically represented in Fig.5.:

First, Algorithm 2 uses DMST algorithm [24–26] to process the directed graph $\mathcal{G}$ to obtain a directed minimum spanning tree $T_{MST}(V, E^{'})$ with v_0 as the starting point. [7]

- To satisfy constraint 1, at this point, the directed edge sequences generated by T_{MST} (These edges can also be noted as DMST's edge in Fig.5) all start from v_0 and can reach any node in the directed graph $\mathcal{G}$, determining the shortest path to reach that node.
- To satisfy constraint 4, Algorithm 2 only generates directed edge sequences to reach the leaf nodes of the minimum spanning tree T_{MST}. These sequences are translated into corresponding input symbol sequences and added to the input symbol sequence set according to the mapping relationship(corresponding to lines 6-10 of the algorithm2).

Then, Algorithm 2 needs to handle the directed edges $E - E^{'}$ that are not included in the minimum spanning tree T_{MST} but can cause state transitions(These edges can also be noted as State Changes edges in Fig.5). To satisfy constraint 2 and 4, the algorithm first traverses each node in the directed graph and classifies and processes each outgoing edge of the node into one of the following three types:

- Type 1: Directed edges that are part of the minimum spanning tree (e.g., edge from node 3 to node 6 in Fig. 5).Since they are already included when constructing the shortest path sequences to reach the leaves of the minimum spanning tree, so the algorithm only needs to consider Type 2 and Type 3 edges.
- Type 2: Directed edges that are not part of the minimum spanning tree and are self-loops (e.g., self-loop edge at node 3 in Fig. 5). These edges will be consider latter
- Type 3, Directed edges that are not part of the minimum spanning tree and are not self-loops (e.g., edge from node 1 to node 4 in Fig. 5).the algorithm first generates the shortest path to reach the edge according to the minimum spanning tree, then adds the edge to the end of the shortest path, translates it into the corresponding input symbol sequence, and adds it to the symbol sequence set (corresponding to lines 17-20 in the algorithm 2)

Furthermore, Algorithm 2 will consider covering all input symbols (to satisfy constraint 3). We use the example of FTP protocol to illustrate the reason why we should consider handling some of Type 2 edges. Consider the HELP

[7] There maybe more than two DMST, our implemented DMST algorithm only generate one DMST according to sequence number of these edges

Algorithm 2: Corpus Generation Algorithm

Input: SVRM model's Graph $G(V, E)$, source node v_0, input alphabet Σ
Output: Corpus $\mathbb{C}$

```
1  C_sym ← ∅;
2  C ← ∅;
3  Σ_used ← ∅;
4  T(V, E') ← DMSTAlgorithm(G, v_0);
5  Leaves ← GetLeaves(T);
6  foreach v ∈ V do
7      EdgeSeq ← GetShortestPath(T, v_0, v);
8      if v ∈ Leaves then
9          SymSeq ← EdgetoSym(EdgeSeq);
10         C_sym ← SymSeq;
11         foreach σ ∈ SymSeq do
12             Σ_used ← σ
13 foreach v ∈ V do
14     foreach e ∈ GetOutEdges(G, v) do
15         if e ∈ E' then
16             continue;
17         else if isNotSelfLoop(e) then
18             EdgeSeq' ← EdgeSeq.append(outedge);
19             SymSeq ← EdgetoSym(EdgeSeq');
20             C_sym ← SymSeq;
21             foreach σ ∈ SymSeq do
22                 Σ_used ← σ
23 foreach σ ∈ Σ − Σ_used do
24     SymSeq ← EdgetoSym(σ);
25     C_sym ← SymSeq;
26 foreach seq ∈ C_sym do
27     C ← SymbolInstantiation(seq)
```

message instance, which is used to print all supported message types and their function information. Although the HELP message does not cause any state transitions in the SUT, it is still a function that needs to be tested. Therefore, the algorithm classifies the input symbols of the SUT based on the characteristics of the directed edges (corresponding to lines 23-25 in the algorithm 2):

- Type A: The input symbol can cause state transition. Since the algorithm 2 have already considered them when processing Type 3 edges, no additional processing is required.
- Type B: The input symbol can not cause state transitions. To satisfy constraint 3, Algorithm 2 adds this input symbol as separate symbol sequence to the symbol sequence set.

Finally, Algorithm 2 translates each symbol sequence in the symbol sequence set into the corresponding message instance sequence, thus achieving the construction of a high-quality corpus adapted to the SUT.

Table 1: Information of SUTs from ProFuzzBench used for evaluation

Program	Protocol Type	Version	LoC	Language
Lightftp	FTP	5980ea1	4.4k	C
Bftpd	FTP	v5.7	4.7k	C
EXIM	SMTP	38903fb	101.7k	C
Openssl	TLS1.2/1.3	0437435a	442.9k	C
Live555	RTSP	ceeb4f4	52.2k	C++
Dcmtk	DICOM	7f8564c	576.9k	C++

4. Implementations

SVRM-Learner is implemented using 1.4k lines of C++ code and 2k lines of Java code. SVRM-Composer is implemented using 600 lines of Python code. We have open-sourced the code here[8] for further research in this field.

SVRM-Learner instruments the state variables of the SUT using the LLVM framework[27], compiling the SUT into an object suitable for SVRM model learning (Section 3.2). It then employs the active learning algorithm provided by Learnlib[28] to construct the SVRM model of the SUT (Section 3.3). Finally, SVRM-Composer parses the generated SVRM model to create a high-quality corpus tailored to the SUT (Section 3.4).

5. Evaluation

We evaluate SVRM-Learner and SVRM-Composer to answer the follwing research questions:

- **RQ1**: Can SVRM-Learner build protocol-independent automata model for various type of network services?
- **RQ2**: How corpus generated by SVRM-Composer perform compared with human-selected corpus?

Benchmark. The benchmark for evaluation selects 6 different protocol inplementations from ProFuzzBench[29], which involve 6 different types of protocols (As shown in Table 1). On the one hand, these targets are selected because of their popularity in protocol fuzzing research. On the other hand, SVRM-Learner currently only implements TCP interface, and will support automaton modeling of UDP-based protocol in the future. The message instances of the SUT come from ProFuzzBench and the corresponding instances of the message types involved in its dictionary.

Baseline. To evaluate the ability of SVRM-Learner to construct the automaton model of the SUTs, we compare it with other open-source modelers Aflnet and StateLearner. To evaluate the fuzzing effect of the corpus generated by SVRM-Composer, we compare the fuzzing results of the initial corpus provided by ProFuzzBench and the corpus constructed by SVRM-Composer on Aflnet and SGFuzz [18].

Experiment environment. All experiments were conducted in Docker on Ubuntu 20.04 LTS server, with 200 GiB of RAM on Intel® Xeon® Gold 6254 3.10 GHz CPU with 64 cores.

5.1. Feasibility of learning SVRM Model on Various SUT

To evaluate the feasibility of SVRM-Learner from two perspectives: versatility and correctness. We evaluate versatility with Aflnet and StateLearner on various SUTs by comparing the implementation details of these modelers and the correctness of SVRM model by case study on Lightftp comparing with model providing by NSFUZZ.

5.1.1. Versatility of SVRM Model

Table 2 shows the ability of SVRM-Learner and other state machine modeling-capable tools on modeling different protocols.

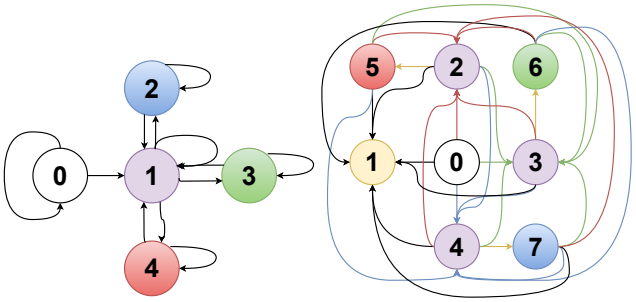
Aflnet and SVRM-Learner both support modeling for all 6 protocols. However, StateLearner only supports modeling for protocols up to TLS 1.2.

To support testing new protocols, both Aflnet and StateLearner require manual customization of their tools, while of SVRM-Learner does not. Specifically, Aflnet requires implementing a message parser to extract status

[8] https://github.com/Br1m4zz/SVRM-CorpusGen

Table 2: Different tools' ability to modeling the protocol implementation and whether they need to be modified based on protocol specification. ✓ denotes the tool support and ✗ denotes NOT.

Modeling Tools	Support Modeling / Protocol-independent					
	FTP	SMTP	TLS1.2	TLS1.3	RTSP	DICOM
AFLNET	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗
STATELEARNER	✗/✗	✗/✗	✓/✗	✗/✗	✗/✗	✗/✗
SVRM-LEARNER	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓



(a) NSFUZZ's model (b) SVRM model

Figure 6: Lightftp's State Machine Models extracted by NSFUZZ (6a) and SVRM-LEARNER (6b). Both of them use the same state variable for modeling.

codes according to protocol specification. STATELEARNER needs to create a harness to translate input symbol into message instance and server response into output symbol based on TLS protocol specification. SVRM-LEARNER only requires providing typical message instances and state variables to build state machine models for various protocol implementations, which can be obtained through automatic methods mentioned in Section 3.2.

Therefore, SVRM-LEARNER can construct communication model for all protocol implementations without manual modification on leaning tools.The generated models are protocol-independent, which shows versatility of our Model

5.1.2. Correctness of SVRM Model

Fig.6 illustrates the detailed model generated by SVRM-LEARNER and NSFUZZ for LightFTP. Both modelers utilize the same state variable `Access` to construct the target model for LightFTP.

As shown in List 1, LightFTP employs state variable `Access` to represent different user groups' permissions within the FTP protocol, and the execution results of different message types vary based on these permissions.

While both modelers distinguish these permission states(color-different node), the SVRM model further subdivides each authentication state into "pre-authentication"(purple-colord state 2,3,4 in Fig.6b) and "authenticated" states(state 5,6,7 in Fig.6b). Additionally, it identifies "inaccessible" states(purple-colord state 1 in Fig.6b) within the SUT. Thus, for the same set of state variables, SVRM-LEARNER can create a more granular state model of SUT.

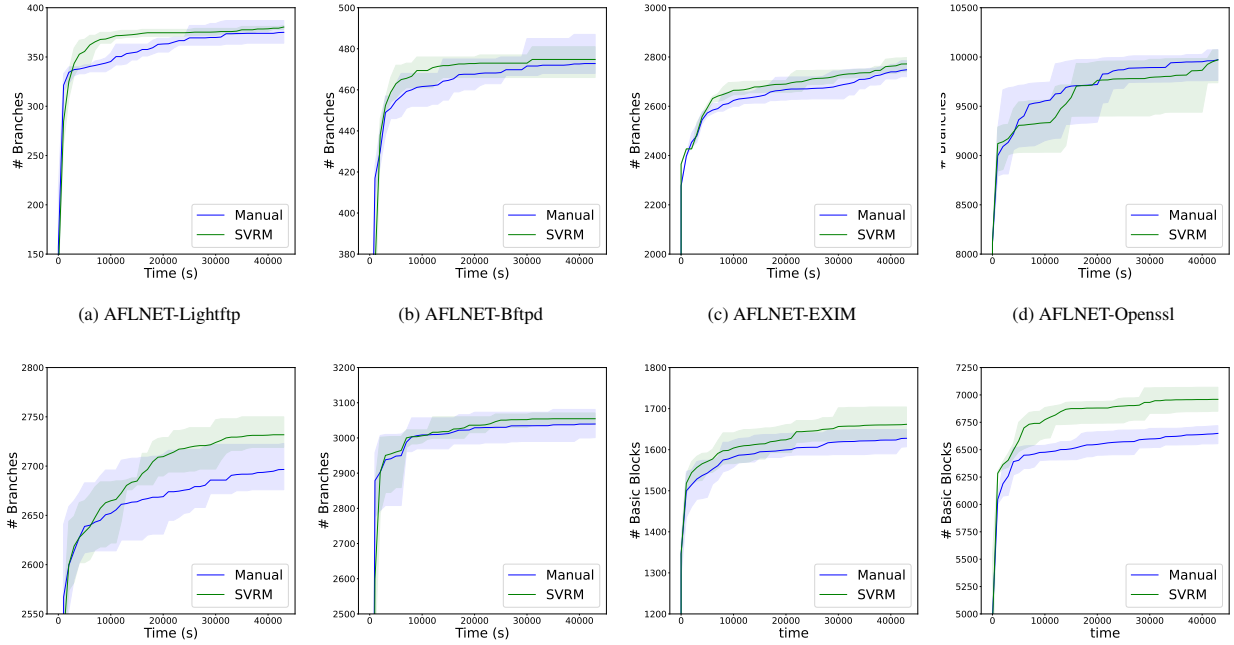
The reason why SVRM model is more granular than NSFUZZ's model is that NSFUZZ only consider whether state variable value changes, which serves as differentiator of states. SVRM consider both input difference and output difference. The definition of different state of SVRM model is more rigorous.

It should be noted that three of the SUTs are unable to obtain the final SVRM model even within 72 hours. The specific reasons are related to the communication efficiency and the complexity of target states model. In Appendix B, we studied the fuzz testing effects of the corpus corresponding to the SVRM models with different numbers of iterations.

In summary, the correctness of model built by SVRM-LEARNER can be ensured and we can use the model to generate high quality corpus.

5.2. Improvement on Code and State Coverage

To evaluate the corpus generated by SVRM-COMPOSER (SVRM Corpus), we compare it with manually crafted corpus provided by PROFUZZBENCH(Manual Corpus). We evaluate the quality of SVRM Corpus by comparing the



(a) AFLNET-Lightftp (b) AFLNET-Bftpd (c) AFLNET-EXIM (d) AFLNET-Openssl

(e) AFLNET-Live555 (f) AFLNET-Dcmtk (g) SGFuzz-Live555 (h) SGFuzz-Openssl

Figure 7: The branches/basicblocks coverage growth of AFLNET and SGFUZZ among 5 runs of 12-hour fuzzing on two corpora. Note that the CGF test corpus initial seeds have already covered some code branches, and the y-axis does not start from 0.

Table 3: The average final code coverage of AFLNET and SGFUZZ among 5 runs of 12-hour fuzzing on two corpora. Note that AFLNET's code coverage in this table is branches coverage and SGFUZZ uses basicblocks coverage. State Coverage for AFLNET is the number of IPSM edge[5], and for SGFUZZ is the number of leaves node in State Transition Tree[18].

Fuzzer-Target	**Code coverage (#)**			**State coverage (#)**		
	Manual corpus	SVRM corpus	inc.rate	Manual corpus	SVRM corpus	inc.rate
AFLNET-Lightftp	386.4	391.4	+1.29%	199.8	234.6	+17.4%
AFLNET-Bftpd	472.8	474.8	+0.42%	226.2	234.0	+3.4%
AFLNET-EXIM	2748.0	2771.8	+0.86%	51.8	77.8	+50.2%
AFLNET-Openssl	9971.6	9970.8	-0.01%	27.6	61.4	+122.4%
AFLNET-Live555	2696.6	2731.8	+1.31%	94.6	121.6	+29.1%
AFLNET-Dcmtk	3039.6	3054.6	+0.49%	3.0	3.0	0
SGFUZZ-Live555	1627.8	1661.8	+2.08%	13089.5	14755.5	+12.7%
SGFUZZ-Openssl	6648.6	6959.6	+4.67%	62.2	366.2	+453.1%

Table 4: The average time to discover the first crash and unique crash number for AFLNET in two corpora among 5 runs experiments in 12 hours.

Target	First Crash Avg Time (s)		Unique Crash(#)		Speed up
	Manual Corpus	SVRM Corpus	Manual Corpus	SVRM Corpus	
AFLNET-Bftpd	8714.0	4369.0	106.2	70.8	+0.99x
AFLNET-Live555	916.6	124.8	23	38.4	+6.34x
AFLNET-Dcmtk	5589.0	6209.8	8.4	3	-0.10x

code coverage and state coverage achieved using the two corpora with different SUTs and CGF tools. The SVRM model used in this experiment is the **final SVRM model** or the latest **hypothesis model learned in 12h**. (this is the empirical time that all targets take more than 3 learning iteration). In order to ensure the fairness of fuzzing, we manually add message instances whose types are not involved in ProFuzzBench as single initial test cases into Manual Corpus.

Table 3 shows the comparison of code coverage and state coverage achieved by two CGF tools, Aflnet and SGFuzz, using SVRM Corpus and Manual Corpus on different network services. Fig. 7 demonstrates detailed comparison of code coverage growth over time for different targets. In the fuzz testing of 6 subjects in AFLNET, the SVRM Corpus can effectively improve the efficiency in discovering new state transitions, increasing the state coverage rate of each SUT by an average of 37.1%. In terms of branch coverage, except for Openssl, the SVRM Corpus can on average cover 13 more branches for each SUT than the Manual Corpus. The fuzzing experiments conducted with SGFuzz showed that the SVRM Corpus also improved its efficiency, increasing the state coverage of two targets by 12.7% and 453.1%, respectively, and increasing the code coverage by 2.08% and 4.67%.

In summary, the corpus generated by SVRM-Learner and SVRM-Composer can increase the number of new states discovered by protocol fuzzers within 12 hours, thus improving code coverage in most SUTs. The generated corpus can meet the needs of current state-of-art CGF tools, and in most targets, they outperform the initial corpus provided by ProFuzzBench.

5.3. Improvement on Bugs Finding

To evaluate the ability of the corpus built by SVRM-Composer in finding vulnerabilities, we evaluates the Manual Corpus and the SVRM Corpus by comparing the time to find the first crash found by Aflnet within 12 hours on different targets.

Table 4 shows the average time to discover the first crash and unique crash number in 12h for Aflnet fuzzing under two corpora. In the experiments against Bftpd and Live555, the SVRM Corpus can discover the first crash more quickly, especially for Live555, where it improved the speed by approximately 6x. The crash in Live555 is a UAF vulnerability, which is related to the state of the RTSP protocol. The SVRM Corpus can provide test cases that thoroughly traverse detailed state transitions in advance. Therefore, compared to Manual Corpus that only provide coverage for common states, the SVRM corpus enables Aflnet to discover state-related vulnerabilities more quickly.

However, for Dcmtk, The performance of the SVRM corpus is suboptimal. The potential reason could be the inherent few states of DICOM protocol. Aflnet is only capable of distinguishing three state transitions. The inclusion of more state-detailed test cases in the SVRM corpus seems to have inadvertently increased the burden of fuzzing seed selection. Aflnet will take more time fuzzing on SVRM corpus with same state guidance.

The results demonstrate that the corpus generated by SVRM-Learner and SVRM-Composer can effectively enhance the speed of discovering vulnerabilities related to protocol state transitions, compared to initial corpus provided by ProFuzzBench.

6. Discussion

We discuss the limitations of SVRM-Learner and SVRM-Composer and how these limitations can be addressed in future work.

First, the reliability of building the SVRM model for the SUT depends on the accuracy of identifying state variables and representative messages selection. For state variables, there is some false positive in state variable identification[18, 13]. Future work can reduce false positives by further introducing dereference chain analysis or

function semantic information analysis. For representative messages selection, Netpiler's cluster method's effectiveness relies on the number of network traffic [21]. And if given few network traffic, the generated model can not represent the actual state machine of SUT. Further research on how to select better representative messages will be conducted.

Second, like other known protocol fuzzers, throughput is a challenge for SVRM-Learner, which can be addressed by introducing "desocket" or "snapshot mechanism" methods from existing works [14, 17, 15] to speed up the learning process.

Third, SVRM-Learner cannot handle the impact of verification fields in messages (such as sequence verification, checksums, etc.). SVRM-Learner uses fuzzing patches provided by ProFuzzBench to ignore the impact of these fields.

Finally, the composed corpus have not find new 0-day vulnerability in ProFuzzBench. ProFuzzBench provides a benchmark for fuzzing network service by manually adapting protocol implementations for Aflnet and its derivatives. Works like AFLNET-LEGION, state-AFL, and NSFUZZ use metrics like code coverage to evaluate effectiveness and also have not identified new vulnerabilities in ProFuzzBench. The goal of this paper is to automatically generate high-quality corpus for these CGF tools. To find new 0-day vulnerabilities, further fuzzing on new protocol targets is required.

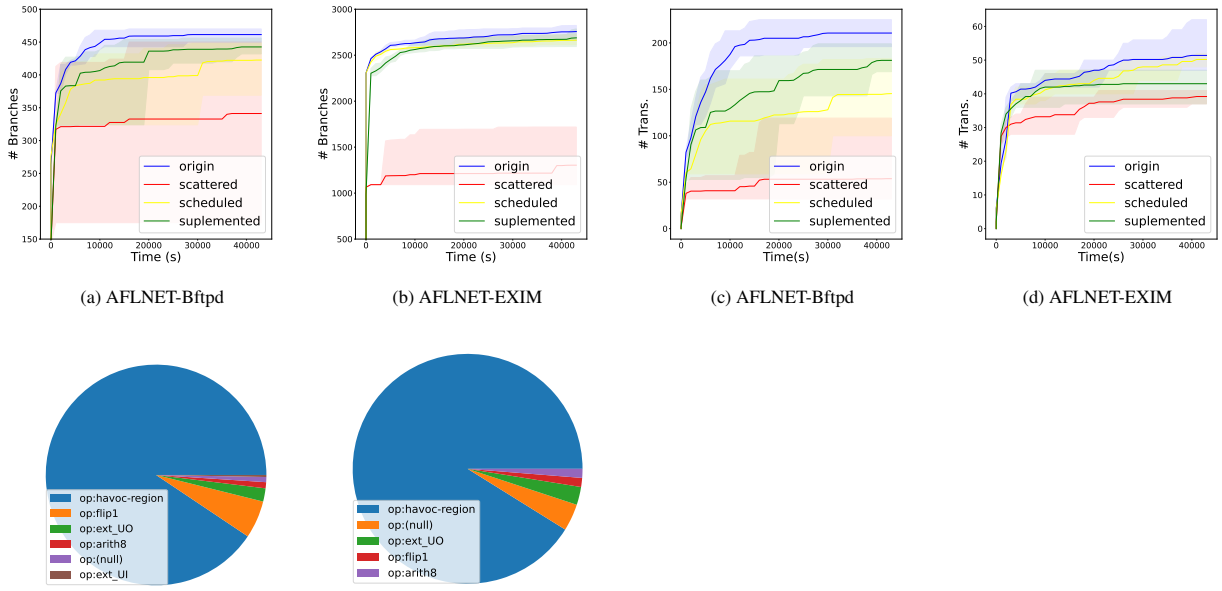
7. Related Work

Protocol Implementation Vulnerability Discovery. Technique for vulnerability discovery in network service can be divided into dynamic methods and static methods based on whether the tested protocol implementations needs to be executed. Dynamic methods reveal potential vulnerabilities based on the runtime abnormal behavior of the tested target, such as fuzzing[4, 5] and symbolic execution[30]. Static methods, on the other hand, use static detection rules to analyze the code attribute graph of the tested target and find defects that violate the rules, such as taint analysis[31]. This paper focuses on serving the coverage-guided fuzzing of protocol implementations, aiming to automatically generate high-quality corpus for fuzzers.

Protocol fuzzing corpus enrichment. ChatAfl[20] learns the target protocol specifications through LLM and then supplements and rewrites each test case with the knowledge of specifications to improve fuzzing performance. In contrast, we aim to generate corpus tailored to target without protocol specifications. In detail, we reconstruct sequences based on representative messages and state variables, which can be obtained by existing automatic methods. Though we both use ProFuzzBench to evaluate our methods, We build a high-quality corpus of the target without protocol specification knowledge.

Protocol state variable indentification. The state variables mentioned in this paper specifically refer to the program variables used to represent the protocol-interaction state. In order to identify the state variables of the SUT, NSFuzz [13] and SGFuzz [18] respectively use static analysis methods based on heuristic rules to filter state variables from all variables. Among them, NSFuzz filters state variables based on the difference between state variables and other variables in IR in operations such as STORE and LOAD; SGFuzz filters state variables based on the programming institution that protocol implementation usually uses enum type to represent state variables in C/C++ language implementation. At present, the identification of state variables still has some false positive, and testers still need to further filter out variables such as configuration variables and response variables that are irrelevant to the protocol state according to variable semantics. We use state variables and their values to construct the output alphabet of the SVRM model, and the reliability of the SVRM model depends on the accuracy of the identification results of the above methods.

Active Model Learning. This approach aims to construct the state automaton model of the SUT by building the MAT of the SUT and using the active learning algorithm. Existing works reveal the logical defects of the SUT by finding the state transition paths that violate the protocol specification design. For example, StateLearner and other similar works[32–35] uses the active learning method to construct the implemented protocol automaton (like DTLS, VPN, SSH, etc.), and manually find semantic vulnerabilities that violate the protocol specification. Compared to these works, we propose a protocol-independent automaton model for open-sourced protocol implementation, which require no labor and protocol knowledge on implementing specific harness. We use the model to generate high-quality corpus for CGF and whether SVRM model can help finding semantic vulnerabilities remains to be discussed.



(a) AFLNET-Bftpd (b) AFLNET-EXIM (c) AFLNET-Bftpd (d) AFLNET-EXIM

(e) Favorable mutators in Bftpd (f) Favorable mutators in EXIM

Figure A.8: Discovered branch coverage (Fig.A.8a, A.8b) and state transition (Fig.A.8c,A.8d) on AFLNET of 12 hours for 4 corpus. Fig.A.8e and Fig.A.8f show the proportion of AFLNET's mutators used in **Scattered Corpus** thar bring new state transitions. Havoc-region is the sequence mutator in AFLNET

8. Conclusion

Building high-quality corpus for protocol fuzzers relies on developers' understanding of the specifications of the service under test(SUT) and prior knowledge of the protocol implementation. To achieve protocol-independent and target-adaptive methods for constructing high-quality corpus, this paper proposes the SVRM state machine model for network service and implements the SVRM-Learner tool for constructing these models. It also introduces the SVRM-Composer tool based on the characteristics of a high-quality corpus to generate target-tailored corpus for fuzzing. Experimental results demonstrate that SVRM-Learner and SVRM-Composer can automatically build universal models for various protocol implementations, generate high-quality corpus tailored to the SUT. The generated corpus can improve the average state coverage and code coverage of coverage-guided fuzzers by 37.1% and discover real vulnerabilities in protocol implementations 2.47x faster.

Appendix A. Corpus comparative experiment

The comparative experiment (results are shown in Fig. A.8) uses four different corpora, and performs fuzzing with Aflnet on Bftpd and Exim respectively. Meanwhile, we count the percentage of different types of mutation operators in the scattered corpus that can bring new state transitions to the SUT. The sources and differences of test cases in each corpus are shown as follows: ❶**Origin Corpus**: Test cases are initial corpus provided by ProFuzzBench [29]. ❷**Scattered Corpus**: The different types of message instances involved in the **Origin Corpus** are separately used as single test cases for the **Scattered Corpus**. ❸**Scheduled Corpus**: Some messages are reorganized in sequence from the **Scattered Corpus** according to SUT's protocol specification ❹**Supplemented corpus**: Some message instances are supplemented as test cases to the **Scheduled Corpus**, which include instances of the message types that have not been covered by **Scheduled Corpus**.

By comparing state coverage and code coverage over time for different corpora(Fig.A.8), we find that the corpus that cover more protocol state transitions (**Scheduled Corpus**) and involve more comprehensive message types

Table B.5: SVRM Corpus generated from hypothesis SVRM model at 6 hours, 12 hours, and 72 hours, including the number of iterations(It.), the number of edges in the model, and the number of generated test cases(T.C.).

Target	6 hour			12 hour			72 hour		
	It.	#edge	#T.C.	It.	#edge	# T.C.	It.	#edge	#T.C.
Bftpd	1	133	13	2	189	193	8	416	644
EXIM	1	117	35	4	540	504	8	1458	1378
Dcmtk	1	49	46	6	494	477	11	713	691

Table B.6: The average final code coverage of AFLNET among 5 runs of 12-hour fuzzing on 3 different time span SVRM corpora. Note that AFLNET's code coverage in this table is branch coverage. State Coverage for AFLNET is the number of IPSM edge,

Target	Code coverage (# branch)			State coverage (# edge)		
	6 hour	12 hour	72 hour	6 hour	12 hour	72 hour
AFLNET-Bftpd	471.8	475.8	471.4	227.6	241.8	234
AFLNET-EXIM	2769.6	2771.8	2777.6	74.0	77.8	75.6
AFLNET-Dcmtk	3016.8	3053.0	3039.0	3	3	3

(**Supplemented Corpus**) have better performance in protocol CGF's code coverage and state transitions for different targets. By analyzing the proportion of mutation operators that bring new state coverage in "Fig.A.8e" and "Fig.A.8f", we find that sequence mutator (AFLNET's "Havoc-region") can bring more state coverage for fuzzing tests. Furthermore, we infer that the message sequence composition of test cases is the main factor affecting the protocol state of the SUT.

Appendix B. Fuzzing Perfromance on Various Hypothesis SVRM Model

To study the impact of different stage of iterations of Hyphthesis SVRM Model on the performance of fuzzing, we use SVRM-COMPOSER to generate corpus for Hyphthesis model learned in 6h, 12h, and 72h respectively, and use AFLNET to perform 12 hours fuzzing among 5 rounds experiments.

Appendix B.1. Number of Test Cases

Table B.5 shows detailed information of these corpora. The number of test cases generated by SVRM-COMPOSER is related to the complexity of SUT's SVRM model. As time progresses and the number of iterations increases, the SVRM model of the SUT becomes increasingly detailed and complex, and the number of test cases in the generated corpus correspondingly increases.

Appendix B.2. Fuzzing Performance

Table B.6 shows the final code coverage and state coverage for different iteration SVRM model. The results show that the model generated over a span of 12 hours achieves the highest code coverage and state coverage in Bftpd and Dcmtk. Moreover, the branch coverage from EXIM indicate that the corpus of 72 hours can lead to greater code coverages.

We have made the following conjectures regarding the specific reasons for the poor fuzz testing performance of the corpora generated by models with either few or many iterations. Models with insufficient iterations suffer from low precision, resulting in generated test cases that do not adequately cover the actual protocol state machine of the SUT. Conversely, excessive iterations lead to overly detailed models, which in turn generate an excessive number of test cases, imposing a significant burden on the selection of test cases for fuzz testing. In the absence of a Final SVRM model, there should exist an optimal Hypothesis SVRM model among multiple iterations, which yields the best performance in fuzz testing.

In conclusion, when it is impossible to obtain the Final SVRM Model of the SUT, there exists a Hypothesis SVRM Model that yields the best fuzz testing performance. The number of iterations for this Hypothesis SVRM Model should not be too many nor too few, which is independent of the iteration time.

Appendix C. Example process of building SVRM model

Appendix C.1. Selection of Representative Message and State Variable for LightFTP

We get target's initial network traffic and source code from ProFuzzBench.

For representative message, since we have a few initial network traffic, we construct Representative Messages by finding and clustering them by repeated string(like PASS USER) and select the message from one network trace.

For state variables, we write an analysis tool by SVF[36] to identify them (The detail of improved state variable static analysis will be present in future work) and for LightFTP, the identified state variables are `Access` and `mode`.

We instrument the source code with a LLVM pass(the instrumentation is shown in 1).All these result can be found in our artifact.

Listing 1: Instrumented Lightftp code snippet.

```
instrumentedcode    context->Access = FTP_ACCESS_NOT_LOGGED_IN;
    do {
        if ( strcasecmp(temptext, "admin") == 0 ) {
            context->Access = FTP_ACCESS_FULL;
            \textbackslash{}__SVRM_hash(SVRM_id(context->Access), FTP_ACCESS_FULL) \\
                instrumented code
            break;
        }
        if ( strcasecmp(temptext, "upload") == 0 ) {
            context->Access = FTP_ACCESS_CREATENEW;
            __SVRM_hash(SVRM_id(context->Access), FTP_ACCESS_CREATENEW) \\ instrumented
                code
            break;
        }
        if ( strcasecmp(temptext, "readonly") == 0 ) {
            context->Access = FTP_ACCESS_READONLY;
            __SVRM_hash(SVRM_id(context->Access), FTP_ACCESS_READONLY) \\ instrumented
                code
            break;
        }
        return sendstring(context, error530_b);
    } while (0);
```

Appendix C.2. Runtime exceptions examples

When handling "QUIT", Lightftp will send a goodbye message and then shutdown the socket. Although we can get StateBitmap's hash, it is not correct to represent the incommunicable state with this hash, Algorithm 1 will use λ_0 to represent the state.

In the learning process of Live555, the initial version of SVRM-Learner find that Learlib will exit with exception that they find the output symbol is not the same with same input. We find that Live555 will take time processing some messages and the state variable value is not stable. Thus Algorithm 1 will handle the phenomenon and wait till the state variable value is stable.

References

[1] Heartbleed Bug.
URL `https://heartbleed.com/`

[2] CVE-2020-0796 - Security Update Guide - Microsoft - Windows SMBv3 Client/Server Remote Code Execution Vulnerability.
URL `https://msrc.microsoft.com/update-guide/en-US/advisory/CVE-2020-0796`

[3] boofuzz: Network Protocol Fuzzing for Humans — boofuzz 0.4.2 documentation.
URL `https://boofuzz.readthedocs.io/en/stable/index.html`

[4] Peach by MozillaSecurity.
URL `https://mozillasecurity.github.io/peach/`

[5] V.-T. Pham, M. Bohme, A. Roychoudhury, AFLNET: A Greybox Fuzzer for Network Protocols, in: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), IEEE, Porto, Portugal, 2020, pp. 460–465. doi:10.1109/ICST46399.2020.00062.

[6] Y. Yu, Z. Chen, S. Gan, X. Wang, SGPFuzzer: A State-Driven Smart Graybox Protocol Fuzzer for Network Protocol Implementations, IEEE Access 8 (2020) 198668–198678. doi:10.1109/ACCESS.2020.3025037.

[7] R. Natella, StateAFL: Greybox Fuzzing for Stateful Network Servers, arXiv:2110.06253 [cs] (Oct. 2021).
[8] L. Hayes, H. Gunadi, A. Herrera, J. Milford, S. Magrath, M. Sebastian, M. Norrish, A. L. Hosking, Moonlight: Effective fuzzing with near-optimal corpus distillation, arXiv preprint arXiv:1905.13055.
[9] A. Herrera, H. Gunadi, L. Hayes, S. Magrath, F. Friedlander, M. Sebastian, M. Norrish, A. L. Hosking, Corpus distillation for effective fuzzing: A comparative evaluation, arXiv preprint arXiv:1905.13055.
[10] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, K. Rieck, Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols, in: B. Thuraisingham, X. Wang, V. Yegneswaran (Eds.), Security and Privacy in Communication Networks, Springer International Publishing, Cham, 2015, pp. 330–347.
[11] Y. Hsu, G. Shu, D. Lee, A model-based approach to security flaw detection of network protocol implementations, in: 2008 IEEE International Conference on Network Protocols, IEEE, Orlando, FL, USA, 2008, pp. 114–123, iSSN: 1092-1648. doi:10.1109/ICNP.2008.4697030.
[12] J. de Ruiter, E. Poll, Protocol state fuzzing of TLS implementations 14.
[13] S. Qin, F. Hu, B. Zhao, T. Yin, C. Zhang, Registered Report: NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing 9.
[14] J. Fu, S. Xiong, N. Wang, R. Ren, A. Zhou, B. K. Bhargava, A Framework of High-Speed Network Protocol Fuzzing Based on Shared Memory, IEEE Transactions on Dependable and Secure Computing (2023) 1–18doi:10.1109/TDSC.2023.3318571.
[15] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, T. Holz, Nyx-net: network fuzzing with incremental snapshots, in: Proceedings of the Seventeenth European Conference on Computer Systems, ACM, Rennes France, 2022, pp. 166–180. doi:10.1145/3492321.3519591.
[16] J. Li, S. Li, G. Sun, T. Chen, H. Yu, Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots (2022). doi:10.1109/TIFS.2022.3192991.
[17] A. Andronidis, C. Cadar, SnapFuzz: high-throughput fuzzing of network applications, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, Virtual South Korea, 2022, pp. 340–351. doi:10.1145/3533767.3534376.
[18] J. Ba, M. Böhme, Z. Mirzamomen, A. Roychoudhury, Stateful Greybox Fuzzing, arXiv:2204.02545 [cs] (May 2022).
[19] D. Liu, V.-T. Pham, G. Ernst, T. Murray, B. I. Rubinstein, State Selection Algorithms and Their Impact on The Performance of Stateful Network Protocol Fuzzing, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, Honolulu, HI, USA, 2022, pp. 720–730. doi:10.1109/SANER53432.2022.00089.
[20] R. Meng, M. Mirchev, M. Bohme, A. Roychoudhury, Large Language Model guided Protocol Fuzzing.
[21] Y. Ye, Z. Zhang, F. Wang, X. Zhang, D. Xu, NetPlier: Probabilistic Network Protocol Reverse Engineering from Message Traces, in: Proceedings 2021 Network and Distributed System Security Symposium, Internet Society, Virtual, 2021. doi:10.14722/ndss.2021.24531.
[22] american fuzzy lop.
URL `https://lcamtuf.coredump.cx/afl/`
[23] D. Angluin, Learning regular sets from queries and counterexamples, Information and Computation 75 (2) (1987) 87–106. doi:10.1016/0890-5401(87)90052-6.
[24] Y. Chu, T. Liu, On the Shortest Arborescences of a Directed Graph, Scientia Sinica 14 (1965) 1396–1400.
[25] J. Edmonds, Optimum branchings, Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics 71B (4) (1967) 233. doi:10.6028/jres.071B.032.
[26] H. N. Gabow, Z. Galil, T. Spencer, R. E. Tarjan, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, Combinatorica 6 (2) (1986) 109–122. doi:10.1007/BF02579168.
[27] The LLVM Compiler Infrastructure Project.
URL `https://llvm.org/`
[28] LearnLib.
URL `https://learnlib.de/`
[29] R. Natella, V.-T. Pham, Profuzzbench: A benchmark for stateful protocol fuzzing, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021.
[30] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, D. Song, Mace: model-inference-assisted concolic exploration for protocol and vulnerability discovery, in: Proceedings of the 20th USENIX Conference on Security, SEC'11, USENIX Association, USA, 2011, p. 10.
[31] J. Cai, P. Zou, D. Xiong, J. He, A guided fuzzing approach for security testing of network protocol software, in: 2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS), 2015, pp. 726–729. doi:10.1109/ICSESS.2015.7339160.
[32] L.-A. Daniel, E. Poll, J. de Ruiter, Inferring openvpn state machines using protocol state fuzzing, in: 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), IEEE, 2018, pp. 11–19.
[33] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. De Ruiter, K. Sagonas, J. Somorovsky, Analysis of {DTLS} implementations using protocol state fuzzing, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 2523–2540.
[34] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, 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, SPIN 2017, Association for Computing Machinery, New York, NY, USA, 2017, pp. 142–151. doi:10.1145/3092282.3092289.
[35] K. Sagonas, T. Typaldos, EDHOC-Fuzzer: An EDHOC Protocol State Fuzzer, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, Seattle WA USA, 2023, pp. 1495–1498. doi:10.1145/3597926.3604922.
[36] Y. Sui, J. Xue, Svf: interprocedural static value-flow analysis in llvm, in: Proceedings of the 25th international conference on compiler construction, ACM, 2016, pp. 265–266.