Protocol Fuzzing With Specification Guided Message Generation

Senyi Li*, Junqiang Li†, Jingxuan Fu‡, Mingwu Xue§, Hongfang Yu¶ and Gang Sun‖
School of Information and Communication Engineering, University of Electronic Science and Technology of China
Chengdu, China
{*lisy, †lijunqiang, ‡fujingxuan, §xuemingwu}@std.uestc.edu.cn, {¶yuhf, ‖gangsun}@uestc.edu.cn

***Abstract*—Fuzzing is one of the most successful techniques for protocol implementation security analysis and vulnerability discovery. Greybox fuzzing for network protocol has become a popular area for its high efficiency. However, present fuzzers are suffering performance drop as they pay no or not enough attention to network protocol specification. To be more detailed, unconstrained mutation and illegal state transition lead to low pass rate for messages. We analyzed present popular fuzzers and found that most fuzzers generate low quality messages that would be rejected by target program even at the beginning. To overcome this problem, we proposed a novel fuzzing approach which contains constrained message mutation and more efficient state chain searching algorithm based on network protocol specification. Our 24 hours experiments demonstrated that our approach outperforms the popular protocol fuzzing tool in both pass rate(at least 9.3%) and edge coverage(at least 12.1%).**

***Keywords*—network protocol security, fuzzing, network protocol specification**

I. Introduction

Network protocol fuzzing is one of the most effective vulnerability discovery method for network applications. Fuzzing techniques can be divided into black-box fuzzing, white-box fuzzing and greybox fuzzing from respective of access to source code or executable binary[1]. Existing researches on network protocol fuzzing mainly focus on black-box fuzzing several years ago[2–5]. While recently greybox network fuzzing has become the most popular technique with high efficiency [6, 7].

However, current network protocol fuzzing works suffer performance drop as they pay not enough attention to network protocol specification(NPS). Specification in network protocol fuzzing reflects in message structure and state transition, which affect the pass rate of messages and finally influence fuzzing efficiency. We analyzed the sessions between popular fuzzer and program under testing and found that a large part of messages were rejected by target program or lead to meaningless states which is a waste of time and results into low efficiency. For instance, SPIKE Fuzzer[8] is one of the earliest fuzzer that can fuzz network protocol but it only utilize message structure information while attention to state machine is paid by user. Peach Fuzzer[9] is one of the most successful black-box fuzzer which takes use of message template to generate test cases quickly and it explicitly defined state transition model in its template file. But Peach Fuzzer is limited to the man-given state transition models therefore it cannot learn state transitions that are not given. AFLNet[6] extends AFL[10] with network publishers and it can learn states and state transitions from scratch automatically. However, AFL-based tools utilize bit level mutations which would seriously destroy the message structure, and in addition, AFLNet doesn't classify legal state transitions and illegal state transitions.

For addressing the challenges mentioned above, and promoting the performance of network protocol fuzzing, we proposed two methods based on NPS in this paper. The first one takes use of grammar and semantics for message generation and mutation. The second method controls the construction and mutation of state chain with sequential specification. We conduct experiments on Real Time Streaming Protocol(RTSP) and Domain Name System(DNS) protocol and the results demonstrate that our method outperforms AFLNet by at least 9.3% in pass rate, 12.1% in edge coverage.

To summarize, this paper makes the following contributions:

- We analyzed previous researches on network protocol fuzzing which pay not enough attention to NPS and found that generation and mutation on message and state chains violating NPS may result into low pass rate and finally lead to low efficiency.
- We proposed a novel approach for the generation and mutation process of message and state chain which bring in NPS information as guidance.
- We implemented our method called NPSFuzzer , and conducted experiments on DNS and RTSP protocols. The results demonstrated that NPSFuzzer outperforms the popular network protocol fuzzing tool AFLNet on both pass rate and edge coverage.

II. Background & Related Work

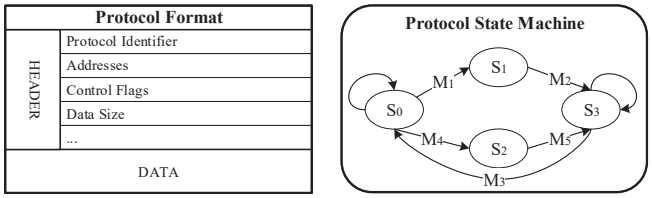
A. Network Protocol Specification

Network Protocol Specification contains three key elements for communication in computer network, i.e., *Semantics*, *Syntax* and *Timing*. To be more detailed, *Semantics* specifies the content to exchange, *Syntax* specifies how to communicate in network and *Timing* defines the order of communication. These key elements become constraints for network protocol fuzzing in terms of message structure and state chain. As mentioned above, a test case in network protocol fuzzing is actually a message chain(MC) contains

* Hongfang Yu is corresponding author.

2021 International Conference on UK-China Emerging Technologies (UCET) | 978-1-6654-9575-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/UCET54125.2021.9674965

several mutated messages. It is basic for fuzzing to ensure that test cases are able to be accepted by target program so that further program codes can be reached. In order to pass the syntax check of target program, each message should be well structured and the order of messages sent need to meet the future of NPS's finite state machine(FSM).



(a) Message Structure (b) Finite State Machine

Fig. 1. Message Structure and Finite State Machine of Network Protocol

Message Structure describes the content and data format of a message as shown in Figure 1(a). The most significant part of message structure in network protocol fuzzing is *HEADER*, which determines the protocol type of message, control flags and the size of data carried by this message. An effective fuzzer with high pass rate should keep the data format as well as possible so as to successfully trigger state transition and explore more logic region in target program.

Finite State Machine shown in Figure 1(b) exists in most network protocols and can be accessed from their Request For Comments(RFC). FSM describes the number of states and transition relationships among them, which help developers to implement network protocol service more effectively.

B. Network Protocol Greybox Fuzzing

Network Protocol Greybox Fuzzing(NPGF) takes network protocol program(NPP) as target and generates messages which are random in both content and sequence, aiming at expose unknown vulnerabilities of the program under testing. The main difference between network protocol fuzzing and normal fuzzing[10–12] depends on whether the input is sequential data[6]. Greybox fuzzing for network protocol implementations not only utilizes coverage measures for normal programs[13–16], it also requires the ability to keep message structure during mutation and considering state machine of target protocol[2, 17].

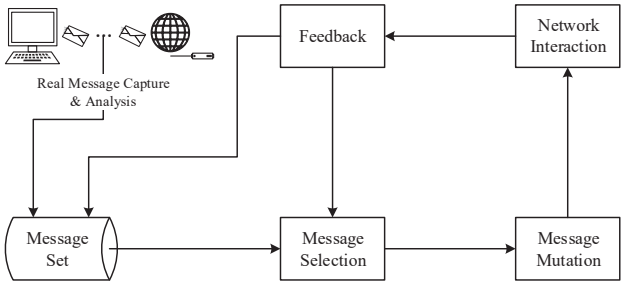


Fig. 2. Network Protocol Greybox Fuzzing

Figure 2 illustrates the general procedure of NPGF, which starts from capturing and analyzing the real messages over network in order to form an initial *Message Set*. After the initialization NPGF enters the main loop, which would performs *Message Selection*, *Message Mutation*, *Network Interaction* and *Feedback* sequentially and repeatedly. In detail, *Message Selection* selects most potential message from *Message Set* as seed where potential here means the probability that a seed could result into coverage gains or vulnerabilities exposure after mutation. Then *Message Mutation* performs single or several mutations on selected seed to generate test case that consists of one or a sequence of mutated messages. After that *Network Interaction* interacts with target NPP by sending messages picked from test case sequentially and receiving corresponding responses. At last *Feedback* would check the process content of target NPP to gather execution information, based on which NPGF determines whether a test case is potential enough to join in the *Message Set* and to wait for next selection. This procedure is classical and effective for being capable to fuzz NPP and naturally it brings the question that how to perform NPGF efficiently which would be discussed in Section III.

III. Efficiency Analysis of Network Protocol Fuzzing

Not all Network Protocol Fuzzers pay equal attention to NPS, thus an intuitive assumption is that they have different efficiency on fuzzing NPPs. To verify this hypothesis, we analyzed the state transition procedure with graph theory and conducted experiments on AFLNet[6] and Peach[9] to analyze their efficiencies.

A. Analyze State Transition with Graph Theory

As network protocol fuzzing aims to expose vulnerabilities in NPP, it is beneficial that a test case is able to trigger deeper state. This is because a vulnerability can be exposed if and only if the logic region where the vulnerability locates can be reached. While in NPP most states own their independent logic region therefore triggering deeper state is an effective method to explore more logic region. Figure 3 illustrates the timing relationship between message chain and state chain, where EOM means end of message. $M_{t(i)}$ and $S_{t(i)}$ indicate message sent to target NPP at time i and state of target NPP at time i, respectively. In the view of graph theory, states and transitions can be regarded as vertices and edges. And transition relationship of FSM can be present by its adjacent matrix $T_{n\times n}$ where n is the number of states and the value of element $T_{i,j}$ indicates whether state S_i can transit to state S_j by accepting message M. In the example of Figure 1(b), the adjacent matrix can represent as

$$T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

, where $T_{1,1} = 1$ means there is one way for S_0 to perform self-transition while $T_{2,3} = 0$ means it's impossible for S_1 to transit to S_2.

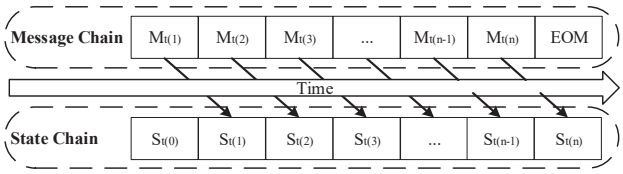


Fig. 3. Timing Relationship of Message Chain and State Chain

B. Evaluate Fuzzing Efficiency with Successful State Transition Rate

We proposed a new approach to measure test cases' potential of reaching deep state. We define $F(.,.)$ to present the transition from $S_t(i)$ to $S_{t(i+1)}$ with $M_{t(i)}$ by $S_{t(i+1)} = F(S_{t(i)}, M_{t(i)})$. Notice that this equation holds only when $T_{t,t+1} = 1$ therefore a message chain able to reach deeper sate should keeps $T_{t,t+1} = 1$ for $t = 0, 1, 2, ..n$. Based on this, we define successful state transition rate(SSTR) $P(S_0, MC)$ to evaluate the quality of a test case from the respective of pass rate as shown in Algorithm 1. The value of SSTR ranges from 0 to 1 and measures the state chain depth of a message chain, which is positively correlated with coverage of logic region.

Algorithm 1: Algorithm for calculating SSTR

Input: Initial state S_0; Message chain MC; Adjacent matrix T.

```
S ← S_0; // current state
q ← 0; // successful state transition counter
for i in range(1, length(MC)) do
    M ← MC(i); // get the i_th message
    S' ← F(S, M)
    q ← q + T_{S,S'};
    S ← S'; // state transition
end
ref q ← q/length(MC); // SSTR
```

Output: SSTR q.

We conducted experiments on RTSP protocol with one of its implementation *live555*[18]. Choosing RTSP protocol as target makes it easy to calculate $P(S_0, MC)$ because the RTSP server responses *200 OK* only when correct sate transition happens while it responses error codes for other cases. We chose 2 fuzzers to fuzz *live555* shown in Table I and measure their SSTR separately. Peach[9] is a black-box fuzzer which is able to generate customized data model defined by users and we wrote a well-formed RTSP protocol message model. While AFLNet[6] supports several protocol fuzzing including RTSP.

TABLE I
PASS RATE ANALYSIS EXPERIMENTS SETTING

Group ID	Fuzzer	Description
1	Peach	well-formed message generation
2	AFLNet	stateful greybox fuzzing

The experiment result is shown in Figure 4 and it demonstrated that Peach Fuzzer can achieve better pass rate than AFLNet with NPS-based message generation.

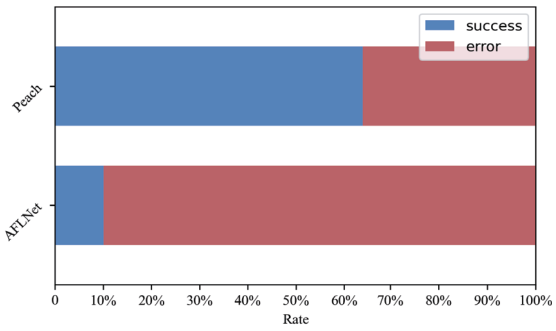


Fig. 4. SSTR Results of Different Fuzzers

Based on the observations above, we come to the conclusion that NPS effects NPGF deeply and existing greybox fuzzers are not efficient because they pay not enough attention to NPS. Fuzzers should constrains their test case generation according to NPS which could promote the SSTR of test cases and finally improves the fuzzing efficiency.

IV. Improving Network Protocol Fuzzing with NPS Guided Message Generation

In order to overcome challenges discussed in Section III and improve the performance of network protocol fuzzing, we proposed a NPS-based strategy for NPGF that maximizes the SSTR of test cases generated during fuzzing. Our approach contains two key points. The first one is *Message Generation* that controls the content and structure of message to keep more protocol information. The other key point is *State Chain Generation* which utilizes FSM information to constrain the generation of state chains in order to reach deeper state.

We implemented our approach as a tool called NPSFUZZER (**N**etwork **P**rotocol **S**pecification based Greybox **Fuzzer**) based on popular greybox fuzzer AFL[10] and black-box fuzzer Peach[9].

A. Overview of NPSFUZZER

Traditional NPGF starts with an initial message set as seed inputs and repeats fuzzing on these messages by performing random message mutations. Test cases in form of message chains generated from random mutation are then used to interact with the target NPP. And message chains that increase the coverage are added to an interesting message chain set for later fuzzing and analysis. The procedures of message structure mutation and message chain generation are where we extended.

Algorithm 2 shows our extension on the basis of traditional network protocol fuzzing, which are contained in blue boxes. We introduce NPS as a new type of input and extract constraints such as input formats and state transition rules; before the main loop of fuzzing procedure, NOSFUZZ takes FSM of NPS to generate state chains SC which are absolutely legal. Then some messages sampled from S_M are feed into the state

Algorithm 2: NPSFUZZER Algorithm

Input: Target program P; Initial Message Set S_M; Network Protocol Specification NPS.

```
/* generate well-formed state chains
   and message chains                    */
```

$SC \leftarrow$ GENERATESTATECHAINS(NPS);
$MC \leftarrow$ GENERATEMESSAGECHAINS(S_M,SC);
$MC_{vio} \leftarrow \emptyset$;
repeat
  foreach $mc \in MC$ **do**
    $m \leftarrow$ MESSAGESELECTION(mc);
    $m' \leftarrow$ MESSAGEMUTATION(m,NPS);
    $mc' \leftarrow$ MESSAGECHAINMODIFICATION(mc,m');
    result=NETWORKINTERACTION(mc',P);
    if *result is trigger crash* **then**
      add mc' to MC_{vio};
    else if *result is increasing coverage* **then**
      add mc' to MC;
    end
  end
until TIMEOUT*()*;

Output: Message Chains that Violating Security Policy MC_{vio}.

chains to generate initial message chains into MC. This guarantees that all message chains meet some specific requirements as different NPPs have different implementations for the same network protocol. Steps above finishes the initialization works and would be detailed discussed in Subsection IV-B. In main loop, the fuzzer iterate every message chain $mc \in MC$ and sample one message m. The message m is then applied to NPS-based message mutation to generate a new message m' which will be used to perform message chain modification on mc. The details about message generation would be introduced in Subsection IV-C. Finally, a mutated message chain mc' is generated and be feed into target NPP via network interaction. If any crashes happen or violating the fuzzer's security policy, mc' is added into MC_{vio}; else if coverage is increased by this message chain, mc' is added into MC for later fuzzing.

B. State Chain Generation

State chain generation is the first entry point where we bring NPS information into network protocol fuzzing. As mentioned in Subsection II-A, there is one finite state machine that guides the implementation of target protocol. With finite state machine, we can infer all legal state chains according to the state transition rules. In this work, by regarding state transitions as edges of graph, we proposed a NPS-based method to search all paths between two vertexes on all pairs of states to generate all legal state chains at the initial stage. The detailed algorithm for path search can be found in Algorithm 3. It's worth noting that, Algorithm 3 only shows the method to search paths from original graph, therefore paths found are length limited. To generate more long state chains, we can simply repeatedly performance path searching based on end vertex of existing path. At the same time of state chain searching, an corresponding message chain can be easily formed which contributes to the initial message chains.

For instance, RTSP is a network control protocol designed for use in entertainment and communications systems to control streaming media servers. The finite state machine of RTSP server is defined in [19], and we can infer all legal state chains from $Init$ to $Playing$. By classifying these state chains by length, we can get that the most short state chain is $Init \rightarrow Ready \rightarrow Playing$ and its corresponding message chain is $SETUP \rightarrow PLAY$. While a longer state chain of length 5 and its message chain are $Init \rightarrow Ready \rightarrow Recording \rightarrow Ready \rightarrow Playing$ and $SETUP \rightarrow RECOD \rightarrow PAUSE \rightarrow PLAY$, respectively.

Algorithm 3: Algorithm for Path Search Between Two Vertexes

Input: Vertex set V; Adjacent Matrix T; Start Vertex v_S; End Vertex v_E.

$P \leftarrow \emptyset$; `// path found`
$S \leftarrow \emptyset$; `// vertex stack`
S.push(v_S);
while *not S.isEmpty()* **do**
  `/* get the top element of` S `and mark it as visited */`
  $v \leftarrow S$.peak();
  $N \leftarrow 0$;
  foreach $v_n \in V$ **do**
    if $T_{v,v_n} = 1$ *and* $v_n \notin S$ **then**
      S.push(v_n);
      if v_n *equals* v_E **then**
        P.add(S);
        S.pop();
      end
      $N \leftarrow N + 1$;
    end
  end
  if *N = 0* **then**
    S.pop();
  end
end

Output: All paths found P.

Discussing: This kind of searching cannot guarantee that all the state chains found works correctly for all implementations of specific network protocol. However, in terms of SSTR, it is already better compared to fuzzers that pay little attention to NPS. Before further discussing, we have to indicate that state chain generation strategy using purely random searching is equal to violently enumerating all possible state chains ignoring state machine. To evaluate how efficient is this search approach compared to random state chain searching, we have the following analysis:

- let $N(l)$ to denote the number of legal state chains with

length l, we have $N(l) = \sum_{i=1}^{d} \sum_{j=1}^{d} T_{i,j}^{l}$, where d is the size of adjacent matrix T, and T^l denotes the lth power of T.

- let $M(l)$ to denote the number of state chains with length l that random searching could explore, then $M(l) = d^{l+1}$.
- On the condition that state chain's length is l, let $p(l) = \frac{N(l)}{M(l)}$ to denote the ratio of legal state chains compared to the entire state chain space. The smaller $p(l)$ is, the harder that searching all legal states by random method will be.

Based on this, we conducted analysis on RTSP and Datagram Transport Layer Security(DTLS) protocol to calculate the l-$p(l)$ curve as shown in Figure 5. The result demonstrate that our searching method is much more efficient compared to random searching as state chain's length goes longer. Besides, we can find that in terms of RTSP, legal state chains occupies less than 1 percent of total state chains, even though T is a dense matrix which generates a larger value of $N(l)$. For other protocols such as DTLS that contains sparser matrix $p(l)$ would be much smaller. Therefore, it is necessary to cut down the searching space for stateful network protocol fuzzing and our NPS-based searching method is a good resolution.

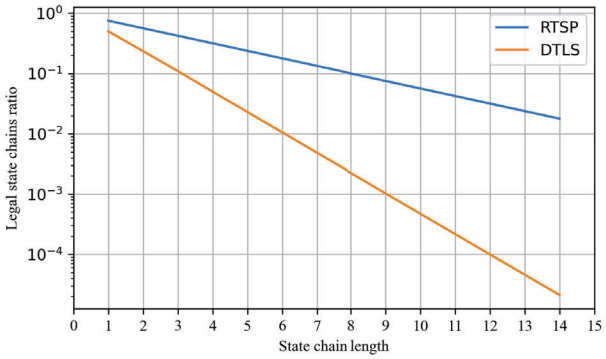


Fig. 5. $p(l)$ of RTSP protocol

C. Message Generation

Message generation includes three parts, i.e., *MS*, *MM* and *Message chain modification*.

At first, *MS* selects one message from message chain for mutation, and in our approach, we simply select the last message and regard the messages before as pilot messages.

Then we apply NPS-based mutations on selected message. As illustrated in Figure 1(a), messages of network protocol is supposed to follow specific formation. For example, *HEADER* contains several fixed length fields that carry different information such as protocol identifier, target address, data size, etc.. Illegal message content could also cause state transition failed even though the state chain is legal.

In order to overcome this problem, we constrain message mutations to avoid dramatically destroying message structure as much as possible. We consider message mutation from three aspects by descending order in granularity as bellow:

1) **Rough Structure**: we divide a message into blocks, each of which has a fixed length's data specified by NPS.
2) **Inner Structure**: message block consists of two kinds of data, i.e., unchangeable filed and changeable filed and we put most mutation energy into the changeable filed.
3) **Value**: data type and range of different filed are considered during mutation to maximally remain syntax and semantic.

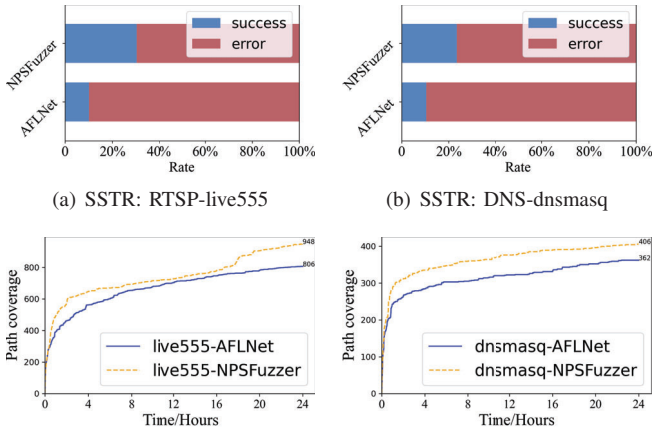
Finally, we modify the message chain with mutated message via replacement, and a new message chain is generated.

V. Evaluation

Our main hypothesis is that taking use of NPS information could increase the fuzzing efficiency. More detailed, our hypothesis consists of two parts: 1) maximally remaining syntax and semantics during message mutation improves network fuzzing efficiency; 2) generating more illegal state chains increases the pass rate and fuzzing efficiency. To evaluate our hypothesis, we implemented NPSFuzzer and conducted comparison experiments on RTSP and DNS[20] protocols.

Implementation: we implemented our NPS-based message generation based on Peach[9] and utilize edge coverage of AFL[10] as feedback to make NPSFuzzer a greybox fuzzer. Our state chain generation algorithm was implemented independently and it works before main fuzzing loop.

Experimental Settings: we compared our tool NPSFuzzer with AFLNet[6] on RTSP and DNS protocols. For program under testing, we selected *live555*[18] as RTSP's implementation and *dnsmasq*[21] as DNS's implementation. All experiments were conducted on a machine with Intel(R) Xeon(R) CPU E5-2620 2.00GHz with a total of 24 cores and 16GB RAM. To ensure the fairness, we run every pair of experiments simultaneously with virtual machines and each virtual machine was bound to 2 CPU cores and 4GB RAM. All experiments lasted for 24 hours and took the same messages as initial seeds.



(a) SSTR: RTSP-live555 (b) SSTR: DNS-dnsmasq

(c) Edge Coverage: RTSP-live555 (d) Edge Coverage: DNS-dnsmasq

Fig. 6. Fuzzing Efficiency Comparison

Results of our 24 hours experiments are illustrated in Figure 6(a) and 6(b). The results in Figure 6(c) and 6(d) show that our tool NPSFuzzer outperforms AFLNet in both final edge coverage and convergence speed, which confirm

our hypothesis that NPS information is significant in NPF and taking use of it could increase the fuzzing efficiency of network protocol programs. In addition, SSTR shown in Figure 6(a) and 6(b) proved that NPSFUZZER successfully utilizes NPS information to fuzz NPP as it has higher SSTR on both *live555* and *dnsmasq*.

Analysis: We conducted experiments on RTSP and DNS to evaluate our hypothesis. It's worth noting that RTSP contains state machine with multiple states, while DNS always stay in single state. Due to state count, there is no impact on SSTR for testing *dnsmasq* from respective of state chain. However, NPSFUZZER still outperforms AFLNet, this proved part of our hypothesis: utilizing NPS information to keep message syntax and semantics improves the efficiency of network protocol fuzzing. As for experiment on RTSP, NPSFUZZER outperforms AFLNet on both SSTR and edge coverage, which evaluate the other part of hypothesis that more legal state chains increases the pass rate and fuzzing efficiency.

VI. CONCLUSION

In this paper, we analyzed present popular network protocol fuzzers and found that most of them pay not enough attention to network protocol specification, which lead to low efficiency. Based on this, we proposed a novel measure of network protocol fuzzing, named *successful state transition rate(SSTR)*. Our 24 hours preliminary experiments demonstrated that our NPS-based fuzzing tool NPSFUZZER outperforms popular fuzzer AFLNet both on edge coverage and SSTR.

In the future work, we plan to conduct more experiments on more popular network protocols (include but not limited to application layer protocols) to further evaluate our hypothesis. Besides, we plan to research on more efficient methods to generate state chains and filtering for potential state chains.

ACKNOWLEDGEMENT

This research was supported by Chinese National Key Laboratory of Science and Technology on Information System Security (Project No. 6142111200303).

REFERENCES

[1] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[2] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 330–347.

[3] 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, 2017.

[4] B. d. S. Melo, P. L. de Geus, and A. A. Gregio, "Robustness testing of coap server-side implementations through black-box fuzzing techniques," in *Proceedings of the Brazilian Symposium on Information Security and Computer Systems*, 2017, pp. 533–540.

[5] 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 on Computing Frontiers*, 2018, pp. 138–145.

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

[7] B. Blumbergs and R. Vaarandi, "Bbuzz: A bit-aware fuzzing framework for network protocol systematic reverse engineering and analysis," in *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*. IEEE, 2017, pp. 707–712.

[8] "Spike fuzzer." [Online]. Available: https://github.com/guilhermeferreira/spikepp

[9] M. Eddington, "Peach fuzzer." [Online]. Available: http://www.peachfuzzer.com

[10] Z. Michał, "American fuzzy lop fuzzer." [Online]. Available: https://lcamtuf.coredump.cx/afl

[11] "Libfuzzer: A library for coverage-guided fuzz testing." [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[12] "Boofuzz: A fork and successor of the sulley fuzzing framework." [Online]. Available: https://github.com/jtpereyda/boofuzz,

[13] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[14] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[15] Y. Yang, Y. Jiang, Z. Zuo, Y. Wang, H. Sun, H. Lu, Y. Zhou, and B. Xu, "Automatic self-validation for code coverage profilers," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 79–90.

[16] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *NDSS*, 2020.

[17] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540.

[18] R. Gaufman, "live555." [Online]. Available: https://github.com/rgaufman/live555

[19] A. Rao, R. Lanphier, and H. Schulzrinne, "Real Time Streaming Protocol (RTSP)," RFC 2326, Apr. 1998.

[20] "Domain names - concepts and facilities," RFC 1034, Nov. 1987.

[21] S. Kelley, “dnsmasq.” [Online]. Available: https://thekelleys.org.uk/dnsmasq/doc.html