AFLNET: A Greybox Fuzzer for Network Protocols

Van-Thuan Pham
Monash University
thuan.pham@monash.edu

Marcel Böhme
Monash University
marcel.boehme@monash.edu

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

***Abstract*—Server fuzzing is difficult. Unlike simple command-line tools, servers feature a massive state space that can be traversed effectively only with well-defined sequences of input messages. Valid sequences are specified in a *protocol*. In this paper, we present AFLNET, the first greybox fuzzer for protocol implementations. Unlike existing protocol fuzzers, AFLNET takes a mutational approach and uses state-feedback to guide the fuzzing process. AFLNET is seeded with a corpus of recorded message exchanges between the server and an actual client. No protocol specification or message grammars are required. AFLNET acts as a client and replays variations of the original sequence of messages sent to the server and retains those variations that were effective at increasing the coverage of the code or state space. To identify the server states that are exercised by a message sequence, AFLNET uses the server's response codes. From this feedback, AFLNET identifies progressive regions in the state space, and systematically steers towards such regions. The case studies with AFLNET on two popular protocol implementations demonstrate a substantial performance boost over the state-of-the-art. AFLNET discovered two new CVEs which are classified as critical (CVSS score CRITICAL 9.8).**

> *"One of the things that I struggle with is the limitation AFL seems to have, in that it only performs fuzzing with one input (a file). For many systems such as network protocols, it would be useful if fuzzing could be done on a sequence of inputs. This sequence of inputs might be for example messages necessary to complete a handshake in TLS/TCP."*
>
> — Paul (a member of the AFL's user group) [8]

> *"I'm interested in doing something fairly non-traditional and definitely not currently supported by AFL. I would like to perform fuzzing of a large and complex external server that cannot easily be stripped down into small test cases."*
>
> — Tim Newsham (a member of the AFL's user group) [8]

Fig. 1. Requests from AFL's users asking for stateful fuzzing support

I. Introduction

It is critical to find security flaws in protocol implementations. Protocols are used by internet-facing servers to talk to each other or to clients in an effective and reliable manner. A *protocol* specifies the exact sequence and structure of messages that can be exchanged between two or more online parties. However, this ability to talk to a server from anywhere in the world provides ample opportunities for remote code execution attacks. An attacker does not even require physical access to the machine. For instance, the famous Heartbleed vulnerability is a security flaw in OpenSSL, an implementation of the SSL/TLS protocol which promises secure communication.[1]

However, finding vulnerabilities in protocol implementations is also difficult. There are several challenges for state-of-the-art fuzzing approaches, like coverage-based greybox fuzzing (CGF) [1], [2] and stateful blackbox fuzzing (SBF) [3], [4]. First, a server is stateful and message-driven. It takes a sequence of messages (a.k.a requests) from a client, handles the messages and sends appropriate responses. Yet, the *implemented* protocol may not entirely correspond to the *specified* protocol. Second, the servers' response depends on both, the current message and the current internal server state which is controlled by earlier messages. Meanwhile, vanilla CGF fuzzers like AFL and its extensions [5]–[7] neither know the server state information nor the required structure or order of the messages to be sent. These CGF fuzzers were mainly designed to test stateless programs (e.g., file processing programs) which produce output for the current input, where no internal state is maintained or taken into account.

Developers only have workaround solutions to fuzz protocol implementations using current CGF approaches. They would need to write test harnesses for unit testing of specific program states of the server under test (SUT) [2] or to concatenate message sequences into files and use them as seeds to do normal mutational file fuzzing [1]. These two approaches have several drawbacks. While unit testing is effective at some specific program states, it may not be able to thoroughly test the interactions/transitions between several program states. Moreover, it normally requires a substantial effort to write a new test harness to maintain correct program states and avoid false positives. Importantly, it is not applicable for end-to-end fuzzing to test the whole server whose source code may not be available.

Working on concatenated files leads to inefficiency and ineffectiveness in bug finding. First, for each fuzzing iteration, the whole selected seed file needs to be mutated. Given a file f which is constructed by concatenating a sequence of messages from m_1 to m_n, CGF mutates the whole file f and treats all messages equally. Suppose a message m_i is the most interesting one (e.g., exploring it leads to higher code coverage and potential bugs), CGF repeats mutating uninteresting messages m_1 to m_{i-1} before working on m_i and it has no knowledge to focus on m_i. Second, lacking

[1] See http://heartbleed.com/

state transition information, CGF could produce many invalid sequences of messages which are likely to be rejected by the SUT. Indeed, the users of AFL are well aware of these limitations so they have sent several requests and questions to its developers' group [8]. Figure 1 show two requests from AFL's users asking for stateful fuzzing support.

Due to the aforementioned limitations of CGF on stateful server fuzzing, the most popular technique is still stateful blackbox fuzzing (SBF). Several SBF tools have been developed in both academia (e.g., Sulley, BooFuzz [4], [9]), and in the industry (e.g., Peach, beSTORM [3], [10]). These tools traverse a given protocol model, in form of a finite state machine or a graph, and leverage data models/grammars of messages accepted at the states to generate (syntactically valid) message sequences and stress test the SUT. However, their effectiveness heavily depends on the completeness of the given state model and data model which are normally written manually based on the developers' understanding of the protocol specification and the sample captured network traffic between the client and the server. These manually provided models may not capture correctly the protocols implemented inside the SUT. Protocol specifications contain hundreds of pages of prose-form text. Developers of implementations may misinterpret existing or add new states or transitions. Moreover, like other blackbox approaches, SBF does not retain interesting test cases for further fuzzing. More specifically, even though SBF could produce test cases leading to new interesting states, which have not been defined in its state model, SBF does not retain such valuable test cases for further explorations. It also does not update the state model at run-time.

In this work, we introduce AFLNet– the first stateful CGF (SCGF) tool to address the aforementioned limitations of current CGF and SBF approaches. AFLNet makes automated state model inferencing and coverage guided fuzzing work hand in hand; fuzzing helps to generate new message sequences to cover new states and make the state model gradually more complete. Meanwhile, the dynamically constructed state model helps to drive the fuzzing towards more important code parts by using both the state coverage and code coverage information of the retained message sequences. We evaluated AFLNet on implementations of two well-known protocols: the File Transfer Protocol (FTP) and the Real Time Streaming Protocol (RTSP). Our preliminary results show that AFLNet substantially outperforms the state-of-the-art in terms of code coverage, the coverage of the state space, and bug finding ability. AFLNet exposed two previously unknown security flaws (CVEs assigned) in an RTSP implementation.

We are planning to release the source code of AFLNet at https://github.com/aflnet/aflnet.

II. Example: File Transfer Protocol

We begin with an informal introduction of the main concepts behind server communication and the terminology we are using in this paper. A *server* is a software system that can be accessed remotely, e.g., via the internet. A *client* is a software system that uses the services which are provided by a server. In our setting, the fuzzer acts as a client while the server acts as the fuzz target.

In order to exchange information, both network participants send messages. A *message* is a distinct data packet. A *message sequence* is a vector of messages. A valid order of messages is governed by a protocol. A message from the client is also called *request* while a message from the server is called *response*. Each request may advance the server state, e.g., from initial state to authenticated. The *server state* is a specific status of the server in the communication with the client.

Listing 1 shows an exchange of messages according to the File Transfer Protocol (FTP) between a client and `LightFTP` [11], a server which implements FTP and is one of the subjects in our evaluation. The message sequence sent from the client is highlighted in red. FTP specifies that a client must first authenticate itself at the server. Only after successful authentication can the client issue other commands (i.e., transfer parameter commands and service commands). For each request message from the client, the FTP server replies with a response message containing a status code (e.g., 230 [login successful] or 430 [invalid user/pass]). The status code in the response ensures that client requests are acknowledged and informs the client about the current server state.

```
220 LightFTP server v2.0a ready
USER foo
331 User foo OK. Password required
PASS foo
230 User logged in, proceed.
MKD demo
257 Directory created.
CWD demo
250 Requested file action okay, completed.
STOR test.txt
150 File status okay
226 Transfer complete
LIST
150 File status okay
226 Transfer complete
QUIT
221 Goodbye!
```

Listing 1. Message exchange between an FTP client (red) and the LightFTP server (black) on the `control` channel.

III. Tool Design and Implementation

We implemented our tool AFLNet as an extension of the popular and successful greybox fuzzer AFL [1], [12]. The architecture of AFLNet is shown in Figure 2. To facilitate communication with the server, we first enabled network communication over sockets, which is not supported by the vanilla AFL. AFLNet supports two channels, one to send and one to receive messages/responses from the **Server Under Test**. The response-receiving channel forms the state feedback channel in addition to the code coverage feedback channel as implemented in all CGF approaches. AFLNet uses standard C Socket APIs (i.e., $connect, poll, send$, and $recv$)[2] to implement this feature. To ensure proper synchronization between

[2] http://man7.org/linux/man-pages/man2/socket.2.html

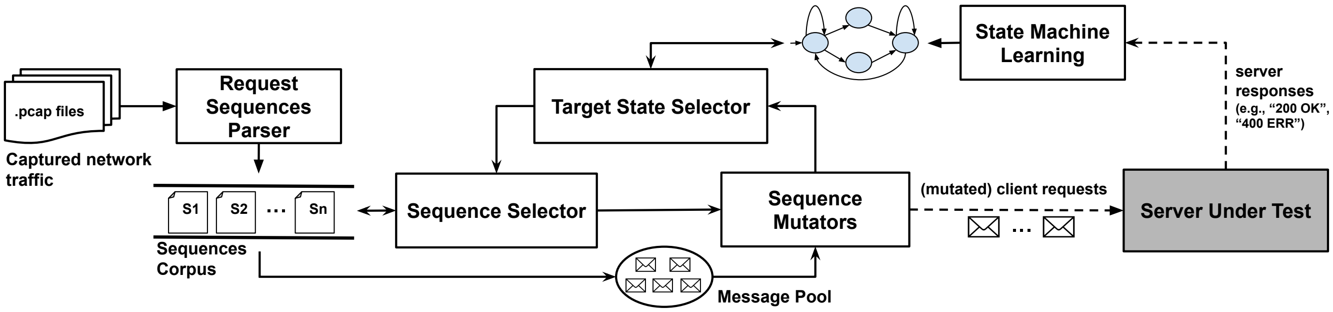

Fig. 2. Architecture and Implementation of Stateful Greybox Fuzzing into AFLNet

AFLNet and the server under test, we added delays between requests. Otherwise, several server implementations drop the connection if a new message is received before the response is sent and acknowledged.

The input for AFLNet are the `pcap` files containing the captured network traffic (e.g., requests and responses between the FTP client and the FTP server as shown in Listing 1). To *record* a realistic message exchange between client and server in a pcap file, a network sniffer (e.g., `tcpdump`[3]) can be used. The relevant message exchange can be extracted using a packet analyzer. For instance, we used the packet analyzer `Wireshark`[4] to automatically extract the sequence of FTP requests.

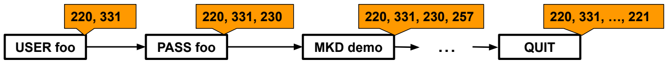

Fig. 3. An annotated FTP message sequence processed for mutational fuzzing (from the sniffer trace in Listing 1)

AFLNet uses its **Request Sequence Parser** component to produce the initial corpus of message sequences. AFLNet uses protocol-specific information of the message structure to extract individual requests, in correct order, from the captured network traffic. It first filters out the responses from the `pcap` files to get the traces of client requests. Then, it parses the filtered traces to identify the start and end of every messages in the trace. We implemented a lightweight method that finds header and terminator of a message as specified in the given protocol. For instance, each FTP message starts with a valid FTP command (e.g., USER, PASS) and is terminated with a carriage return followed by a line feed character (i.e., `0x0D0A`). Moreover, SCGF associates with each message in the sequence the corresponding server state transitions (cf. Figure 3). This is done by sending the messages and parsing the responses one by one.

The **State Machine Learner** takes the server responses and augments the implemented protocol state machine (IPSM) with newly observed states and transitions. AFLNet reads the server response into a byte buffer, extracts the status code as specified in the protocol, and determines the executed states (transitions). A new graph node, which represents a new state, is added if there is a new status code in the server response.

The **Target State Selector** takes information from the IPSM to select that state which AFLNet should focus on next. AFLNet uses several heuristics that can be computed from the statistical data available in the learned IPSM to help **Target State Selector** select the next state. For example, to identify *fuzzer blind spots*, i.e., rarely exercised states, it chooses a state s with a probability that is inversely proportional to the proportion of mutated message sequences that have exercised s (`#fuzz`). In order to maximize the probability of discovering new state transitions, AFLNet chooses a state s with higher priority that has been particularly successful in contributing to an increased code or state coverage when they were previously selected (`#paths`). It is worth noting that AFLNet only starts applying these heuristics once the fuzzing process has been working for long-enough time to accumulate statistical data. At the beginning, the **Target State Selector** randomly selects target states.

Once a target state s has been selected, the **Sequence Selector** selects a message sequence (i.e., a seed input), which can reach the state s, from the sequence corpus. AFL/AFLNet implements the *seed corpus* (here, containing message sequences) as a linked list of queue entries. A *queue entry* is the data structure containing pertinent information about the seed input. In addition, AFLNet maintains a *state corpus* which consists of (i) a list of *state entries*, i.e., a data structure containing pertinent state information, and (ii) a hashmap which maps a state identifier to a list of queue entries exercising the state corresponding to the state identifier. The **Sequence Selector** leverages the hashmap to randomly select a sequence, as represented in a queue entry, to exercise the state s.

The **Sequence Mutator** augments AFL's `fuzz_one` method with protocol-aware mutation operators. AFLNet is a *mutation-based fuzzing approach*, i.e., a seed message sequence is chosen from a corpus and mutated to generate new sequences. There are several advantages over existing *generation-based approaches* which generate new message sequences from scratch. First, a mutation-based approach can leverage a valid trace of real network traffic to generate new sequences that are likely valid—albeit entirely without a proto-

[3] https://www.tcpdump.org/pcap.html

[4] https://www.wireshark.org/

col specification. In contrast, a generation-based approach [4], [9], [13] requires a detailed protocol specification, including concrete message templates and the protocol state machine. Second, a mutation-based approach allows to evolve a corpus of particularly interesting message sequences. Generated sequences that have led to the discovery of new states, state transitions, or program branches are added to the corpus for further fuzzing. This evolutionary approach is the secret sauce of the tremendous success of coverage-based greybox fuzzing.

Given a state s and a message sequence M, AFLNet generates a new sequence M' by mutation. In order to ensure that the mutated sequence M' still exercises the chosen state s, AFLNet splits the original sequence M into three parts: 1) the *prefix* M_1 is required to reach the selected state s, 2) the *candidate subsequence* M_2 contains all messages that can be executed after M_1 while still *remaining* in s, and 3) the *suffix* M_3 is simply the left-over subsequence such that $\langle M_1, M_2, M_3 \rangle = M$. The mutated message sequence $M' = \langle M_1, mutate(M_2), M_3 \rangle$. By maintaining the original subsequence M_1, M' will still reach the state s which is the state that the fuzzer is currently focusing on. The mutated candidate subsequence $mutate(M_2)$ produces an alternative sequence of messages *upon* the choosen state s. In our initial experiments, we observed that the alternative requests may not be observable "now", but propagate to later responses. Hence, AFLNet continues with the execution of the suffix M_3.

AFLNet offers several *protocol-aware mutation operators* to modify the candidate subsequence. From the corpus of message sequences C, AFLNet produces a pool of messages. The *Message Pool* is a collection of actual messages from network sniffer traces (plus generated messages) that can be added or substituted into existing message sequences $M \in C$. In order to mutate the candidate sequence M_2, AFLNet supports the replacement, insertion, duplication, and deletion of messages. In addition to these protocol-aware mutation operators, AFLNet uses the common byte-level operators that are known from greybox fuzzing, such as bit flipping, and the substitution, insertion, or deletion of blocks of bytes. The mutations are *stacked*, i.e., several protocol-aware and byte-level mutation operators are applied to generate the mutated candidate sequence.

Generated message sequences M' that are considered as "interesting" are added to the corpus C. A sequence is considered as *interesting* if the server response contains new states or state transitions that have not previously been observed (i.e., they are not recorded in the IPSM S); a sequence is interesting also if it covers new branches in the server's source code.

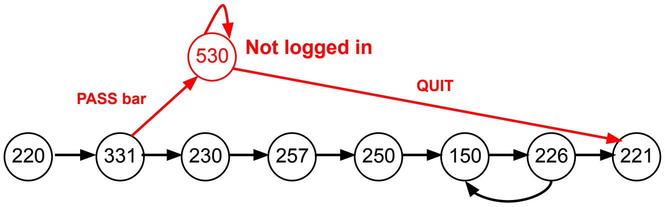

Fig. 4. IPSM learning example

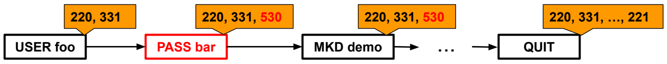

Fig. 5. A sample mutated sequence if the state 331 (User OK) and the message sequence in Figure 3 have been chosen

Now we illustrate how all these components of AFLNet work together to fuzz the LightFTP server. Suppose AFLNet starts with only one pcap file containing the network traffic as shown in Listing 1. First, **Request Sequence Parser** parses the pcap file to generate a single sequence (as visualized in Figure 3) and save it into the corpus C. At the same time, **State Machine Learning** constructs the initial IPSM based on the response codes; this initial IPSM contains black nodes and transitions in Figure 4. Suppose that **Target State Selector** selects state 331 (USER foo OK) as the target state, **Sequence Selector** will then randomly select a sequence from the sequence corpus C, which contains only one sequence at this moment. Afterwards, **Sequence Mutators** identifies the sequence prefix ("USER foo" request), the candidate subsequence ("PASS foo" request), and the remaining subsequence as the suffix. By mutating the candidate subsequence using stacked mutators, **Sequence Mutators** may generate a wrong password request ("PASS bar") leading to an error state (530 Not logged in). Following this wrong password, it replays the suffix (e.g., "MKD demo", "CWD demo") leading to a loop in the state 530 because all these commands are not allowed before a successful authentication. Finally, the "QUIT" request is sent and and the server exits. Since the generated test sequence (as visualized in Figure 5) covers new state and state transitions (as highlighted in red in Figure 4), it is added into the corpus C and the IPSM.

IV. Case Studies

We evaluated the effectiveness of AFLNet in comparisons with two baseline approaches, a stateful blackbox fuzzer (BooFuzz) and a stateless coverage-guided fuzzer (AFLNwe), our network-enabled extension of AFL[5]. Specifically we compared the average branch coverage, state coverage, and number of bugs exposed in 24-hour fuzzing campaigns on two protocol implementations as shown in Figure 6. These two protocols are popular. While FTP has been widely used for file transfer, RTSP is the most common protocol for real-time video streaming which has been implemented in large real-world frameworks like YouTube. Live555, the selected RTSP server in our experiments, has been installed on privacy and security-critical devices like IP Cameras[6].

While AFLNet and AFLNwe are started with an initial seed corpus of recorded message sequences for most common usage scenarios (e.g., upload a file, start streaming a media source), BooFuzz is started with a detailed model of the protocol, including the message templates and the state machine.

[5] To enable network communication, we ported the custom_net_fuzzer component from WinAFL [14] to Linux.

[6] D-Link Camera: http://files.dlink.com.au/products/D-ViewCam/REV_A/Manuals/Manual_v3.51/D-ViewCam_DCS-100_B1_Manual_v3.51(WW).pdf

Subject	Size	Protocol	Description
`LightFTP` [11]	4.7K	**FTP**	Lightweight FTP server.
`Live555` [15]	55.6K	**RTSP**	RTSP-based streaming server.

Fig. 6. Subjects' description and size in kilo of lines of code (KLOC).

Experiment repetition. To mitigate the impact of randomness, for each subject we ran 20 isolated instances of each of BooFuzz, AFLNet, and AFLNwe.

A. Code coverage and state coverage

Figure 7 shows that AFLNet outperforms the state-of-the-art stateful greybox fuzzer BooFuzz with a large effect size (Vargha-Delaney $\hat{A}_{12} > 0.71$) on all measures of effectiveness. The average increase in branch coverage, statement coverage, and state coverage is 60%, 56% and 67%, respectively. We explain this performance increase with AFLNet's ability to mutate real message sequences and to evolve a corpus of message sequences that have been observed to increase the coverage of the server code.

AFLNet also clearly outperforms AFLNwe, especially in LightFTP. The increase in branch coverage, statement coverage, and state coverage is 121%, 79% and 85%, respectively. To understand why AFLNwe and AFLNet are on par for Live555, we have to look at the implemented protocol state machine (IPSM). Firstly, the Live555 IPSM has a smaller depth than the LightFTP IPSM, i.e. the number of messages in a valid sequence is smaller. Secondly, the number of functional states is smaller, i.e, most states (which are not already discovered by the initial sequences) are error states.

B. Vulnerability discovery

Figure 8 shows results on the bug finding capabilities of AFLNet, BooFuzz and AFLNwe. For all fuzzers, we counted the numbers of vulnerabilities found and measured the time they took to expose these vulnerabilities. AFLNet outperforms both fuzzers on all errors. In total, AFLNet discovered four vulnerabilities in which two of them (CVE-2018-4013 and CVE-2019-7733) are known and the remaining two vulnerabilities (CVE-2019-7314 and CVE-2019-15232) are zero-day. Both CVE-2019-7314 and CVE-2019-15232 received the CVSS score CRITICAL 9.8. Given the severity level of these vulnerabilities, the maintainer of Live555 quickly applied patches and acknowledged our findings only two days after the bug reports had been sent. Neither BooFuzz nor AFLNwe was able to discover CVE-2019-7314.

We further analyzed the root cause of CVE-2019-7314 and found that there exists an unspecified shortcut between the INIT and PLAY state (shown in red in Figure 9) when the Setup message contains a RANGE value[7]. While the shortcut itself is harmless, it enables a vulnerability that was found only by our technique. AFLNet generated a random message sequence that discovered this transition, retained the sequence, and systematically evolved it to find a zero-day vulnerability. To exploit the vulnerability, an attacker would need to send a sequence of two messages. The first is a SETUP message with a RANGE value. The second is an arbitrary message of length greater than 20,000 bytes. The attacker can read up to 8 bytes of free'd memory. As the transition is not documented in the standard RTSP specification[8], BooFuzz [4] cannot exercise the unspecified shortcut in Live555.

V. Related Work

A. Coverage-based Greybox Fuzzing

There exist several boosting strategies for greybox fuzzing. By generating more inputs from certain "interesting" seed inputs, a greybox fuzzer can be steered, e.g., towards dangerous [6] or uncovered program statements [5]. Most recently, the community explored the opportunities of making greybox fuzzing aware of the input structure [7], [16]–[18]. In contrast, we suggest to make greybox fuzzing aware of the state space of a stateful program, such as a protocol implementation.

B. Network-Enabled Fuzzing

Many network-enabled fuzzers have been developed, both in academia [9], [13] and industry [3], [4], [10]. Most network-enabled fuzzers take a *blackbox fuzzing* approach, i.e., new message sequences are generated from scratch based on manually constructed protocol specifications. Most network-enabled fuzzers also take a *generation-based* approach, i.e., new message sequences are generated from scratch, using pre-specified message templates. In contrast, SCGF takes a mutation-based approach where new message sequences are generated by mutating existing (recorded) message sequences.

C. Fuzzing Protocol Implementations without Protocol Specifications

Manually constructing a model of the protocol is tedious and error-prone. A better approach is to automatically reverse engineer the protocol either for or during fuzzing. We can distinguish blackbox approaches [19], [20] that learn the message structure from a given corpus of messages and whitebox approaches [21], [22] that actively explore the protocol implementation to uncover message structure. For instance, Polyglot [21] uses dynamic analysis techniques, such as tainting and symbolic execution to extract the message format from the protocol implementation. In terms of *state machine inferencing*, we can distinguish passive learning approaches [23], [24] that learn the protocol state machine from a corpus of message sequences and active learning approaches [25]–[27] that leverage Angluin's L^* algorithm to actively query the protocol implementations with generated message sequences.

In contrast, to these existing approaches, we take a lightweight mutational approach. SCGF requires neither manually constructed message templates nor any message templates inferred. Instead of using message templates to generate new messages, we fuzz actual, real messages. Similarly, in contrast to existing techniques, SCGF does not use the inferred protocol state machine to generate new message sequences. Instead,

[7] Actual comment in the Live555 code file `liveMedia/RTSPServer.cpp` at line 1373: *"This isn't legal, but some clients do this to combine SETUP and PLAY [messages]"*.

[8] RTSP is specified at https://tools.ietf.org/html/rfc7826

		Branch Coverage			Statement Coverage			State Coverage		
		%Increase	$\hat{A}_{12}$	*p-value*	*%Increase*	$\hat{A}_{12}$	*p-value*	*%Increase*	$\hat{A}_{12}$	*p-value*
AFLNET vs AFLNWE	lightftp	121.06 %	1.000	< 0.001	79.45 %	1.000	< 0.001	85.00 %	1.000	< 0.001
	live555	3.49 %	0.335	0.076	2.44 %	0.228	0.003	8.58 %	0.392	0.230
AFLNET vs BOOFUZZ	lightftp	57.73 %	1.000	0.026	49.72 %	1.000	0.026	37.00 %	1.000	0.020
	live555	64.13 %	1.000	0.026	62.09 %	1.000	0.026	100.00 %	1.000	0.019

Fig. 7. Effectiveness. Mean coverage increase (%Increase), effect size ($\hat{A}_{12}$), and statistical significance (p-value) when comparing AFLNET to BOOFUZZ and AFLNWE, respectively. A Vargha-Delaney $\hat{A}_{12}$ measure above 0.71 indicates a large effect size in favor of AFLNET. Statistical significance is computed using the Mann-Whitney U test.

	Time to Error		
Bug ID	**BOOFUZZ**	**AFLNWE**	**AFLNET**
CVE-2018-4013	>24h	$1h\ 21m\ 37s$	$1h\ 18m\ 10s$
CVE-2019-7733	>24h	$2h\ 29m\ 36s$	$1h\ 45m\ 42s$
CVE-2019-7314	>24h	$>24h\ 00m\ 00s$	$1h\ 38m\ 16s$
CVE-2019-15232	>24h	$0h\ 34m\ 21s$	$0h\ 21m\ 26s$

Fig. 8. Bugs found and average time to error comparison.

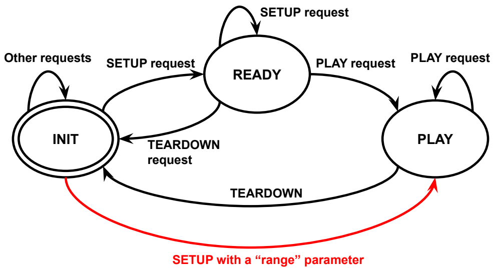

Fig. 9. Root cause analysis for CVE-2019-7314.

existing message sequences within a systematically evolved seed corpus are mutated in a state-centric manner to generate new message sequences.

VI. Future Work

In future work, we plan to conduct more experiments on other popular and critical protocols (e.g., Secure Shell (SSH) and Simple Mail Transfer Protocol (SMTP)) to evaluate the effectiveness and efficiency of AFLNET. Moreover, we also plan to expand the applicability of AFLNET by enhancing its state machine-learning algorithm to support protocol implementations that do not produce response codes.

VII. Acknowledgement

This research was partially funded by the Australian Government through an Australian Research Council (ARC) Discovery Early Career Researcher Award (DE190100046).

References

[1] Website, "American fuzzy lop (afl) fuzzer," http://lcamtuf.coredump.cx/afl/technical_details.txt, 2017, accessed: 2017-05-13.

[2] ——, "Libfuzzer: A library for coverage-guided fuzz testing," http://llvm.org/docs/LibFuzzer.html, 2017, accessed: 2017-05-13.

[3] ——, "Peach Fuzzer Platform," http://www.peachfuzzer.com/products/peach-platform/, 2017, accessed: 2017-05-13.

[4] ——, "Boofuzz: A fork and successor of the sulley fuzzing framework." https://github.com/jtpereyda/boofuzz, 2017, accessed: 2019-08-12.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017.

[7] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[8] Website, "Afl user group," https://groups.google.com/forum/#!forum/afl-users, 2019, accessed: 2019-08-15.

[9] ——, "Sulley: A pure-python fully automated and unattended fuzzing framework." https://github.com/OpenRCE/sulley, 2017, accessed: 2019-08-12.

[10] ——, "beSTORM Black Box Testing," https://www.beyondsecurity.com/bestorm.html, 2017, accessed: 2017-05-13.

[11] ——, "Lightftp server," https://github.com/hfiref0x/LightFTP, 2019, accessed: 2019-08-15.

[12] ——, "Afl vulnerability trophy case," http://lcamtuf.coredump.cx/afl/#bugs, 2017, accessed: 2017-05-13.

[13] ——, "SPIKE Fuzzer Platform," http://www.immunitysec.com, 2017, accessed: 2019-08-12.

[14] I. Fratric, "Winafl: A fork of afl for fuzzing windows binaries," https://github.com/googleprojectzero/winafl#note, 2019, accessed: 2019-08-15.

[15] R. Finlayson, "Live555 media server." http://www.live555.com/mediaServer/, 2006, accessed: 2019-08-15.

[16] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," ser. NDSS '19, 2019.

[17] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," ser. ICSE '19, 2019.

[18] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: Synthesizing structure while fuzzing," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[19] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," TU Darmstadt, Tech. Rep., 2016.

[20] R. Fan and Y. Chang, "Machine learning for black-box fuzzing of network protocols," in *Information and Communications Security*, S. Qing, C. Mitchell, L. Chen, and D. Liu, Eds., 2018.

[21] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, 2007.

[22] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08, 2008.

[23] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks*, B. Thuraisingham, X. Wang, and V. Yegneswaran, Eds., 2015.

[24] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," vol. 10, August 2010.

[25] J. De Ruiter and E. Poll, "Protocol state fuzzing of tls implementations," in *24th USENIX Security Symposium*, ser. USENIX Security '15, 2015.

[26] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery." in *USENIX Security Symposium*, vol. 139, 2011.

[27] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *30th IEEE Symposium on Security and Privacy*, ser. S&P '09, 2009.