Registered Report: NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing

Shisong Qin*, Fan Hu†, Bodong Zhao*, Tingting Yin* and Chao Zhang*✉
*Tsinghua University, {qss19,zbd17,ytt21}@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn
†State Key Laboratory of Mathematical Engineering and Advanced Computing, fanhu_17@foxmail.com

***Abstract*—As the essential component responsible for communication, network services are security-critical, and it is vital to find vulnerabilities in them. Fuzzing is currently one of the most popular software vulnerability discovery techniques, widely adopted due to its high efficiency and low false positives. However, existing coverage-guided fuzzers mainly aim at stateless local applications, leaving stateful network services underexplored. Recently, some fuzzers targeting network services have been proposed but have certain limitations, e.g., insufficient or inaccurate state representation and low testing efficiency.**

In this paper, we propose a new fuzzing solution NSFuzz for stateful network services. Specifically, we studied typical implementations of network service programs and figured out how they represent states and interact with clients, and accordingly propose (1) a program variable-based state representation scheme and (2) an efficient interaction synchronization mechanism to improve efficiency. We have implemented a prototype of NSFuzz, which uses static analysis to identify network event loops and extract state variables, then achieves fast I/O synchronization and efficient state-aware fuzzing via a lightweight compile-time instrumentation. The preliminary evaluation results show that, compared with state-of-the-art network service fuzzers AFLNET and STATEAFL, our solution NSFuzz could infer a more accurate state model during fuzzing and improve the testing throughput by up to 50x and the coverage by up to 20%.

I. INTRODUCTION

Network services are specific implementations of all kinds of network protocols, which define how different entities communicate in the network. However, it introduces more threats to computer systems than local applications, since it is much easier for attackers to exploit vulnerabilities in network services to launch remote attacks than in local applications. For example, the Heartbleed [1] vulnerability from one of the TLS protocol [2] implementations —— OpenSSL [3], could be used by malicious attackers to leak confidential data in the memory of remote devices. In addition, the vulnerability in the implementation of Microsoft's Server Message Block (SMB) protocol [4] has also led to a worldwide WannaCry ransomware cyberattack [5]. Since OpenSSL is a widely used library for TLS encryption communication, and the vulnerable SMB protocol runs on countless Microsoft Windows OS devices, such vulnerabilities have a vast range of influence. Therefore, vulnerabilities of network services are significant threats to the entire cyberspace, and it is vital to explore vulnerabilities in protocol implementations.

Today, fuzzing is one of the most popular vulnerability discovery techniques, and has been widely used and studied in both academia and industry, due to its ease of usage, high efficiency, and low false positives. In the early days, fuzzers for network services mainly worked in black box style [6], [7], which in general blindly and continuously generate and send messages to the service under test (SUT) located at a given IP address and port. Although black box fuzzing is easy to launch, it is relatively blind due to lacking the internal feedback of the SUT during fuzzing, which leads to limited code coverage and vulnerability discovery effectiveness. In recent years, grey box fuzzing solutions that combine genetic algorithms and code coverage feedbacks have become more and more popular [8], [9], [10]. For instance, the representative fuzzer AFL [8] has greatly improved the code coverage and overall fuzzing effectiveness.

However, such traditional grey box fuzzing approaches cannot be directly well applied to network services due to two main challenges: (1) Service state representation. Most existing grey box fuzzers are mainly designed for stateless local applications. The protocol-based network services usually involve state transition during multiple message handling, allowing network service to respond differently according to the current session state when receiving the same input message. Hence, grey box fuzzing solutions without awareness of service states could not acquire complete feedback, which would mislead the evolutionary direction of genetic algorithms. (2) Testing efficiency. Network service programs are in general designed as C/S architecture and action usually involves multiple network I/O interactions, which means that an effective fuzzer has to conduct multiple interactions with the target service. Hence, fuzzers that do not send each message in time with the target service would waste the time of testing.

Notably, there have been some recent attempts to introduce grey box fuzzing for network services. AFLNET [11] first proposed a grey box fuzzing solution targeted at stateful protocol implementations. It extracted the response code from the response messages to represent the service states, then used the response code sequence to infer a state model of the protocol implementation, and further utilized the inferred model to guide the fuzzing process. STATEAFL [12] attempted to use programs' in-memory states to represent the service states, then performed state collection and state model inference by instrumenting the service under test. In each round of network interaction, STATEAFL dumped program variables to an analysis queue and performed post-execution analysis to

International Fuzzing Workshop (FUZZING) 2022
24 April 2022, San Diego, CA, USA
ISBN 1-891562-77-0
https://dx.doi.org/10.14722/fuzzing.2022.23006
www.ndss-symposium.org

update the state model. However, these two network service fuzzers still suffer from the aforementioned two challenges. As for the state representation challenge, the response code scheme proposed by AFLNET assumes the protocol will embed special code in response messages, which is not always the case. Besides, as pointed out in STATEAFL, response code could only provide a poor indication of the network service state and even lead to redundant states. Regarding the testing efficiency challenge, since there is no clear signal indicating whether the SUT has completed processing one message or not, both AFLNET and STATEAFL use a fixed timer to control the fuzzer to send messages to SUT. However, the time window of the timer is either too short (in which case the SUT will miss messages sent by the fuzzer) or too long (in which case the fuzzer will waste too much time on waiting). Besides, STATEAFL requires post-execution analysis for state sequence collection and state model inference, introducing the additional run-time overhead and further lowering the testing throughput.

In this paper, we propose NSFuzz, an efficient state-aware grey box fuzzing solution for network services. We have studied many representative network service programs to understand their typical implementations. We found that such programs always use program variables to describe the program states directly. Besides, we also noticed that the network services always come with a network event loop, which is responsible for dispatching incoming messages. Hence, to address the first challenge, we propose a lightweight *variable-based state representation* scheme to represent the service states with higher accuracy. As for the second challenge, the intrinsic event loop of network services could be utilized to yield appropriate signal feedback, which enables an *efficient synchronization* between network services and the fuzzer and facilitate the fuzzer sending new messages to reduce waiting time overheads. Moreover, the signal-based synchronization could also enable the fuzzer to actively conduct state sequence collection and state model inference, thereby avoiding heavy post-execution analysis used by STATEAFL.

Specifically, we use static analysis to automatically recognize network event loops and state variables from the source code of network services, and conduct lightweight compile-time instrumentation to enable state-aware fuzzing and signal-based fast I/O synchronization. We have implemented a prototype of NSFuzz. The evaluation results showed that, NSFuzz could infer a more accurate state model during the fuzzing process, and has a significantly higher fuzzing throughput than AFLNET and STATEAFL. Besides, NSFuzz could reach higher coverage and trigger crashes in much less time.

In summary, this paper makes the following contributions:

- We propose a variable-based state representation scheme to recognize states of network services and infer their state models, and use it to guide the fuzzing process.
- We propose an efficient synchronization mechanism based on the network event loop of SUT, which enables a much higher throughput for network service fuzzers.
- We have implemented a prototype of NSFuzz and briefly evaluated it on several real-world network services. The preliminary results showed that, NSFuzz could perform state model inference more accurately and achieve much better fuzzing effects than the state-of-the-art network fuzzers.

II. RELATED WORK

A. Black Box Network Fuzzing

Since the fuzzing technique was proposed, early researchers mainly used generation-based fuzzers to perform black box fuzzing. For network services, such methods rely on prior knowledge of protocol format to generate valid testcases for fuzzing. SPIKE [13] used a block-based analysis method to automatically generate valid data blocks of protocol messages with pre-defined generation rules. PROTOS [14] provided some template-based generation and error injection primitives for users to specify particular fields in the protocol format to generate testcases. SNOOZE [15] and KiF [16] proposed a scenario-based fuzzing method, which further required users to pre-build the interaction scenarios by specifying the message order to achieve fuzzing stateful protocols services. Until today, the widely used black box network fuzzing tools such as Peach [6], Sulley [17], and boofuzz [7] are still built on top of these ideas. To reduce the reliance on prior knowledge and manual work before fuzzing, AutoFuzz [18] proposed a more automated fuzzing framework by using network traffic analysis to extract the message format and the protocol state model, and traversed the service state space by modifying the messages as a built-in proxy during fuzzing. AspFuzz [19] would directly change the sending order of generated messages sequence to perform state-level fuzzing. PULSAR [20] went a step further. It guided the fuzzer to traverse less fuzzed subspace in the finite state model, which was built based on the analysis of adjacent messages, to perform adequate fuzzing.

B. Grey Box Network Fuzzing

In recent years, after the grey box fuzzing solutions represented by AFL were proposed, combining program internal feedback and genetic mutation algorithm to conduct fuzzing has attracted much more attention. IoTHunter [21] first applied the grey box fuzzing for stateful network services in IoT devices. It used a multi-stage generation method to perform fuzzing based on coverage feedback. yFuzz [22] performed a similar idea to IoTHunter, while it mainly analyzed the implementation of AFL and proposed a multi-forkserver structure design to achieve multi-stage fuzzing of the stateful protocol services. AFLNET [11] and SGPFuzz [23], etc. [24], [25] used response codes to represent the service state and performed run-time state model building and state-guided fuzzing, and these solutions are relatively more general state-aware fuzzers towards network services. STATEAFL [12] tried to overcome the limitation of the response code based state representation scheme by using in-memory state to represent service state. This method could infer a more reasonable state model, but its vast post-execution analysis overhead still led to a low fuzzing efficiency. Nyx-Net [26] used a snapshot-based method to fastly restore the service state and achieved efficient fuzzing for stateful services, but it lacks fine-grained state analysis and guidance for network applications during the fuzzing process.

C. Program State Model Inference

The automated construction of program state models (especially for stateful protocols) has always been an attractive and widely studied research area. In addition to AutoFuzz and PULSAR, which are based on traffic analysis for automated state model inference, Prospex [27] tried to use dynamic taint

analysis to infer the protocol state model and message format, while PRETT [28] used binary tokens combined with network traces to build minimized state model. Besides, IJON [29] and FazzFactory [30] allowed users to annotate the specific variables in the program under test via provided APIs, using specific feedback to guide the fuzzer to perform domain-specific fuzzing. In addition to AFLNET, STATEAFL, etc. [24], [23] that built state models during fuzzing and performed state guidance to improve the fuzzing effects. Recently some researchers have also used fuzzing as a method to infer the TLS/DTLS protocol state model and verify its security manually to check whether the state model in the implementation of services has logic flaws [31], [32].

III. STUDY ON NETWORK SERVICE

A. Implementation of Network Service

We carry out extensive research on network services programs and find they usually contain three typical stages:

- *Service Initialization Stage.* In this stage, network services perform initial operations such as reading configuration and initializing related data structure based on the startup parameters. Besides, services would use network programming interfaces (such as *socket()* in Glibc) to conduct socket creation, network port binding, and socket listening until remote clients request socket connection.
- *Service Processing Stage.* During the processing stage, network services work in an event loop. They process requests from the client and give responses in this loop. Once the client requests coming, the services parse the message, invoke the corresponding function handler, and return the response message. Network services would exit this stage only when the client actively quits or any exception occurs.
- *Service Cleanup Stage.* When the network services are going to be actively terminated, this stage is responsible for program cleanup such as the resource releasing, then exit to stop providing service.

Since network services need to provide long-term services for arbitrary remote clients, they usually run persistently in the background or run as a daemon. Therefore, the network services program runs most of the time in the second stage, processing the request message sent by the remote client continuously in the network event loop. In addition, it should be noted that developers always use some variables to represent the server state in their implementation for network services running stateful protocols.

Listing 1 shows a *Service Processing Stage* code snippet in a real-world network service implementation Bftpd [33]. From Line3 to Line6 contains a network event loop, where Bftpd receives the request message at Line3 and parses it at Line 5. Bftpd FTP protocol service would repeat these operations until remote client user actively closes the network socket or the service ends itself by any exception. In Bftpd, developers use *commands* structure to store each kind of request's type name, function pointer of the handler, and state requirements. In the *parsecmd()* function, Bftpd first tries to match the type of the incoming request message (Line13), and performs a state check (Line16), then invokes the corresponding handler (Line18) only after passing the state check, otherwise responds a failure message to the client based on current service state.

```
1  int main() {
2    ... // service initialization stage
3    while (fgets(str, MAXCMD, stdin)) {
4      ... // trim "\r\n" from str
5      parsecmd(str);
6    }
7    ... // service cleanup stage
8  }
9  int parsecmd(char *str) {
10   ... // remove garbage in the string
11   for (i = 0; commands[i].name; i++) {
12     // str matches the commands[i]
13     if (!strncasecmp(str, commands[i].name, strlen(
       commands[i].name))){
14       ... // split request type and parameters
15       // state check
16       if (state >= commands[i].state_needed) {
17         // invoke the corresponding handler
18         commands[i].function(str);
19         return 0;
20       } else {
21       switch (state) {
22       case STATE_CONNECTED:
23         response("503 USER expected");
24         return 1;
25       case STATE_USER:
26         response("503 PASS expected");
27         return 1;
28       case STATE_AUTHENTICATED:
29         response("503 RNFR before RNTO expected");
30         return 1;}
31       }
32     }
33   }
34 }
35 // handler for "PASS" request message
36 void command_pass(char *password) {
37   if (state > STATE_USER) {
38     response("503 Already logged in");
39     return;
40   }
41   if (bftpd_login(password)) {
42     state = STATE_CONNECTED;
43     return;
44   }
45 }
```

Listing 1: Simplified Code Snippet from Bftpd v5.7.

During the message handler, Bftpd would conduct specific business processing and update the state of network services.

B. Insight

Based on the study, we find that the network event loop serving as a natural structure of network services could provide appropriate feedback to the fuzzer, indicating the network services have finished a round of message handling. We take the code snippet from Bftpd as an example to illustrate the functionality of the network event loop. In the message processing stage of Bftpd, the entry of the network event loop (Line3) can indicate that the Bftpd server has processed the previous FTP request message and is ready to receive the following FTP request message. Therefore, the entry of the event loop can be used as a synchronization point to actively notify the fuzzer to send the following request message, thereby avoiding useless time waiting. In addition, such synchronization can also tell the fuzzer to perform state collection to avoid post-execution analysis of state transition.

As mentioned above, network services always use variables to represent the run-time service state. For example, when Bftpd receives a `PASS` request message (used for user login), it would invoke the *command_pass* function with the password field of the request as the function parameter (Line18) and executes different code branches according to the current service state (from Line37 to Line44). Bftpd would update the service

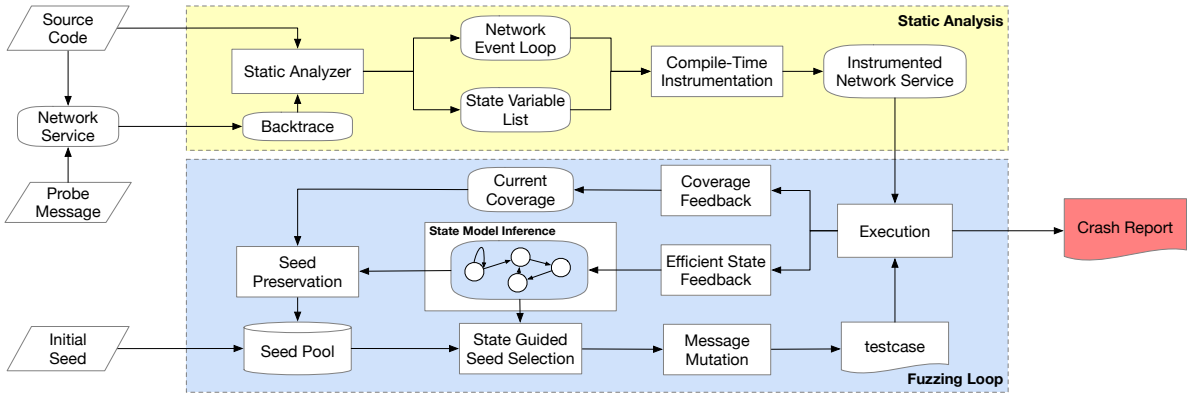

Fig. 1: Workflow of NSFuzz. It first performs static analysis on the source code to recognize state variables and event loops, then instruments the target network service to enable fast I/O synchronization and yield state feedback to guide fuzzing.

state to *STATE_CONNECTED* when login is successful. It should be noted that an enumerated global variable *state* is used to represent the state of the network service. It always gets read (e.g., Line 21) or gets updated (e.g., Line 42) within the network event loop. In addition, compared with the response code, using the program variable to represent the service state is also more accurate and reasonable. In the Bftpd code snippet from Listing 1, response codes in the failure response message under different service states are all *503* (from Line22 to Line30), which shows that the response code cannot be used to distinguish the actual state of network services.

In summary, according to our study, we propose to use specific variables in the network service to represent the service state accurately and use the network event loop as an indicator to achieve efficient I/O synchronization.

IV. METHODOLOGY

A. Overview Design

Figure 1 depicts the workflow of NSFuzz. At the high level, NSFuzz has two main components: static analysis and fuzzing loop. Firstly, NSFuzz takes the network service source code as input and performs static analysis to identify the network event loop (see Section IV-B1) and to extract state variables (see Section IV-B2). Then it uses the result of static analysis to conduct compile-time instrumentation (see Section IV-B3), enabling the target service to have the capabilities of signal-based synchronization and variable-based service state tracing. In the fuzzing loop, NSFuzz takes the initial seeds as input and performs seed selection and message mutation under the guidance of code coverage and state information to generate testcases. When executing testcase, NSFuzz sends a request message each time and waits for the synchronizing signal(see Section IV-C1). Once NSFuzz receives the signal, it collects the program state and updates the state model to carry out state-aware fuzzing(see Section IV-C2) NSFuzz would repeat this process to find program crash until stop the fuzzing loop.

B. Static Analysis

NSFuzz uses static analysis to find two types of information in the service under test. The first is the network event loop, which is responsible for incoming request message handling in the service processing stage. The second is the crucial state variables representing the service state in the program implementation. As we mentioned in III-B, the network event loop is a natural indicator for message handling. Thus it could provide timely feedback to the fuzzer in each loop to avoid time-wasting. Compared with the response code-based state representation scheme, the variable-based state representation scheme could reflect the state of the network service more realistically, thereby inferring a more accurate state model. Besides, instead of StateAFL that needs post-execution online analysis to identify state variables, NSFuzz extracts state variables by using pre-execution offline static analysis and instruments the service under test during compilation to achieve real-time mapping of state variable value to shared memory, thereby further improving fuzzing efficiency.

1) Network Event Loop Identification: The main challenge to identifying the event loop is to distinguish it from other loops in the network service program because there are too many different loops in the implementation. A typical example is in the service initialization stage. Many services may use a file I/O loop to read configurations. Besides, the event loop itself may also contain nested loops, which also brings difficulty for the static analyzer to identify the network event loop accurately. Hence, we trace the network I/O operations in the service processing stage and distinguish the outer loop via backtraces to address the problems.

Firstly, we set breakpoints on input-related system calls such as read, recv, recvmsg, etc., when the network service has completed the initialization and enters the service processing stage. Then the fuzzer establishes a socket connection to the SUT and sends a probe message. The SUT saves the backtrace of the function call stack when breakpoints are hit. Finally, we take the backtrace as an assistance input of our static analyzer to identify the network event loop. The static analyzer first records all the loops contained I/O operation in the service as candidate loops, and scans the backtrace call stack from the bottom (e.g., __libc_start_main) to match the first function (the outer one) that contains an I/O loop, then regards it as network event loops. This is because that the backtrace only contains the call stack within the network event loop during the service processing stage, and matching the outer function that contains an I/O loop could also avoid nested loops.

2) State Variable Extraction: After identifying the network event loop, the static analyzer then extracts the state variables. Due to the lack of run-time information, static analysis always has false positives. Therefore, based on our study about the network service implementation and our analysis of the char-

acteristics of state variables in the service programs, we use the following heuristic rules for the static analyzer to reduce false positives when extracting state variables.

- The state variables related operations in network services are always executed in the network event loop. Hence the static analyzer only performs analysis within the network event loop to reduce the analysis range.
- The state variables in network services are always read (for state checking) or written (for state updating) in the network event loop or message handlers. Hence the static analyzer only extracts variables that are both loaded and stored.
- The state variables of network services are often global enumeration variables or integer member variables in data structures, and they are only be assigned constants to represent a specific state. Hence the static analyzer only keeps global integer variables or user-defined structure members assigned constant values in their store operation.

According to the above heuristic rules, the static analyzer could extract all state variables from services with relatively low false positives and assign a unique string ID to each variable.

3) Compile-Time Instrumentation: After obtaining the network event loop and the list of state variables through static analysis, NSFuzz performs two types of instrumentation on the target network service at compile time. On the one hand, NSFuzz inserts a *raise (SIGSTOP)* statement at the entry of the network event loop, thus the service under test could raise signal feedback to the fuzzer after each request message has been processed, to indicate that it is ready for receiving the following request message. On the other hand, to pass the state variable's value feedback to the fuzzer engine in real-time, NSFuzz instruments the *STORE* operation of each state variable, which uses the variable value being written as the key to update the state-related memory shared between the SUT and the fuzzer. To distinguish it from the shared memory that records code coverage, we denote the shared memory that records state information as *shared_state.* The specific mapping method is as follows:

$$shared\_state[hash(var\_id) \oplus cur\_store\_val] = 1;$$

$$shared\_state[hash(var\_id) \oplus pre\_store\_val] = 0;$$

Firstly, NSFuzz hashes the unique string ID of each state variable and XORs the hash value with the new state value to be written, then uses the XOR result as an index to update the *shared_state*. Besides, NSFuzz also uses the same method to calculate the index based on the previous old state value to restore the *shared_state*. After that, NSFuzz has generated the instrumented program as a service under test for fuzzing.

C. Fuzzing Loop

After using static analysis to obtain the two types of information and instrument the target service, NSFuzz then starts to fuzz the instrumented service under test. Compared with the traditional coverage-guided grey box fuzzers, NSFuzz introduces a signal raising feedback, so the fuzzer also needs to cooperate with the instrumented signal feedback to achieve fast I/O synchronization during the testcase execution. In addition, the instrumentation at the *STORE* operation of the state variable makes NSFuzz also need to check the *shared_state* at an appropriate time and collect the state transition sequence to help infer the state model.

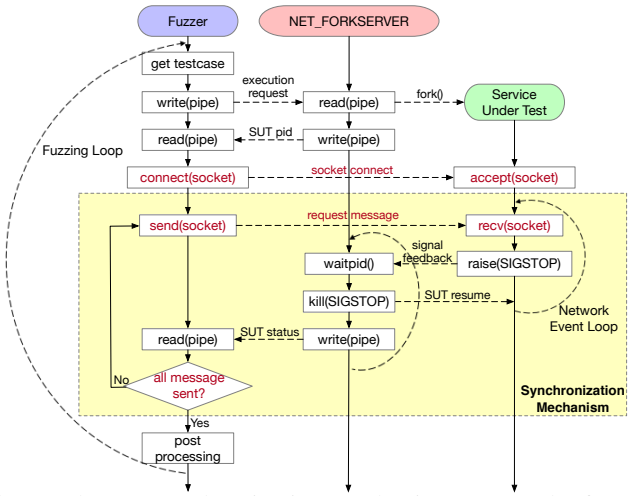

Fig. 2: The I/O synchronization mechanism among the fuzzer, NET_FORKSERVER, and service under test.

1) Fast I/O Synchronization: To improve fuzzing efficiency, AFL introduced FORKSERVER to be responsible for the fork creation and recycling of the target under test. Both AFLNET and STATEAFL are implemented based on AFL. When they execute a testcase, they first notify the FORKSERVER to create a process to be fuzzed, then send each request message in turn at a manually specified time interval and finally wait for the FORKSERVER to write the execution result through the communication pipe after the service ended.

NSFuzz implements NET_FORKSERVER to cooperate with the signal feedback from service under test to achieve fast I/O synchronization, thus avoiding the manually specified time wait interval. In such cases, each time the fuzzer of NSFuzz sends a request message, it waits for feedback from NET_FORKSERVER through the pipe. As the parent process of the fuzzed network service, NET_FORKSERVER waits directly for the raised signal from the service under test to determine whether the target has completed a round of I/O interaction or just crashed based on the signal type, then passes such information back to the fuzzer. Figure 2 shows the I/O synchronization among the fuzzer, NET_FORKSERVER, and service under test in the NSFuzz framework.

2) State-Aware Fuzzing: Whenever the fuzzer receives the message processing result from NET_FORKSERVER via the communication pipe, it calculates the hash of the *shared_state* buffer. This hash can be used to represent the current state of the SUT, and the reason is as follows. If a message causes a state transition, it will update the values of certain state variables, which will then update *shared_state* buffer. As a result, after this message is processed, the fuzzer will get a hash of the *shared_state* buffer which is different from the one before this message. In other words, whenever the SUT state changes, the hash of the buffer will change. Therefore, the fuzzer could use the buffer's hash to collect a state transition sequence for inferring the state model after each synchronization. Compared with STATEAFL, which continuously dumps all the variable values to an analysis queue during each message processing and performs post-execution analysis to conduct state inference, NSFuzz uses the *shared_state* hash to represent the state more adequately. This could fully use the variable-based state representation scheme, thereby achieving more efficient state feedback. Apart from new code coverage, NSFuzz would also

TABLE I: The target services from ProFuzzBench used for preliminary evaluation.

Target Service	Network Protocol	Version/Commit	Transport Layer	Language
LightFTP	FTP	5980ea1	TCP	C
Bftpd	FTP	v5.7	TCP	C
Pure-FTPd	FTP	c21b45f	TCP	C
Exim	SMTP	38903fb	TCP	C
Dnsmasq	DNS	v2.73rc6	UDP	C
TinyDTLS	DTLS	06995d4	UDP	C
Kamailio	SIP	2648eb3	UDP	C

preserve the testcase that triggered new service state or state transition as an interesting seed for subsequent fuzzing. And then NSFuzz would perform state-guided seed selection and message mutation similar to AFLNET.

V. EVALUATION

We have built a prototype of NSFuzz. The implementation of NSFuzz adds about 4.5k lines of C/C++ code and about 100 lines of Python script. In detail, we implement the static analyzer and compile-time instrumentation based on LLVM [34] framework, and the fuzzer engine is implemented based on AFLNET. To elaborate the preliminary evaluation of NSFuzz, we have performed several experiments to answer the following research questions:

- **RQ1: Accurateness of state module inferred by NSFuzz:** Could NSFuzz inference relatively more accurate state model based on the state variables during the fuzzing loop?
- **RQ2: Effectiveness of NSFuzz state-aware fuzzing:** Could NSFuzz achieve higher fuzzing efficiency and overview results than other existing approaches?

A. Experiment Setup

We selected some fuzzing targets in the network protocol fuzzing benchmark, ProFuzzBench [35] to evaluate NSFuzz. ProFuzzBench is a benchmark for stateful protocol fuzzing which contains 13 network service implementations from 10 network protocols (including FTP, SMTP, SIP, etc.), and it covers various network protocols based on TCP and UDP with all implemented in C/C++. In addition, ProFuzzBench applies necessary patches (such as derandomization) for these network services to ensure the reliability of the fuzzing evaluation. In this preliminary evaluation, we currently chose 7 network services in ProFuzzBench as the evaluation targets of NSFuzz. Table I shows the information of the target services we chose.

To make the comparison, we selected two state-of-the-art grey box network fuzzers, AFLNET and STATEAFL, and another network-enabled version of AFL, AFLNWE [36] as baseline fuzzers to evaluate NSFuzz. AFLNET uses message response codes to represent the service state and inference the state model, then conducts state-guided fuzzing based on such model during the fuzzing loop. STATEAFL collects the changed variables during network I/O rounds. Then extracts state variables and infers the state model through post-execution analysis. AFLNWE is another network service fuzzer proposed by the author of AFLNET. It only changes the file I/O interface from the original AFL to socket-based network I/O to achieve network service fuzzing.

All experiments were running on the same testing machine during this evaluation, and the testing machine contains 128 Intel(R) Xeon(R) Platinum 8358 CPUs and 384GB of SSD memory. We built each target service with each fuzzer in docker containers and used the same config for experimental evaluation. We fuzzed each target service with different fuzzer containers for 12 hours and repeated 4 times for a total of 1536 CPU hours of fuzzing evaluation.

TABLE II: The static analysis result of NSFuzz. LoC means the line of code of target services. $\surd$ denotes NSFuzz could identify the network event loop of the corresponding target.

Target Service	LoC	Network Event Loop	State Variable		Analysis Time
			Number	Example	
LightFTP	4.4k	$\surd$	1	Access	0.7s
Bftpd	4.7k	$\surd$	6	state	1.8s
Pure-FTPd	30k	$\surd$	22	loggedin	3.9s
Exim	101.7k	$\surd$	58	helo_seen	45.1s
Dnsmasq	27.6k	$\surd$	15	found	11.4s
TinyDTLS	10.8k	$\surd$	4	state	3.2s
Kamailio	766.7k	$\surd$	58	state	441.9s

B. State Module Inference Evaluation (RQ1)

1) Static Analysis: NSFuzz first performed static analysis to identify the network event loop and used such loop as starting point to extract state variables. Table II shows the static analysis results of NSFuzz for 7 target services.

As we can see, NSFuzz could successfully locate the network event loop from different target services, which involves multiple network protocols, and NSFuzz could also extract state variables via static analysis. Besides, the number of extracted state variables and the analysis time are generally positively correlated with the scale of the target services (LoC), which is consistent with intuition. It should be noted that the name of the extracted state variable could sometimes directly indicate its function of representing the state of the network service. For example, one of the state variables extracted from Pure-FTPD [37] by static analysis is *loggedin*, which is a global variable used to indicate whether the incoming client session has completed the FTP login authorization. Moreover, the message handler could execute different code logic for the same request message according to whether the client session has completed the authorization in Pure-FTPD. Nevertheless, even though multiple heuristic rules have been introduced, the static analysis results may still contain some false positives.

2) Inferred State Module: After extracting the state variables and completing the instrumentation, NSFuzz performed state-aware fuzzing on target network services. Like AFLNET and STATEAFL, NSFuzz would also infer a state model for the service under test during the fuzzing process. Table III shows the average number of vertexes and edges of the state model inferred these fuzzers for the target service in the 12-hour fuzzing experiment.

Here we take LightFTP [38] as an example for a case study. After the 12 hours fuzzing of LightFTP, NSFuzz inferred a state model with 5 vertexes and 12 edges, as shown in Figure 3. In this target, the static analyzer extracted one state variable named *Access*, which is used to represent the access authority of client sessions. After analyzing the source code manually, we found that *Access* has 4 constant values to represent different permission of the client user (NOT_LOGGED_IN, READONLY, CREATENEW, FULL), and LightFTP would conduct different message handling processes according to the user permission. Note that the state model inferred by

TABLE III: The average number of vertexes and edges of state models inferred by different fuzzers.

Target Service	Fuzzer	State Module	
		Vertexes	Edges
LightFTP	AFLNET	23	158
	STATEAFL	11	47
	NSFuzz	5	12
Bftpd	AFLNET	24	126
	STATEAFL	4	6
	NSFuzz	43	137
Pure-FTPd	AFLNET	27	260
	STATEAFL	7	22
	NSFuzz	8	22
Exim	AFLNET	12	60
	STATEAFL	7	17
	NSFuzz	128	225
Dnsmasq	AFLNET	89	271
	STATEAFL	108	467
	NSFuzz	3	5
TinyDTLS	AFLNET	9	24
	STATEAFL	29	69
	NSFuzz	32	115
Kamailio	AFLNET	13	93
	STATEAFL	4	4
	NSFuzz	99	328

NSFuzz contains all these 4 states with an additional initial dumb state, which showed that NSFuzz could accurately infer all states during the fuzzing process on LightFTP, thereby establishing a direct mapping between state variables and the model. On the other hand, the state model inferred by AFLNET and STATEAFL with the same initial seeds respectively had 23 vertexes/158 edges and 11 vertexes/47 edges at the end of fuzzing. However, according to our manual analysis, these state models could not distinguish the different permissions of client users, which may lead to incomplete state guidance. Moreover, these models are also difficult to reflect a clear relationship with the target service. Therefore, to a certain extent, the state model inferred by NSFuzz is relatively more accurate and interpretable than other works.

It is worth noting that even though for all FTP protocol services, the inferred state model (number of vertexes and edges) in different implementations would also be different. As we mentioned above, the static analyzer would not only extract the variables used to represent the service's internal state, but it would also identify some variables such as flag or mode in the program as output, which is a refinement of the state of service implementations. However, this shows that NSFuzz has the ability to identify finer-grained states in the network service. In this case, the state model inferred by NSFuzz by default is an extension of the basic state model of the protocol. However, since the static analysis in NSFuzz is a decoupled module from the fuzzing loop, users could also choose to specify which state variables to monitor to construct a state model, achieving state model inference with different granularities.

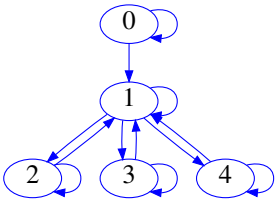

Fig. 3: The inferred state model of LightFTP.

TABLE IV: The average time for the fuzzers to trigger the first crash among 4 runs experiments.

Target Service	First Crash Trigger Time (s)			
	AFLNET	AFLNWE	STATEAFL	NSFuzz
Dnsmasq	1551s	788.25s	765.25s	**583s**
TinyDTLS	26s	11.75s	47.75s	**< 1s**

C. Fuzzing Efficiency Evaluation (RQ2)

To evaluate the efficiency of NSFuzz, we have counted the average results of NSFuzz and other fuzzers among 4 running experiments. Table V and Table IV illustrate the average fuzzing throughput, final code branch coverage, and the time to trigger the first crash during fuzzing evaluation.

1) Fuzzing Throughtput: The fuzzing throughput of fuzzer demonstrates the number of testcases executed per second. Obviously, a higher fuzzing throughput indicates that the program under test has executed more testcases, and the overall test efficiency is also higher.

As shown in Table V, the fuzzing throughput of NSFuzz is significantly better than that of AFLNET and other works. NSFuzz has improved various from 1x to more than 50x throughput in different target services. Because STATEAFL needs to collect variable values during the fuzzing process for post-execution analysis to perform state model inference, the additional overhead introduced by STATEAFL causes a decline in its fuzzing throughput compared with AFLNET. It is worth noting that AFLNWE, the only fuzzer without state-aware, also improves fuzzing throughput compared to AFLNET. This is mainly because that AFLNWE is just a network I/O enabled version of AFL, which only sends one-time data to the service under test, thus saving the waiting delay and state-related overhead in multiple network interactions. However, even in this case, its fuzzing throughput is still not as good as NSFuzz, which proves that the synchronization mechanism of NSFuzz based on lightweight instrumentation could significantly improve the overall fuzzing efficiency.

2) Code Coverage: Code coverage is always a standard metric for evaluating fuzzers, which indicates how much code in the service under test has been executed during the whole fuzzing process. Usually, the higher the code coverage, the more program vulnerabilities may be triggered.

Table V illustrates that NSFuzz could achieve a higher code branch coverage on almost all target services. Moreover, although AFLNWE has a relatively high fuzzing throughput among these fuzzers, it could not achieve good code coverage when fuzzing on network services due to its lack of multiple network I/O interactions and state guidance. Besides, the average number of code branches covered by STATEAFL during the 12-hour fuzzing is not significantly different from AFLNET. Figure 4 shows the growth of the number of code branches explored with the fuzzing time during the 12-hour fuzzing process among several fuzzers. As can be seen from the figure, NSFuzz could not only cover more code branches on most target services but also could explore the branches much faster than any other fuzzers.

3) Crash Trigger: In addition to fuzzing throughput and code coverage, the triggering of the crash directly reflects the vulnerability discovery capabilities of fuzzers. Table IV

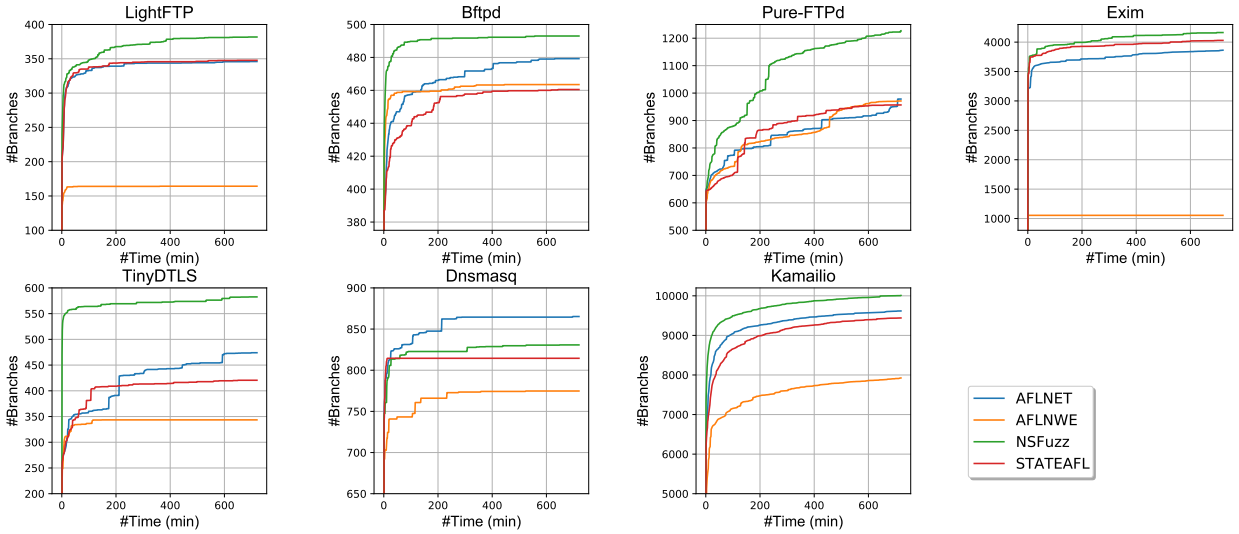

Fig. 4: The average code branch coverage growth of various fuzzers among 4 runs of 12 hours toward each target service. Note that the initial seeds used by target services have covered some code branches, and the y-axis mainly displays the new coverage triggered after the seeds, and thus it does not start from 0.

TABLE V: The average fuzzing throughput and final code branch coverage of various fuzzers among 4 runs of 12 hours toward each target service compared to AFLNET. The AFLNET column displays the absolute number of these metrics, and all other columns denote the changes compared to AFLNET.

Target Service	Fuzzing Throughput (exec/s)				Final Code Branch Coverage			
	AFLNET	AFLNWE	STATEAFL	NSFuzz	AFLNET	AFLNWE	STATEAFL	NSFuzz
LightFTP	8.42	+330.8%	-55.6%	**+558.9%**	346	-52.5%	+0.5%	**+10.3%**
Bftpd	4.09	+144.0%	-45.2%	**+869.7%**	479.25	-3.3%	-3.9%	**+2.9%**
Pure-FTPd	5.29	+115.3%	-80.0%	**+175.0%**	978.25	-0.7%	-2.2%	**+25.5%**
Exim	2.69	+108.6%	+35.3%	**+113.4%**	3862.5	-72.7%	+4.3%	**+7.8%**
Dnsmasq	7.24	+560.6%	-84.4%	**+669.1%**	**865.25**	-10.5%	-5.9%	-4.0%
TinyDTLS	2.66	+458.3%	-47.0%	**+5488.0%**	474	-27.5%	-11.3%	**+22.9%**
Kamailio	5.19	+20.8%	-49.7%	**+512.5%**	9617.5	-17.6%	-1.8%	**+4.1%**

illustrates the time taken by different fuzzers to trigger the first crash during the fuzzing experiments on target network services. As shown in the table, NSFuzz could trigger crashes on the same targets compared to other fuzzers, and the average time to trigger the first crash is significantly lower than other competitors. Especially during the fuzzing process of TinyDTLS network service, NSFuzz could always trigger the program crash in less than 1s, which to some extent shows that NSFuzz could trigger potential vulnerabilities in the target program faster when fuzzing network services.

VI. Future Work

Currently, we preliminary evaluate NSFuzz under only some of the targets in ProFuzzBench. The main reason is that the static analysis part of NSFuzz still has some limitations. Firstly, the static analysis part of NSFuzz currently could only support analyzing C language targets but does not support the analysis of C++ targets. This is mainly due to the indirect calls generated by virtual functions in C++ that make it difficult to build an accurate call graph which affects the result of static analysis. Secondly, we notice that some network applications use event-driven libraries (e.g., libevent) to build their service programs. In this case, the network event loop we defined is implemented in the library instead of the service code, which brings challenges for the static analyzer to find the point to instrument raise function for achieving synchronization. Inspired by IJON [29], etc. [30], we plan to introduce an annotation mechanism into NSFuzz as a supplement to the proposed static analyzer. We will design several APIs to help users annotate state variables and synchronization points manually in the SUT, which is expected to improve the applicability and scalability of NSFuzz.

In addition, the current preliminary evaluation lacks sufficient ablation study, i.e., assessments of the impact of variable-based state representation scheme on the overall fuzzing efficiency. Therefore, we will conduct the corresponding ablation experiments in the future to study the influence of accurate state representation in network service fuzzing.

VII. Conclusion

In this report, we analyzed the problems of state representation and testing efficiency of existing network service fuzzers. Moreover, according to our study on the implementation of network services, we proposed a high-efficiency fuzzing framework combined with variable-based state representation and synchronization mechanism. Then, we implemented the prototype of NSFuzz by using static analysis and lightweight compile-time instrumentation to achieve state-aware fuzzing. Finally, we preliminary evaluated NSFuzz on ProFuzzBench, and the results showed that NSFuzz could inference a relatively more accurate state model to guide the fuzzing. Besides, NSFuzz could achieve higher fuzzing throughput, reach higher code coverage, and trigger crashes in a shorter time than other stateful network fuzzers.

REFERENCES

[1] "The Heartbleed Bug," https://heartbleed.com/, 2020.

[2] "The Transport Layer Security (TLS) Protocol Version 1.3," https://datatracker.ietf.org/doc/html/rfc8446, 2018.

[3] "OpenSSL," https://www.openssl.org/, 2021.

[4] "Server Message Block (SMB) Protocol Versions 2 and 3," https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-smb/, 2021.

[5] "WannaCry ransomware attack," https://en.wikipedia.org/wiki/WannaCry_ransomware_attack, 2021.

[6] M. Security, "Peach," https://github.com/MozillaSecurity/peach, 2021.

[7] J. Pereyda, "boofuzz: Network Protocol Fuzzing for Humans," https://github.com/jtpereyda/boofuzz, 2021.

[8] M. Zalewski, "american fuzzy lop," https://github.com/google/AFL, 2021.

[9] Google, "Honggfuzz," https://github.com/google/honggfuzz, 2021.

[10] "LibFuzzer," https://llvm.org/docs/LibFuzzer.html, 2021.

[11] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: A Greybox Fuzzer for Network Protocols," in *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[12] R. Natella, "StateAFL: Greybox Fuzzing for Stateful Network Servers," *arXiv preprint arXiv:2110.06253*, 2021.

[13] D. Aitel, "The advantages of block-based protocol analysis for security testing," *Immunity Inc., February*, vol. 105, p. 106, 2002.

[14] R. Kaksonen, M. Laakso, and A. Takanen, "Software security assessment through specification mutations and fault injection," in *Communications and Multimedia Security Issues of the New Century*. Springer, 2001, pp. 173–183.

[15] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr," in *International conference on information security*. Springer, 2006, pp. 343–358.

[16] H. J. Abdelnur, R. State, and O. Festor, "KiF: a stateful SIP fuzzer," in *Proceedings of the 1st international Conference on Principles, Systems and Applications of IP Telecommunications*, 2007, pp. 47–56.

[17] OpenRCE, "Sulley," https://github.com/OpenRCE/sulley, 2012.

[18] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *IJCSNS*, vol. 10, no. 8, p. 239, 2010.

[19] T. Kitagawa, M. Hanaoka, and K. Kono, "Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols," in *The IEEE symposium on Computers and Communications*. IEEE, 2010, pp. 202–208.

[20] 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.

[21] B. Yu, P. Wang, T. Yue, and Y. Tang, "Poster: Fuzzing iot firmware via multi-stage message generation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2525–2527.

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

[23] Y. Yu, Z. Chen, S. Gan, and X. Wang, "SGPFuzzer: A State-Driven Smart Graybox Protocol Fuzzer for Network Protocol Implementations," *IEEE Access*, vol. 8, pp. 198 668–198 678, 2020.

[24] C. Song, B. Yu, X. Zhou, and Q. Yang, "SPFuzz: a hierarchical scheduling framework for stateful network protocol fuzzing," *IEEE Access*, vol. 7, pp. 18 490–18 499, 2019.

[25] H. He and Y. Wang, "PNFUZZ: A Stateful NETWORK PROTOCOL FUZZING APPROACH BASED ON PACKET CLUSTERING."

[26] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-Net: Network Fuzzing with Incremental Snapshots," *arXiv preprint arXiv:2111.03013*, 2021.

[27] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 110–125.

[28] C. Lee, J. Bae, and H. Lee, "PRETT: Protocol Reverse Engineering Using Binary Tokens and Network Traces," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2018, pp. 141–155.

[29] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1597–1612.

[30] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[31] J. De Ruiter and E. Poll, "Protocol State Fuzzing of {TLS} Implementations," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 193–206.

[32] 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)*, 2020, pp. 2523–2540.

[33] J. Smith, "Bftpd," http://bftpd.sourceforge.net/, 2021.

[34] "LLVM," https://llvm.org/, 2021.

[35] R. Natella and V. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[36] V. Pham, "AFLNWE," https://github.com/thuanpv/aflnwe, 2021.

[37] F. Denis, "Pure-FTPd," https://www.pureftpd.org/project/pure-ftpd/, 2021.

[38] "LightFTP," https://github.com/hfiref0x/LightFTP, 2021.