NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing

SHISONG QIN, Tsinghua University, China
FAN HU, State Key Laboratory of Mathematical Engineering and Advanced Computing, China
ZHEYU MA, BODONG ZHAO, TINGTING YIN, and CHAO ZHANG, Tsinghua University, China

As an essential component responsible for communication, network services are security critical, thus, it is vital to find their vulnerabilities. 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, for example, insufficient or inaccurate state representation and low testing efficiency.

In this article, we propose a new fuzzing solution NSFuzz for stateful network services. We studied typical implementations of network service programs to determine how they represent states and interact with clients. Accordingly, we propose (1) a program variable–based state representation scheme and (2) an efficient interaction synchronization mechanism to improve fuzzing efficiency. We implemented a prototype of NSFuzz, which uses static analysis and annotation application programming interfaces (APIs) to identify synchronization points and state variables within the services. It then achieves fast I/O synchronization and accurate service state tracing to carry out efficient state-aware fuzzing via lightweight compile-time instrumentation. The evaluation results show that compared with other network service fuzzers, including AFLNET and STATEAFL, our solution NSFuzz could infer a more accurate state model during fuzzing and improve fuzzing throughput by up to 200×. In addition, NSFuzz could improve code coverage by up to 25% and trigger more crashes in less time. We also performed a fuzzing campaign to find new bugs in the latest version of the target services; 8 zero-day vulnerabilities have been found by NSFuzz.

CCS Concepts: • **Security and privacy** → **Software and application security**; • **Networks** → **Protocol testing and verification;**

Additional Key Words and Phrases: Network service, fuzzing, vulnerability discovery

S. Qin and F. Hu contributed equally.

This project has received funding in part from the National Key Research and Development Program of China (grant no. 2021YFB2701000) and the National Natural Science Foundation of China (grant no. 61972224).

Authors' addresses: S, Qin, Z. Ma, B. Zhao, T. Yin, and C. Zhang (corresponding author), Tsinghua University, 30 Shuangqing Rd, Beijing, China; emails: {qss19, mzy20, zbd17, tty21}@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn; F. Hu, State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, Henan, China; email: fanhu_17@foxmail.com.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2023 Copyright held by the owner/author(s).

1049-331X/2023/09-ART160 $15.00
https://doi.org/10.1145/3580598

ACM Reference format:
Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. 2023. NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 160 (September 2023), 26 pages.
https://doi.org/10.1145/3580598

1 INTRODUCTION

Network services are specific implementations of all kinds of network protocols, which define how different entities communicate in the network. However, they introduce 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 [2] vulnerability from one of the most famous implementations of the Transport Layer Security (TLS) protocol [1] — OpenSSL [9], 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 [6] has also led to a worldwide WannaCry ransomware cyberattack [8]. 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 discover vulnerabilities in such targets.

Fuzzing is one of the most popular vulnerability discovery techniques. It 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 [37, 41], which blindly and continuously generated sent 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 combining genetic algorithms and code coverage feedback have become increasingly popular [3, 26, 47]. For instance, the representative fuzzer AFL [47] has dramatically improved the code coverage and overall fuzzing effectiveness on most of the command-line applications, such as `readelf` [5].

However, 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. As for protocol-based network services, on the one hand, network services respond differently according to the current session state when receiving the same input message; on the other hand, most bugs are *stateful*, which can be triggered only by a sequence of specific messages. 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 always designed as C/S architecture. Action usually involves multiple network I/O interactions, which means that an effective fuzzer needs to conduct multiple interactions with the target service. Hence, fuzzers should send each message to the target service in time to save testing time and improve the testing throughput.

Notably, some recent research works have introduced grey-box fuzzing for network services. AFLNET [39] 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 [33] attempted to use programs' in-memory states to represent the service states, then performed state collection and state model inference by instrumenting the SUT. In each round of network interaction, StateAFL dumped program variables to an analysis queue and performed post-execution analysis to update the state model.

However, existing works still suffer from the aforementioned two challenges. As for the state representation challenge, the response code scheme proposed by AFLnet assumes that the protocol will embed special code in response messages, which is not always the case. In addition, as pointed out in StateAFL, the indication of the network service state provided by the response code is not robust. To overcome the limitation of the response code–based method, StateAFL used the program in-memory state to represent the service state. However, due to the complexity of the program's in-memory state, it is unrealistic to map such contents into the service state directly. Hence, StateAFL used locality-sensitive hashing to approximate the state mapping, introducing less accurate state representation. SGFuzz [18] proposed to use state variables to represent the state of network services. It automatically recognized so-called state variables and used them to build a state transition tree (STT), which was considered to represent the explored state space of the service program. However, SGFuzz may introduce false positives in the state representation since SGFuzz directly uses variables with enumeration types as state variables without filtering. Regarding the testing efficiency challenge, since there is no clear signal indicating the message process of the SUT, both AFLnet and StateAFL use a fixed timer to control the fuzzer to send messages to the 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 waiting). StateAFL also requires post-execution analysis for state sequence collection and state model inference, introducing additional runtime overhead and further lowering the testing throughput.

In this article, 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 service states directly. We also noticed that the network services always come with a network event loop, which is responsible for continuously processing incoming messages. Hence, to address the first challenge, we propose a lightweight *variable-based state representation* scheme to represent the network service state. We refer to the variable denoting the service state as the **state variable**. Since state variables contain the inherent semantic information of the service program, the variable-based state representation scheme could represent the service state with higher accuracy and interpretability. As for the second challenge, the intrinsic event loop of network services could yield appropriate signal feedback, enabling an *efficient I/O synchronization* between network services and the fuzzer. We refer to the location in the event loop where the signal feedback can be raised as the **I/O synchronization point** since it is responsible for synchronizing the I/O interaction with the fuzzer. Signal-based synchronization could facilitate the fuzzer sending new messages to reduce waiting time overheads. This mechanism could also enable the fuzzer to collect state transition sequences and infer the state model actively, thereby avoiding heavy post-execution analysis used by StateAFL.

We use both static analysis and annotation APIs to identify I/O synchronization points and state variables from the source code of the SUT. Then, we conduct lightweight compile-time instrumentation to enable the service with signal-based fast I/O synchronization and variable-based service state tracing capability. Finally, we use the instrumented target service to carry out efficient state-aware network service fuzzing. Currently, 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. In addition, NSFuzz could reach higher code coverage and trigger more crashes in less time.

In summary, this article makes the following contributions:

- We propose a variable-based state representation scheme to represent the network services state and infer more accurate state models during fuzzing. We design an efficient I/O synchronization mechanism based on the network event loop of the SUT, which enables a much higher throughput for network service fuzzing.
- We present NSFuzz, an efficient and state-aware network service fuzzing solution. We use static analysis and annotation API to identify the synchronization points and state variables within the SUT, then enable it with signal feedback and state feedback capabilities through compile-time instrumentation for efficient state-aware fuzzing.
- We have implemented a prototype of NSFuzz and evaluated it on several real-world network services provided by ProFuzzBench [34]. The evaluation results showed that NSFuzz could infer a more accurate state model and achieve better fuzzing throughput than the state-of-the-art network fuzzers. As a result, the overall fuzzing effect of NSFuzz is better than other solutions on most targets, which includes higher code coverage and more crashes triggered in less time. In addition, NSFuzz found 8 zero-day vulnerabilities in the latest version of three popular network services, which shows its ability to find real-world vulnerabilities.

2 RELATED WORK

2.1 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 test cases for fuzzing. SPIKE [14] used a block-based analysis method to automatically generate valid data blocks of protocol messages with predefined generation rules. PROTOS [29] provided some template-based generation and fault injection primitives for users to specify particular fields in the protocol format to generate test cases.

In addition to generating test cases directly from templates, some works introduced state models to help improve fuzzing efficiency for stateful protocol services. SNOOZE [19] and KiF [13] proposed a scenario-based fuzzing method, which required users to prebuild the interaction scenarios by specifying the message order to achieve the fuzzing of stateful protocols services. Until today, the widely used black-box network fuzzing tools, such as Peach [41], Sulley [35], and boofuzz [37], are still built on top of these ideas. To reduce reliance on prior knowledge and manual work before fuzzing, AutoFuzz [27] proposed a more automated fuzzing framework by using network traffic analysis to extract the message format and protocol state model, and traversed the service state space by modifying the messages as a built-in proxy during fuzzing. AspFuzz [30] would directly change the sending order of generated message sequences to perform state-level fuzzing. PULSAR [25] 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.

2.2 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 algorithms to conduct fuzzing has attracted much more attention. Applying this method in the fuzzing of network services has also become a research hotspot. For example, Peach* [32] combined code coverage feedback with the original Peach [41] fuzzer to carry out testing on Industrial Control Systems (ICS) protocols. It used the AFL-like compile-time instrumentation mechanism to collect the ICS protocol services' code coverage during fuzzing, and used the crack ability of Peach to combine and generate more efficient test cases.

Although coverage feedback is a widely employed scheme in grey-box fuzzing, it is not sufficient enough for stateful network protocol fuzzing because some code is likely to be triggered only after a specific state is triggered. Recently, researchers have proposed some methods to use state feedback for network protocol program fuzzing. IoTHunter [45] first applied grey-box fuzzing for stateful network services in Internet of Things (IoT) devices. It not only used code coverage to guide fuzzing, but also implemented a multi-stage testing method for different states based on the state feedback of the protocol. yFuzz [20] performed a similar idea to IoTHunter, while it mainly analyzed the implementation of AFL and proposed a multi-forkserver structure to achieve multi-stage fuzzing of the stateful protocol services. AFLNET [39] is the first grey-box fuzzer for protocol implementations, employing state feedback to guide the fuzzing process. It acts as a client, replaying variations of the original sequence of messages sent to the server and retaining those variations that were effective at increasing the coverage of the code or state space. STATEAFL [33] is the variation of AFLNET, which used an in-memory state to represent the service state. It instruments the target server at compile-time and infers the current protocol state of the target during runtime. It incrementally builds a protocol state machine for guiding fuzzing. SGFuzz [18] proposed to analyze the source code to identify state variables offline and then gradually constructed the STT during runtime, which is then used as a guide for the stateful grey-box fuzzing.

As is proved by AFLNET, STATEAFL, SGFuzz, and more, using state coverage to guide grey-box fuzzing showed its power in testing network protocol programs. However, as discussed in Section 1, how to represent the *state* is an important research question. There are several existing schemes for state representation. AFLNET [39], SGPFuzz [46], and others [28, 44] used response code to represent the service state. The response code was used to perform run-time state model building and state-guided fuzzing. Using response code to represent the protocol state is an active attempt and this solution is relatively more state aware. However, the response code is not accurate enough to represent the state of the protocol program. STATEAFL [33] tried to overcome the limitation of the response code–based state representation scheme by using an in-memory state to represent the service state. This method could infer a more reasonable state model, but its vast post-execution analysis overhead further led to a low fuzzing efficiency. SGFuzz employed state variables to build an STT during fuzzing that was further used to guide the fuzzing process. It proposed to automatically identify so-called state variables and track the sequence of values assigned to them during fuzzing to produce the STT that could represent the state space.

As discussed in Section 1, another challenge faced by network protocol fuzzing is testing efficiency. yFuzz proposed a multi-FORKSERVER mechanism to achieve multi-stage fuzzing of target protocol programs. Similarly, Nyx-Net [40] used a snapshot-based method to quickly restore the service state and selectively emulate network functionality to avoid the high cost of handling the full network traffic. SnapFuzz [15] proposed a new solution to overcome the influence of complex fuzzing harnesses involving custom time delays and clean-up scripts. It intercepts all network communication interactions through binary rewriting and achieves fast I/O synchronization. In addition, it introduces a series of optimization measures such as the in-memory file system and smart deferred-forkserver to boost up the throughput of network service fuzzing. In this article, we are also introducing a fast I/O synchronization mechanism (see Section 4.5.1) to improve the fuzzing efficiency of network services. The essential idea is the same as SnapFuzz's, which is to eliminate unnecessary waiting time. However, we achieved it in two different ways. NSFuzz used static analysis and compile-time instrumentation to make the target server send synchronization signals to the fuzzer, whereas SnapFuzz used binary rewriting to keep track of the target's action and realized a serials of optimizations.

2.3 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 [21] tried to use dynamic taint analysis to infer the protocol state model and message format, while PRETT [31] used binary tokens combined with network traces to build minimized state models. Besides, IJON [17] and FuzzFactory [36] allowed users to annotate the specific variables in the program under test via provided application programming interfaces (APIs), using specific feedback to guide the fuzzer to perform domain-specific fuzzing. In addition to AFLNET, STATEAFL, the authors of [44, 46] 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 (Datagram Transport Layer Security) protocol state model and verified its security manually to check whether the state model in the implementation of services has logic flaws. De Ruiter and Poll [22] proposed Protocol State Fuzzing, which combines fuzzing with Angluin's L* algorithm [16] to automatically learn and infer the state machine implemented in the TLS protocol client/server program. Fiterau-Brostean et al. [24] applied a similar method to the protocol state fuzzing work for DTLS services based on TLS-Attacker [7] to perform protocol implementation security analysis under more configurations.

3 STUDY ON NETWORK SERVICE

3.1 Implementation of Network Service

We carry out extensive research on network services programs and find that 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. In addition, services would use network programming interfaces (such as *socket()* in Glibc [10]) 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 process requests from the client and give responses in an event loop. Once the client requests come, 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. The most typical implementation of the event loop is a single loop structure. It could also be constructed with a multi-level nesting loop or event-driven architecture.
- *Service Cleanup Stage.* When the network services are going to be actively terminated, this stage is responsible for program cleanup, such as 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. It should also be noted that developers always use some variables to represent the service state in their implementation of stateful network protocols.

3.2 Insight

Listing 1 shows a *Service Processing Stage* code snippet in a real-world network service implementation Bftpd [43]. Line 5 to Line 8 contain a network event loop implemented by a single loop structure, where Bftpd receives the request message at Line 5 and parses it at Line 7. Bftpd FTP protocol service would repeat these operations until the remote client user actively closes the

```
 1 // bftpd/main.c
 2 int state = STATE_CONNECTED;
 3 int main() {
 4   ... // service initialization stage
 5   while (fgets(str, MAXCMD, stdin)) {
 6     ... // trim "\r\n" from str
 7     parsecmd(str);
 8   }
 9   ... // service cleanup stage
10 }
11
12 int parsecmd(char *str) {
13   ... // remove garbage in the string
14   for (i = 0; commands[i].name; i++) {
15     // str matches the commands[i]
16     if (!strncasecmp(str, commands[i].name, strlen(commands[i].name))){
17       ... // split request type and parameters
18       // state check
19       if (state >= commands[i].state_needed) {
20         // invoke the corresponding handler
21         commands[i].function(str);
22         return 0;
23       } else {
24         switch (state) {
25           case STATE_CONNECTED:
26             response("503 USER expected");
27             return 1;
28           case STATE_USER:
29             response("503 PASS expected");
30             return 1;
31           case STATE_AUTHENTICATED:
32             response("503 RNFR before RNTO expected");
33             return 1;
34         }
35       }
36     }
37   }
38 }
39
40 // handler for "PASS" request message
41 void command_pass(char *password) {
42   if (state > STATE_USER) {
43     response("503 Already logged in");
44     return;
45   }
46   if (bftpd_login(password)) {
47     state = STATE_CONNECTED;
48     return;
49   }
50 }
```

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

network socket or the service ends 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 (Line 16), and performs a state check (Line 19), then invokes the corresponding handler (Line 21) only after passing the state check. Otherwise, it responds with a failure message to the client based on the current service state. During the message handler, Bftpd would conduct specific business processing and update the state of network services.

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 that 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 (Line 5) 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 an I/O synchronization point to actively notify the fuzzer to send the following request message, thereby avoiding useless time waiting. This synchronization can also tell the fuzzer to perform state collection to avoid post-execution analysis of state transition.

As mentioned earlier, network services always use variables to represent the runtime 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 (Line 21) and executes different code branches according to the current service state (from Line 42 to Line 49). Bftpd would update the service 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 24) or gets updated (e.g., Line 47) within the network event loop. In addition, compared with the response code, using the program variable to represent the service state is 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 Line 25 to Line 33), which shows that the response code is not robust.

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

4 METHODOLOGY

4.1 Overview Design

Figure 1 depicts the workflow of NSFuzz. At a high level, NSFuzz has four main components: static analysis, annotation API, compile-time instrumentation, and fuzzing loop. First, NSFuzz takes the network service source code as input and performs static analysis to identify the potential network event loop (see Section 4.2.1) and state variables (see Section 4.2.2). Next, NSFuzz introduces annotation APIs, which allow users to select from the potential results of static analysis to accurately annotate the I/O synchronization points (see Section 4.3.1) and state variables (see Section 4.3.2). For targets that cannot be adapted by static analysis, users can also analyze the source code of target services manually and annotate the synchronization and state information of the SUT directly. Then, NSFuzz uses the parsing result of annotated information (see Section 4.3.3) to conduct compile-time instrumentation (see Section 4.4), enabling the target service to have the capabilities of signal feedback and 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 model to generate test cases. During test case execution, when a request message is sent, NSFuzz would wait for the feedback signal from the SUT to perform fast I/O synchronization (see Section 4.5.1) to achieve efficient service fuzzing. Each time NSFuzz receives the signal, it would trace the variable-based service state and collect the state transition sequence to update the state model (see Section 4.5.2), thus performing state-aware fuzzing (see Section 4.5.3). NSFuzz would repeat this process to find program crashes until the fuzzing loop stops.

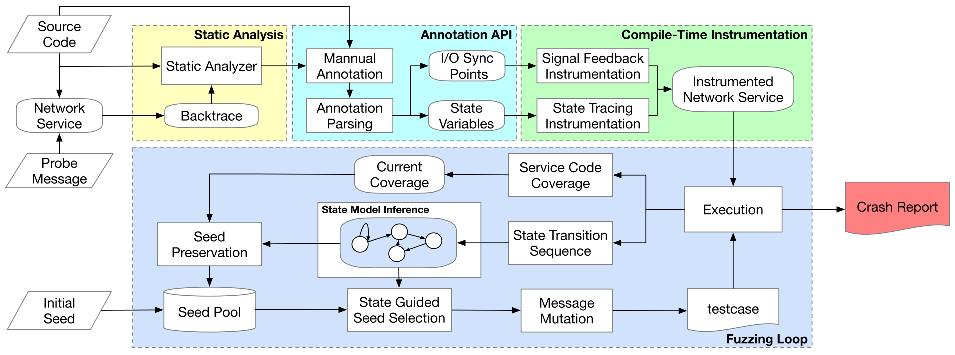

Fig. 1. Workflow of NSFuzz. It first uses static analysis and annotation API on the source code to identify I/O synchronization points and state variables. Then, it instruments the target network service to enable signal feedback and state tracing. Finally, NSFuzz performs fast I/O synchronization and infers the variable-based service state model to achieve an efficient state-aware fuzzing loop.

4.2 Static Analysis

NSFuzz uses static analysis to find two types of information in the SUT. The first is the network event loop, which is responsible for handling incoming request messages in the service processing stage. The second is the crucial state variables representing the service state in the program implementation. As mentioned in Section 3.2, 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. In addition, instead of StateAFL, which needs post-execution online analysis to identify state variables, NSFuzz extracts state variables by using pre-execution offline analysis and instruments the SUT during compilation to achieve real-time state mapping, thereby further improving fuzzing efficiency.

4.2.1 Loop Structure Identification. The most common implementation of the event loop is to use a single loop structure for continuous message processing, and the entry of the loop structure could be used as the I/O synchronization point. The main challenge to identifying such a loop structure is distinguishing it from other loops in the network service program because there may be various loops in the implementation. A typical example is that many services may use a file I/O loop to read configurations during the service initialization stage. The event loop itself may also contain nested loops, which makes it difficult for the static analyzer to identify the target 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.

First, we set breakpoints on input-related system calls such as read, recv, recvmsg, and so on, 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 target loop structure. The static analyzer first records all loops containing 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 the target loop. This is because the backtrace only contains the call stack within the event loop during the service processing stage, and matching the outer function that contains an I/O loop could also avoid nested loops. NSFuzz would record the position of the identified target loop entry for subsequent usage.

```c
// bftpd/main.c
int main() {
   ... // service initialization stage
  // the network event loop implemented by a signal loop
  while (fgets(str, MAXCMD, sock)) {
    _NSFUZZ_SYNC(); // using the synchronization point annotation API
    ...
    parsecmd(str);
  }
  ...
  return 0;
}
```

Listing 2. Example of the I/O synchronization point annotation API usage.

4.2.2 State Variable Extraction. After identifying the loop structure of the SUT, the static analyzer will extract the state variables. Since not all variables in the service program can be used to represent the service state, the static analysis needs to focus on the state variables in the target services. Therefore, based on our observation of the network service implementation and our analysis of the characteristics of state variables in the service programs, we use the following heuristic rules to filter the identification results of the static analyzer to extract the more likely state variables.

- The state variables–related operations in network services are always executed in the network event loop. Hence, the static analyzer performs analysis only 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 loop or message handlers. Hence, the static analyzer extracts only 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, which are assigned constants to represent a specific state only. Hence, the static analyzer keeps only global integer variables or user-defined structure members assigned constant values in their store operation.

Based on these heuristic rules, the static analyzer could extract potential state variables within the SUT.

4.3 Annotation API

In addition to automatic static analysis, NSFuzz also provides annotation APIs for users to manually annotate I/O synchronization points and state variables in the SUT. For services that implement the event loop by multi-level loops or event-driven libraries that are beyond the ability of the static analyzer, users could use the annotation API to directly annotate the I/O synchronization points of the SUT accurately, thereby adapting more network service targets to the NSFuzz solution for efficient fuzzing. On the other hand, when extracting the state variables within the SUT, the static analyzer may still encounter false positives due to the lack of service runtime information. Hence, this annotation API could be used to refine the automatic analysis result on state variables to help build a more accurate state model. In addition, users could only annotate the state variables they want to trace during fuzzing to achieve different granularity of state model inference.

4.3.1 Synchronization Point Annotation. The synchronization point annotation API provided by NSFuzz enables users to locate and annotate the I/O synchronization points in the source code of the SUT. Listing 2 shows an example of using the API **_NSFUZZ_SYNC** to annotate I/O synchronization points in the real network service program.

In the example shown in Listing 2, the I/O synchronization point annotation API is placed at the entry of the event loop. Each time the service accesses this point, it would send a signal feedback to indicate to the fuzzer that the service has processed the previous message. In addition, for the service programs that have multi-level or event driven–type event loops, users could also use this API to annotate multiple appropriate locations in the source code. Specifically, when there are multiple synchronization points, users could use this API to specify the synchronization points to make sure that only one synchronization signal is sent for each message. For subjects with no loop detected, users could also use this API to specify the synchronization points. In such cases, the state variables are also annotated using the annotation API (Section 4.3.2). In this way, the SUT could ensure that the synchronization signal feedback could always be sent after each request message is processed, thereby performing fast I/O synchronization to achieve efficient fuzzing.

4.3.2 State Variable Annotation. The state variable annotation API provided by NSFuzz enables users to manually annotate the crucial state variables that are used to represent the service state in the SUT. Listing 3 shows the example of using the **_NSFUZZ_STATE** API to annotate two types of state variables in real network service programs.

```
// bftpd/commands.h
enum { // Declaration of enumeration type
    STATE_CONNECTED, STATE_USER, STATE_AUTHENTICATED, STATE_RENAME,
     STATE_ADMIN
};
// bftpd/commands.c
int _NSFUZZ_STATE(state) = STATE_CONNECTED; // Annotating global state
     variables via API

// tinydtls/dtls-state.h
typedef enum { // Declaration of enumeration type
  DTLS_STATE_INIT = 0, DTLS_STATE_WAIT_CLIENTHELLO,
  ...,
  DTLS_STATE_CLOSING, DTLS_STATE_CLOSED
} dtls_state_t;
// tinydtls/dtls-peer.h
typedef struct dtls_peer_t {
  struct dtls_peer_t *next;
  ...
  dtls_state_t _NSFUZZ_STATE(state); // Annotating structure member state
     variables via API
} dtls_peer_t;
```

Listing 3. Example of the state variable annotation API usage.

In network service programs, the state variable annotation API can be used to annotate state variables at both the definition of global variables and the declaration of structure member variables (Listing 3). This API-based manual annotation could allow the users to avoid the redundant variables brought by the false positives from the static analysis and construct a state model with appropriate granularity in the fuzzing process.

4.3.3 Annotation Parsing. After completing the manual annotation, NSFuzz would parse these annotations to generate the output for instrumentation. For all I/O synchronization points marked by **_NSFUZZ_SYNC** in the code, the annotation parsing engine would output the location of I/O synchronization points in the source code. As for all variables marked by **_NSFUZZ_STATE**, the annotation parsing engine would assign a unique string ID for each variable and outputs the ID list.

4.4 Compile-Time Instrumentation

After obtaining the I/O synchronization points and list of state variables through static analysis and annotation API, 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 I/O synchronization points. Hence, the SUT 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 service state to the fuzzer engine in real time, NSFuzz instruments the *STORE* operation of each state variable. Each time any state variable is written, the service would use the new value to update its corresponding item in 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 state mapping method is as follows:

$$shared\_state[hash(var\_id)] = cur\_store\_val. \tag{1}$$

NSFuzz hashes the unique string ID (var_id) of each state variable and uses the hash result as an index. Then, it updates the *shared_state* with the new stored value of each state variable during fuzzing in the corresponding region. After the compilation, NSFuzz has generated the instrumented program as the final SUT for real fuzzing.

4.5 Fuzzing Loop

After using static analysis and annotation API to obtain the two types of information and instrument the target service, NSFuzz then starts to fuzz the instrumented SUT. Compared with the traditional coverage-guided grey-box fuzzers, NSFuzz introduces signal feedback to control service I/O interaction; thus, the fuzzer also needs to cooperate with the instrumented service to achieve fast I/O synchronization during the test case 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.

4.5.1 Fast I/O Synchronization. To improve fuzzing efficiency, AFL introduced FORKSERVER to be responsible for the fork creation and recycling of the program under test. As both AFLNET and STATEAFL are implemented based on AFL, they both inherit the FORKSERVER from AFL to conduct network service fuzzing. When they execute a test case, 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 the SUT 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 SUT in the NSFuzz framework.

4.5.2 Service State Tracing. 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, the service will update the values of certain state variables. Then, the *shared_state* buffer will also be updated promptly. As a result, after this message is processed,

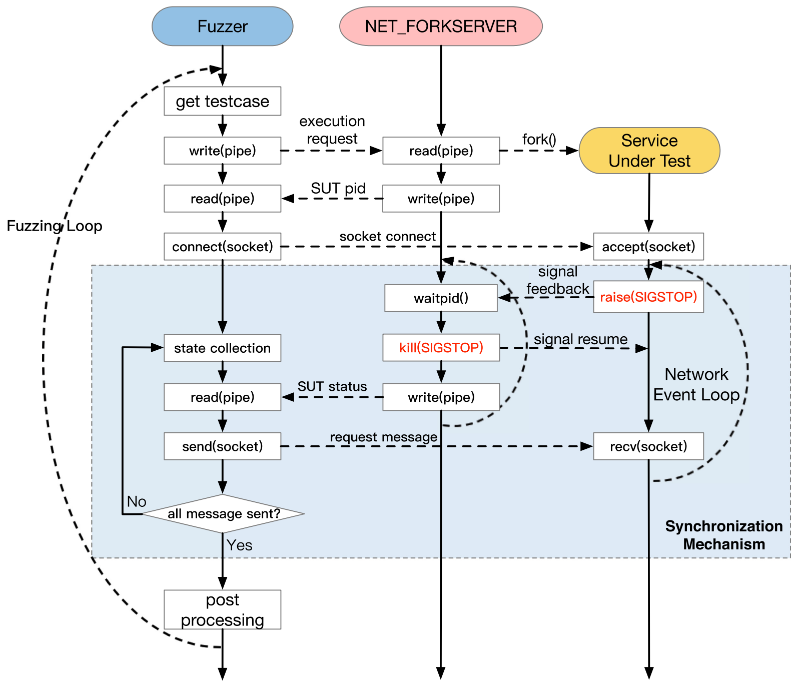

Fig. 2. The I/O synchronization mechanism among the fuzzer, NET_FORKSERVER, and service under test.

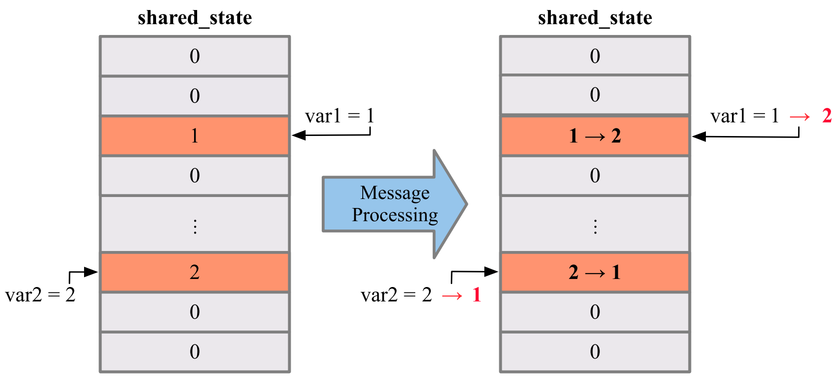

Fig. 3. The update process of shared_state when a request message leads to multiple state variables changed, indicating a service state transition.

the fuzzer will get a hash of the *shared_state* buffer, which is different from the one before this message. Although there may be multiple state variables in the network service program, different variables correspond to different indexes calculated by their unique ID hash. Hence, the change of each state variable does not interfere with the others. The combination of the current values of all state variables represents the comprehensive state of the service. Therefore, whenever any of the state variables of the SUT changes, the hash value of the shared_state will also change. Figure 3 shows the update process of *shared_state* when the service has a state transition.

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 of 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 collection.

4.5.3 State-Aware Fuzzing. Apart from new code coverage, NSFuzz would also use the state feedback information to update the service state transition model. Then, NSFuzz would perform state-guided seed selection and message mutation similar to AFLNET, carrying out network service fuzzing.

5 EVALUATION

We have built a prototype of NSFuzz. The implementation of NSFuzz is about 4.5 k lines of C/C++ code and about 100 lines of Python script. In detail, we implement the static analyzer, annotation parsing engine, and compile-time instrumentation based on LLVM [11] framework, and the fuzzer engine is implemented based on AFLNET (revision 0f51f9e from January 2021). To elaborate on the evaluation of NSFuzz, we have performed several experiments to answer the following research questions:

- **RQ1, Fuzzing efficiency of NSFuzz:** Could NSFuzz bring higher fuzzing efficiency based on the efficient I/O synchronization mechanism during the fuzzing process?
- **RQ2, Accurateness of the inferred state model inferred by NSFuzz:** Could NSFuzz infer a relatively more accurate state model based on the variable-based state representation scheme during the fuzzing process?
- **RQ3, Overall effectiveness of NSFuzz efficient state-aware fuzzing:** Could NSFuzz achieve better overall fuzzing results than other existing approaches?
- **RQ4, State space exploration ability of NSFuzz:** Could NSFuzz achieve higher state space coverage than other approaches?
- **RQ5, Real-world bug finding ability of NSFuzz:** Could NSFuzz find bugs in real-world protocol services continually?

5.1 Experiment Setup

We selected fuzzing targets from the network protocol fuzzing benchmark ProFuzzBench [34] to evaluate NSFuzz. ProFuzzBench is a benchmark for stateful protocol fuzzing. It contains 13 network service implementations from 10 network protocols (including FTP, SMTP, SIP, etc.). 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. To make a more thorough evaluation, we chose all 13 network services in ProFuzzBench as the evaluation targets. Table 1 shows the information of the target services.

To make the comparison, we selected two state-of-the-art grey box network fuzzers, AFLNET[1] (reversion 0f51f9ed from January 2021) and STATEAFL[2] (reversion c1b2aee from October 2021), and another network-enabled version of AFL, AFLNWE[3] [38] (reversion 6ba3a25 from March 2021) 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 this model during the fuzzing loop. STATEAFL collects the changed variables during network I/O rounds. Then, it

[1]https://github.com/aflnet/aflnet.
[2]https://github.com/stateafl/stateafl.
[3]https://github.com/aflnet/aflnwe.

Table 1. The Basic Information of Target Services from ProFuzzBench Used for Evaluation

Target Service	Protocol Type	Version/Commit	Transport Layer	Language
LightFTP	FTP	5980ea1	TCP	C
BFTPD	FTP	v5.7	TCP	C
Pure-FTPD	FTP	c21b45f	TCP	C
ProFTPD	FTP	4017eff8	TCP	C
Dnsmasq	DNS	v2.73rc6	UDP	C
TinyDTLS	DTLS	06995d4	UDP	C
Exim	SMTP	38903fb	TCP	C
Kamailio	SIP	2648eb3	UDP	C
OpenSSH	SSH	7cfea58	TCP	C
OpenSSL	TLS	437435	TCP	C
Forked-daapd	DAAP	2ca10d9	TCP	C
Live555	RTSP	ceeb4f4	TCP	C++
Dcmtk	DICOM	7f8564c	TCP	C++

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. To evaluate the effects of each component of NSFuzz during the fuzzing process, we also proposed a fuzzer NSFuzz-V, which enabled only the variable-based state representation scheme without the I/O synchronization mechanism. Then, we performed an ablation study on the overall effectiveness evaluation (RQ3) and state space coverage evaluation (RQ4) of fuzzing experiments.

All experiments were running on the same testing machine during this evaluation. This testing machine contains 128 Intel(R) Xeon(R) Platinum 8358 CPUs and 384 GB of memory with SSD disk. We set up each target service with each fuzzer in separate docker containers and used the same computing resource for experimental evaluation. We fuzzed each target service with different fuzzers for 24 hours and repeated 4 times each for a total of 6240 CPU hours of fuzzing evaluation.

5.2 Fuzzing Efficiency Evaluation (RQ1)

5.2.1 Static Analysis and Annotation. In order to achieve efficient I/O interaction during the network service fuzzing, NSFuzz first used static analysis and annotation API to identify the I/O synchronization points in the target services. Table 2 shows the static analysis and annotation results of NSFuzz for 13 target services. As can be seen from the table, the static analyzer could identify the network event loop in 9 target services. Among them, some of the targets' event loops could be identified automatically, and the loop structure entry can be directly used as the I/O synchronization point. However, as mentioned earlier, the event loop in some targets is not implemented by a single loop structure but rather by using the multi-level or event-driven framework. In these cases, multiple I/O synchronization points may be required to achieve synchronization for various request messages. In addition, it may be difficult to identify the event loop in some C++ target services due to the indirect calls generated by virtual functions. Therefore, for the targets that cannot be fully adapted by static analysis to identify the event loop, we use the annotation API provided by NSFuzz to calibrate or add I/O synchronization points manually. It should be noted that the I/O synchronization point annotation does not need too much manual work, and it is a one-time effort for each target service. For people unfamiliar with these target services before, the average time required to annotate the I/O synchronization point within different targets is from several minutes to up to 2 hours.

Table 2. Synchronization Point Identification Results in Target Service via Static Analysis and API Annotation of NSFuzz

Target Serivce	**Protocol Type**	**Event Loop Identify**	**# of I/O Sync point**	
			# of Static Analysis	**# After Annotation**
LightFTP	FTP	√	1	1
Bftpd	FTP	√	1	1
Pure-FTPd	FTP	√	1	1
ProFTPD	FTP	√	1	1
Dnsmasq	DNS	√	1	1
TinyDTLS	DTLS	√	1	1
Exim	SMTP	√*	1	1
Kamailio	SIP	√*	1	1
OpenSSH	SSH	√*	1	2
OpenSSL	TLS	×	N/A	1
Forked-daapd	DAAP	×	N/A	3
Live555	RSTP	×	N/A	1
Dcmtk	DICOM	×	N/A	2

√ denotes the static analysis could correctly identify the network event loop automatically in these targets; √* means the static analysis could identify the event loop, but the annotation API still needs to be used; × indicates these targets can only be adapted into the NSFuzz solution via the I/O synchronization point annotation API instead of static analysis.

Table 3. Average Fuzzing Throughput of Various Fuzzers Among 4 Runs of 24 Hours Toward Each Target Service Compared with AFLNET

Target Service	**Fuzzing Throughput (exec/s)**			
	AFLNet	**AFLNwe**	**StateAFL**	**NSFuzz**
LightFTP	6.36	+463.55%	−49.88%	**+931.81%**
Bftpd	3.35	+34.45%	−76.23%	**+2,678.62%**
Pure-FTPd	4.65	+107.63%	−36.56%	**+805.32%**
ProFTPD	2.68	+203.46%	−70.37%	**+2,008.97%**
Dnsmasq	5.76	+333.68%	−84.85%	**+1,004.08%**
TinyDTLS	2.36	+423.70%	−63.39%	**+20,059.68%**
Exim	2.41	+116.91%	+8.40%	**+268.67%**
Kamailio	4.59	+23.99%	−58.11%	**+484.33%**
OpenSSH	22.64	+25.23%	−67.56%	**+228.06%**
OpenSSL	6.29	+1.07%	−29.36%	**+595.39%**
Forked-daapd	0.90	−1.40%	+0.56%	**+357.26%**
Live555	4.71	+480.79%	+8.55%	**+2,025.05%**
Dcmtk	15.50	+104.03%	−71.33%	**+94.60%**
Average	**6.32**	**+178.24%**	**−45.40%**	**+2,426.30%**

The AFLNET column displays the absolute number of the throughput; all other columns denote the changes compared to AFLNET.

5.2.2 Fuzzing Throughput. After identifying the I/O synchronization points and instrumenting the target services, NSFuzz performed efficient fuzzing on network services by fast I/O synchronization. Table 3 illustrates the average fuzzing throughput of NSFuzz and other fuzzers among 4 runs of experiments on different target services. The fuzzing throughput means the number of test cases executed per second. Obviously, a higher fuzzing throughput indicates that more test cases have been executed simultaneously, thus, the overall test efficiency is also higher.

Table 4. State Variable Extraction Results in Different Target Services via Static Analysis and Annotation API of NSFuzz

Target Service	Protocol Type	LoC	State Variable			
			Analysis Time	# of Analysis	# After Annotation	Example
LightFTP	FTP	4.4k	0.7s	1	1	Access
Bftpd	FTP	4.7k	1.8s	6	4	state
Pure-FTPd	FTP	30k	3.9s	22	3	loggedin
ProFTPD	FTP	227.7k	33.5s	81	5	logged_in
Dnsmasq	DNS	27.6k	9.5s	15	7	found
TinyDTLS	DTLS	10.8k	3.2s	4	2	state
Exim	SMTP	101.7k	39.8s	57	8	helo_seen
Kamailio	SIP	766.7k	441.9s	58	5	state
OpenSSH	SSH	97.3k	31.3s	69	7	istate
OpenSSL	TLS	441.8k	N/A	N/A	3	hand_state
Forked-daapd	DAAP	115.9k	N/A	N/A	4	state
Live555	RTSP	52.2k	N/A	N/A	3	fIsActive
Dcmtk	DICOM	575.2k	N/A	N/A	4	state

LoC means the line of code of target services. Analysis time and analysis number denote results from the static analysis process. N/A means NSFuzz cannot perform automatic static analysis in the target service.

As shown in Table 3, the fuzzing throughput of NSFuzz is significantly better than that of AFLNET and other fuzzers. NSFuzz has improved the throughput from 1.8× to more than 200× on different target services and brings an average improvement of 24×. As expected, the I/O synchronization scheme introduced by NSFuzz significantly improved the fuzzing efficiency. In addition, we can see from the result that STATEAFL has the lowest throughput among the four fuzzers. Since 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 may always cause a decline in its fuzzing throughput compared with AFLNET. It is also worth noting that AFLNWE, the only fuzzer without state-aware, also improves fuzzing throughput compared to AFLNET. This is mainly because AFLNWE is just a network I/O–enabled version of AFL, which only sends one-time data to the SUT, thus saving the waiting delay time and reducing state-related overhead among multiple network interactions. However, even in this case, its fuzzing throughput is still not as good as NSFuzz, which proves that the efficient synchronization mechanism of NSFuzz based on lightweight instrumentation could significantly improve the fuzzing efficiency.

5.3 State Model Inference Evaluation (RQ2)

5.3.1 Static Analysis and Annotation. As before, NSFuzz also needs to perform static analysis and use annotation API to extract the state variables in the target services first. Table 4 shows the static analysis and annotation results of NSFuzz for 13 target services.

As we can see, NSFuzz could extract state variables via automatic static analysis in 9 target services, which involves multiple network protocol types. In addition, the number of extracted state variables and analysis time are generally positively correlated with the scale of the target services (LoC [line of code]), 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 network service state. For example, one of the state variables extracted from Pure-FTPD [23] 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.

Table 5. The Average Number of Vertexes and Edges of State Models Inferred by Different Fuzzers

Target Service	**Service State Model**					
	AFLNet		**StateAFL**		**NSFuzz**	
	Vertexes	**Edges**	**Vertexes**	**Edges**	**Vertexes**	**Edges**
LightFTP	23	220	51	254	5	11
Bftpd	24	228	4	8	15	47
Pure-FTPd	29	307	3	4	6	17
ProFTPD	27	307	16	102	37	188
Dnsmasq	102	329	95	359	2	2
TinyDTLS	8	23	28	75	7	26
Exim	12	59	9	29	5	14
Kamailio	13	105	3	4	5	14
OpenSSH	42	84	48	164	19	48
OpenSSL	8	12	24	40	11	21
Forked-daapd	8	20	6	8	9	31
Live555	10	78	18	149	15	48
Dcmtk	3	2	11	10	7	14

For targets not using a single loop structure to implement the network event loop, the static analyzer may fail to extract the state variables because it needs to start the analysis process from the entry of the loop. Therefore, we use the state variable annotation API provided by NSFuzz to annotate the state variables within them directly. Although multiple heuristic rules have been performed, targets that static analyzer could work on may still contain false positives, such as some config flag and message type variables. Hence, we also use the annotation API to refine the output from the static analysis and annotate only those variables that could explicitly represent the service state. Similarly, this lightweight state variable annotation mechanism needs only a one-time effort for each target service. For people unfamiliar with these target services, the average time required to annotate the state variables under different targets manually is about 20 minutes, and the number of final annotated variables is usually around 10.

5.3.2 Inferred State Model. 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 SUT during the fuzzing process. Table 5 shows the average number of vertexes and edges of the state model inferred by these fuzzers for the target service in the 24-hour fuzzing experiment.

To investigate the accuracy of the inferred state model, we take LightFTP [4] as an example for a case study because LightFTP annotates only one state variable; thus, the semantic information is more clear. After 24 hours of fuzzing of LightFTP, NSFuzz inferred the same state model in all four runs with 5 vertexes and 11 edges, as shown in Figure 4. The annotated state variable *Access* 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 client permission. The state model inferred by NSFuzz contains all 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 inferred state model. However, the state model inferred

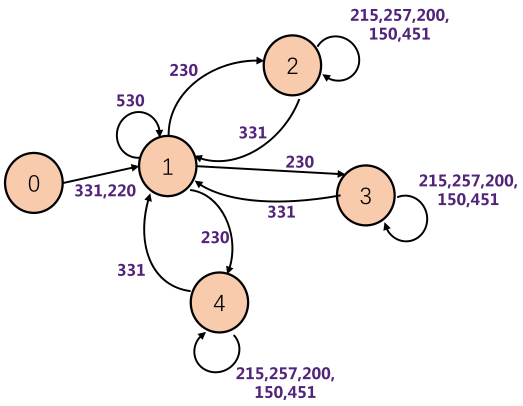

Fig. 4. The inferred state model of LightFTP by NSFuzz.

by AFLNet and StateAFL with the same initial seeds respectively had 23 vertexes/220 edges and 51 vertexes/254 edges on average at the end of fuzzing. According to our manual analysis, these state models inferred by AFLNet and StateAFL 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. This is because various network service implementations for the same protocol may define their state model with specific semantics. For example, variables used to mark the service mode may also be treated as a state variable broadly, which could bring an extension for the basic state model. Moreover, since the state variable extraction part in NSFuzz is a decoupled model from the fuzzing loop, users could also choose to annotate which state variables to monitor, indicating that NSFuzz has the ability to construct different granularities of the state model for network service.

5.4 Overall Effectiveness Evaluation (RQ3)

To evaluate the overall fuzzing effectiveness of NSFuzz, we have counted the average code coverage and crash trigger of NSFuzz and other fuzzers among 4 runs of experiments. Moreover, in order to explore the impact of each part (i.e., the variable-based state representation scheme and the I/O synchronization points-based speed-up scheme) of NSFuzz on the overall effectiveness of network service fuzzing, we also introduced a fuzzer, NSFuzz-V, which enabled only the variable-based state representation scheme, to conduct an ablation experiment.

5.4.1 Code Coverage. Code coverage is always a standard metric for evaluating fuzzers, which indicates how much code in the SUT has been executed during the whole fuzzing process. Usually, the higher the code coverage, the more program vulnerabilities may be triggered. Table 6 illustrates the average final code branch coverage of various fuzzers towards each target service during 4 instances of 24-hour fuzzing.

Results in Table 6 show that NSFuzz could achieve a higher code branch coverage than AFLNet on all 13 targets, which proved the effectiveness of our proposed methods in improving the code coverage. The results also show that on different targets, NSFuzz-V achieves better or slightly less code coverage than AFLNet and has an average improvement of 2.11%. The results of NSFuzz-V indicate that the modification of the state variables representation scheme alone is able to improve

Table 6. The Average Final Code Branch Coverage of Various Fuzzers among 4 Runs of 24 Hours Toward Each Target Service Compared with AFLNET

Target Service	**Final Code Branch Coverage**				
	AFLNet	**AFLNwe**	**StateAFL**	**NSFuzz-V**	**NSFuzz**
LightFTP	396.00	−76.52%	−3.66%	+1.89%	**+3.85%**
Bftpd	476.25	−1.42%	−3.52%	−1.63%	**+1.31%**
Pure-FTPd	1,112.25	+10.00%	−4.27%	+3.44%	**+17.71%**
ProFTPD	5,097.00	+3.35%	−4.26%	−0.28%	**+4.16%**
Dnsmasq	1,214.25	−3.89%	−11.16%	+0.54%	**+2.12%**
TinyDTLS	470.50	−26.89%	−12.70%	+11.26%	**+25.82%**
Exim	3,871.50	−72.78%	+5.85%	+8.26%	**+9.31%**
Kamailio	,704.00	−14.96%	−0.72%	+1.51%	**+9.44%**
OpenSSH	3,602.25	+0.59%	−4.20%	−1.76%	**+1.20%**
OpenSSL	9,654.75	−0.80%	−2.84%	−0.35%	**+1.49%**
Forked-daapd	2,478.25	−18.64%	−3.80%	−1.18%	**+10.10%**
Live555	2,998.00	+1.27%	−1.44%	+0.56%	**+0.81%**
Dcmtk	2,492.25	+7.99%	+2.81%	+5.21%	**+6.61%**
Average	**3,351.33**	**−14.82%**	**−3.38%**	**+2.11%**	**+7.23%**

The AFLNET column displays the absolute number of branch coverage; all other columns denote the changes compared to AFLNET.

the code exploration ability of the fuzzer. However, NSFuzz-V still has a decline of code coverage on 5 of the targets: Forkked-daapd, Bftpd, ProFTPD, OpenSSH, and OpenSSL. After analysis, the throughput of NSFuzz-V has a relatively large drop in these targets. This indicates that the overhead brought by the state variable instrumentation has a negative impact on code exploration. Fortunately, the negative impact can be made up for by the I/O synchronization mechanism of NSFuzz.

Although AFLNWE has a relatively high fuzzing throughput among these fuzzers (see Section 5.2.2), it could not achieve good code coverage on specific targets such as Exim, LightFTP, and TinyDTLS due to its lack of multiple network I/O interactions capability. In addition, the average number of code branches covered by STATEAFL during the 24-hour fuzzing is not significantly different from AFLNET, which indicates that although STATEAFL proposed a more reasonable state representation scheme, fuzzing speed is also a significant influence on the final code coverage.

Figure 5 shows the growth of the number of code branches explored with the fuzzing time during the 24-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.

However, there are two exceptions in which NSFuzz performs not as well as AFLNWE: Dcmtk and Live555. Dcmtk is used for image processing and storing, which has relatively less state transition than other targets. As for Live555, it supports the streaming data processing of protocol messages, which is not in line with the I/O model assumed in other fuzzers. Therefore, even the most straightforward one-time I/O stateless fuzzer AFLNWE could outperform other stateful fuzzers. NSFuzz-V performs better than AFLNET on most targets but is still not as good as AFLNWE on some targets. Except for Dcmtk and Live555 discussed earlier, the testing throughput is the main factor that prohibits the performance of NSFuzz-V. However, we can see from the figures of Dcmtk, ProFTPD, Pure-FTPd, OpenSSH, and so on that although NSFuzz-V may lose the advantage at the beginning, it has the ability to catch up later. This shows the positive effect of the state

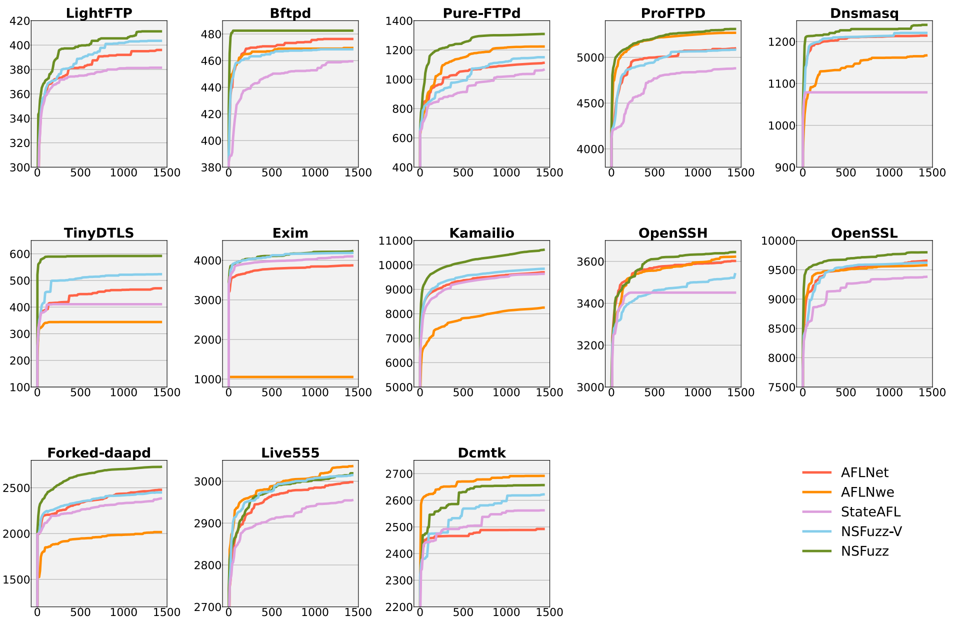

Fig. 5. The average code branch coverage growth of various fuzzers during 4 runs of 24 hours toward each target service. For all sub-figures, the x-axis represents the time(s), and the y-axis represents the code branch coverage. Note that the initial seeds used by target services have covered some code branches, and the y-axis mainly displays the new coverage triggered after the initial seeds are tested. Thus, it does not start from 0.

Table 7. Average Number of Triggered Crashes and Vulnerabilities in Target Services of Different Fuzzers During 4 Runs of Experiments

Target Service	**# of Crashes / # of Vulnerabilities**				
	AFLNet	**AFLNwe**	**StateAFL**	**NSFuzz-V**	**NSFuzz**
Dnsmasq	6 / 1	6 / 1	4 / 1	7 / 1	**7 / 1**
TinyDTLS	3 / 3	3 / 2	4 / 3	4 / 3	**5 / 4**
Live555	4 / 4	4 / 4	5 / 5	4 / 4	**5 / 4**
Dcmtk	1 / 1	2 / 2	1 / 1	1 / 1	**2 / 2**
Total	**14 / 9**	**15 / 9**	**14 / 10**	**16 / 9**	**19 / 11**

The crashes are clustered by the crash address reported by AddressSanitizer [42], and the vulnerabilities are manually analyzed.

representation scheme and explains why NSFuzz has a better performance after combining the speed-up solution.

5.4.2 Crash Trigger. In addition to the code coverage, the triggering of crashes directly reflects the vulnerability discovery capabilities of fuzzers. Table 7 shows the number of crashes and vulnerabilities of the target services triggered by various fuzzers during the 24-hour fuzzing experiment. We use the program crash address reported by AddressSanitizer [42] of the target service to cluster the number of crash triggers. Then, we determine the service vulnerabilities by manually analyzing the program crashes. It should be noted that the same vulnerability may cause the program to crash at different locations; thus, the number of triggered crashes would always be more than the actual number of vulnerabilities. As shown in this table, NSFuzz could always trigger more

Table 8. Average Time of Different Fuzzers to Trigger the First Crash and the Number of Runs with Crash Triggered During 4 Runs of Experiments

Target Service	**First Crash Trigger Time (s) / # of Runs with Crash Triggered**				
	AFLNet	**AFLNwe**	**StateAFL**	**NSFuzz-V**	**NSFuzz**
Dnsmasq	1,554.5 / 4	1,373 / 4	1,057.25 / 4	1,455.25 / 4	**48 / 4**
TinyDTLS	51.25 / 4	17 / 4	51 / 4	33.25 / 4	**0.25 / 4**
Live555	470 / 4	3008 / 4	845 / 4	363.5 / 4	**83.5 / 4**
Dcmtk	2,646.67 / 3	52,915 / 2	2,144 / 1	2,057.25 / 4	**4,628.75 / 4**

or equal crashes and vulnerabilities in these target services than other fuzzers. Compared with the other fuzzers, NSFuzz-V also shows improvement on some targets. Fortunately, all of these vulnerabilities have been fixed by the vendors in their latest version.

We also calculated the average time for various fuzzers to trigger the first crash and the number of runs that the trigger of the crash happens during 4 runs of experiments. The results are shown in Table 8. As we can see, except for Dcmtk, the average time to trigger the first crash of NSFuzz is significantly lower than other competitors. Especially during the fuzzing process of TinyDTLS [12], NSFuzz could always trigger the program crash in less than 1 s, which to some extent shows the high efficiency of NSFuzz to discover vulnerabilities. As for Dcmtk, the average time for the first crash of NSFuzz is larger than that of AFLNet and StateAFL. However, NSFuzz can stably trigger crashes in every run, which is not the case for AFLNet and StateAFL.

5.5 State Space Coverage Evaluation (RQ4)

To evaluate the state space exploration ability of NSFuzz, we check the state space coverage of different fuzzers within 24 hours of fuzzing. Since we propose to use state variables to represent the state of the SUT, the value range of the state variables constitutes the state space. Thus, we use the proportion of the explored values of state variables to represent the state space coverage.

By analyzing the source code of different SUTs manually, we found that the value range of some variables is hard to count. For example, some state variables are used as flags to represent n different kinds of information by setting each of the n bits to 0 or 1. Theoretically, the total number of possible values of such variables is 2^n. However, the actual value that can be reached during execution may be far less than the theoretical, which is determined by complex semantic information and difficult to estimate. Hence, we build an approximate state space using the union set of the state variable values that all fuzzers have triggered in all runs during the 24 hours of fuzzing, and the *state space coverage* refers to the ratio of the number of state variable values that have been triggered during the experiment to the total number of state variable values. For services that contain more than one state variable, we add up the number of state variables.

Figure 6 shows the average state space coverage of different fuzzers on different target services in the 24 hours of fuzzing among 4 runs. The average state space coverage of each fuzzer on different targets is shown in the label of the x-axis. As we can see from the result, NSFuzz could explore more state values than other fuzzers in all cases. NSFuzz-V has a slightly lower state space coverage than NSFuzz on average, but the performance is still better than the other three fuzzers. AFLNwe has the lowest average state space coverage, which is in line with the intuition since AFLNwe is the only non-stateful fuzzer. For more stateful targets, including Forked-daapd, TinyDTLS, ProFTPD, and OpenSSL, the improvement of NSFuzz and NSFuzz-V on state space coverage is more significant. However, for other targets, such as Dnsmasq, Bftpd, and so on, all fuzzers achieve a 100% coverage; thus, the improvement is not obvious. One reason is that the theoretical maximal

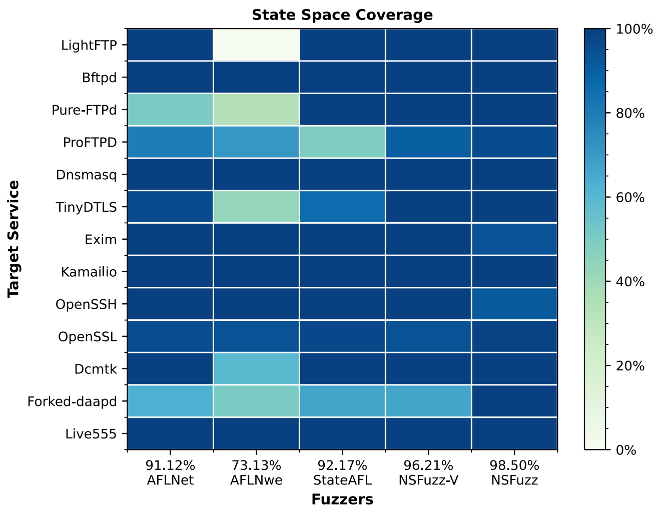

Fig. 6. The average state space coverage of various fuzzers among 4 runs of fuzzing during 24 hours. The values displayed above the fuzzer names in the labels of the x-axis are the average state space coverage of each fuzzer among the 13 targets.

Table 9. Information of Zero-Day Vulnerabilities Found by NSFuzz

Target Service	Commit	File	Vulnerablility Type	Status
TinyDTLS	fce3372	tinydtls/session.c	heap-use-after-free	Fixed
		tinydtls/dtls.c	heap-buffer-overflow	Fixed
	7068882	tinydtls/dtls.c	heap-use-after-free	Confirmed
	0bd943f	tinydtls/crypto.c	global-buffer-overflow	Reported
		tests/dtls-server.c	SEGV	Reported
Dcmtk	c749632	dcmnet/libsrc/assoc.cc	heap-use-after-free	Reported
		ofstd/libsrc/ofstd.cc	SEGV	Reported
		ofstd/libsrc/ofstring.cc	heap-buffer-overflow	Reported

number of different values of some state variables with enum type is easy to reach so that all fuzzers could discover the whole space easily.

5.6 Real-World Bugs Finding Evaluation (RQ5)

To evaluate NSFuzz's ability to discover vulnerabilities in real-world network protocol services, we deploy a long-term fuzzing campaign to find bugs in the latest version of the 13 protocols. We ran the fuzzing campaign for about 2 weeks and found several crashes in 2 of the target services. Details are provided in Table 9.

As shown in Table 9, NSFuzz found 8 vulnerabilities in total, 5 of which are found in 3 different versions of TinyDTLS and 3 are found in Dcmtk.

start here We reported all 8 vulnerabilities to the developers. The 2 crashes of TinyDTLS (commit fce3372) have been confirmed and fixed. One other crash of TinyDTLS (commit 7068882) has also been confirmed but will not be patched directly since the developers refactored this part of the code. The other 5 crashes were reported shortly before the submission of this article and are still waiting for confirmation. The results show that NSFuzz has the ability to continuously discover zero-day vulnerabilities of real-world network services.

6 DISCUSSION

6.1 State Space Exploration

To fully explore the state space of the target during fuzzing, we may need not only a more reasonable state representation but also better utilization of state feedback to guide the fuzzer. Currently, we use state variables to represent the state information of the SUT, which we thought was more in line with the essential meaning of the service state. However, we have not yet delved into how to use the state feedback information to guide the fuzzer better to mutate test cases. The current mutation procedure follows the method of AFLNET. For each round of fuzzing, AFLNET selects a seed that can lead the service to an interesting state. Then, it mutates the messages after reaching the target state to try to trigger more undiscovered states and edges. At present, NSFuzz still faces an insufficient guidance mechanism to fully explore the state space of the SUT, which is an important direction in future work.

6.2 SnapFuzz

SnapFuzz [15] is one of the most advanced network protocol fuzzing tools available now. The problem to be solved by Snapfuzz intersects with NSFuzz, that is, accelerating the network protocol service fuzzing. The essential idea of SnapFuzz and NSFuzz is the same, which is to eliminate unnecessary waiting time. NSFuzz uses a combination of static analysis and manual annotation to send the fuzzer the synchronization signal of the SUT to significantly reduce the waste of time. SnapFuzz, on the other hand, is based on a set of binary rewriting solutions for acceleration. This robust binary rewriting subsystem dynamically intercepts system calls and redirects them to custom handler functions. Based on binary rewriting, SnapFuzz implements several schemes for speeding up fuzzing.

Some of SnapFuzz's solutions are incompatible with NSFuzz. For instance, NSFuzz uses the sync point technique to send messages to the fuzzing tools, whereas SnapFuzz proposes a custom SnapFuzz protocol to keep track of the following action of the target and notifies the fuzzer about it. The SnapFuzz protocol includes a set of rules that define the flow of interaction between the fuzzer and the program under test to avoid the need to set time delays manually. SnapFuzz also uses the in-memory file system to avoid writing cleanup scripts and proposes a way to automate the identification of possible fork points. However, these techniques still cannot be directly applied to the design of NSFuzz as they do not use the custom SnapFuzz protocol.

6.3 Future Work

As discussed in Section 6.1, we have not delved into how to use the state feedback information better to guide the seed selection and mutation process. In the future, we will explore how to use state space coverage to guide the fuzzer to perform fuzz testing more efficiently on stateful targets. Also, since message exchange during protocol communication is highly formatted, we will investigate how to implement more intelligent format-aware message mutation to improve fuzzing efficiency.

7 CONCLUSION

In this article, we analyzed the problems of testing efficiency and state representation of existing network service fuzzers. According to our study on the implementation of network services, we proposed a high-efficiency state-aware fuzzing framework combined with a variable-based state representation scheme and I/O synchronization mechanism. Then, we implemented the prototype of NSFuzz by using static analysis, an annotation API, and lightweight compile-time instrumentation to generate the signal feedback and state tracing enabled SUT, thereby achieving state-aware fuzzing. Finally, we evaluated NSFuzz on all 13 targets of ProFuzzBench. The results showed that NSFuzz could achieve higher fuzzing throughput while inferring a relatively more accurate state model to guide the fuzzing. In addition, NSFuzz could reach higher code coverage and trigger more crashes in less time than other network fuzzers during the fuzzing process. Moreover, NSFuzz could continually find bugs in real-world network protocol services and has found 8 zero-day vulnerabilities.

ACKNOWLEDGMENTS

We would like to thank all anonymous reviewers for their insightful comments and feedback.

REFERENCES

[1] 2018. The Transport Layer Security (TLS) Protocol Version 1.3. Retrieved April 10, 2023 from https://datatracker.ietf.org/doc/html/rfc8446.
[2] 2020. The Heartbleed Bug. Retrieved April 10, 2023 from https://heartbleed.com/.
[3] 2021. libFuzzer —a Library for Coverage-guided Fuzz Testing. Retrieved April 10, 2023 from https://llvm.org/docs/LibFuzzer.html.
[4] 2021. LightFTP: Small x86-32/x64 FTP Server. Retrieved April 10, 2023 from https://github.com/hfiref0x/LightFTP.
[5] 2021. Readelf (GNU Binary Utilities). Retrieved April 10, 2023 from https://sourceware.org/binutils/docs/binutils/readelf.html.
[6] 2021. Server Message Block (SMB) Protocol Versions 2 and 3. Retrieved April 10, 2023 from https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-smb/.
[7] 2021. TLS-Attacker. Retrieved April 10, 2023 from https://github.com/tls-attacker/TLS-Attacker.
[8] 2021. WannaCry Ransomware Attack. Retrieved April 10, 2023 from https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.
[9] 2022. OpenSSL. Retrieved April 10, 2023 from https://www.openssl.org/.
[10] 2022. The GNU C Library (glibc). Retrieved April 10, 2023 from https://www.gnu.org/software/libc/.
[11] 2022. The LLVM Compiler Infrastructure. Retrieved April 10, 2023 from https://llvm.org/.
[12] 2022. TinyDTLS. Retrieved April 10, 2023 from https://projects.eclipse.org/projects/iot.tinydtls.
[13] Humberto J Abdelnur, Radu State, and Olivier Festor. 2007. KiF: A stateful SIP fuzzer. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications*. 47–56.
[14] Dave Aitel. 2002. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February* 105 (2002), 106.
[15] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, Vol. 12. 340–351. https://doi.org/10.1145/3533767.3534376
[16] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106.
[17] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP'20)*. IEEE, 1597–1612.
[18] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. (2022). arXiv:2204.02545 http://arxiv.org/abs/2204.02545.
[19] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *International Conference on Information Security*. Springer, 343–358.
[20] Yurong Chen, Tian Lan, and Guru Venkataramani. 2019. Exploring effective fuzzing strategies to analyze communication protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. 17–23.
[21] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol specification extraction. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 110–125.
[22] Joeri De Ruiter and Erik Poll. 2015. Protocol state fuzzing of {TLS} implementations. In *24th {USENIX} Security Symposium ({USENIX} Security'15)*. 193–206.
[23] Frank Denis. 2021. Pure-FTPd. Retrieved April 10, 2023 from https://www.pureftpd.org/project/pure-ftpd/.
[24] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of {DTLS} implementations using protocol state fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security'20)*. 2523–2540.

[25] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347.

[26] Google. 2021. Honggfuzz. Retrieved April 10, 2023 from https://github.com/google/honggfuzz.

[27] Serge Gorbunov and Arnold Rosenbloom. 2010. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS* 10, 8 (2010), 239.

[28] Huihui He and Yongjun Wang. 2020. PNFUZZ: A stateful network protocol fuzzing approach based on packet clustering. *Computer Science & Information Technology (CS & IT'20)*.

[29] Rauli Kaksonen, Marko Laakso, and Ari Takanen. 2001. Software security assessment through specification mutations and fault injection. In *Communications and Multimedia Security Issues of the New Century*. Springer, 173–183.

[30] Takahisa Kitagawa, Miyuki Hanaoka, and Kenji Kono. 2010. AspFuzz: A state-aware protocol fuzzer based on application-layer protocols. In *IEEE Symposium on Computers and Communications*. IEEE, 202–208.

[31] Choongin Lee, Jeonghan Bae, and Heejo Lee. 2018. PRETT: Protocol reverse engineering using binary tokens and network traces. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 141–155.

[32] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. 2020. ICS protocol fuzzing: Coverage guided packet crack and generation. In *57th ACM/IEEE Design Automation Conference (DAC'20)*. IEEE, 1–6.

[33] Roberto Natella. 2021. StateAFL: Greybox fuzzing for stateful network servers. *arXiv preprint arXiv:2110.06253* (2021).

[34] Roberto Natella and Van-Thuan Pham. 2021. ProFuzzBench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[35] OpenRCE. 2012. Sulley. Retrieved April 10, 2023 from https://github.com/OpenRCE/sulley.

[36] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[37] Joshua Pereyda. 2021. boofuzz: Network Protocol Fuzzing for Humans. Retrieved April 10, 2023 from https://github.com/jtpereyda/boofuzz.

[38] Van-Thuan Pham. 2021. AFLNWE. Retrieved April 10, 2023 from https://github.com/thuanpv/aflnwe.

[39] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A greybox fuzzer for network protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation: Testing Tools Track*.

[40] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2021. Nyx-Net: Network fuzzing with incremental snapshots. *arXiv preprint arXiv:2111.03013* (2021).

[41] Mozilla Security. 2021. Peach. Retrieved April 10, 2023 from https://github.com/MozillaSecurity/peach.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC'12)*. 309–318.

[43] Jesse Smith. 2021. Bftpd. Retrieved April 10, 2023 from http://bftpd.sourceforge.net/.

[44] Congxi Song, Bo Yu, Xu Zhou, and Qiang Yang. 2019. SPFuzz: A hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access* 7 (2019), 18490–18499.

[45] Bo Yu, Pengfei Wang, Tai Yue, and Yong Tang. 2019. Poster: Fuzzing IoT firmware via multi-stage message generation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2525–2527.

[46] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. 2020. SGPFuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access* 8 (2020), 198668–198678.

[47] Michal Zalewski. 2021. American Fuzzy Lop. Retrieved April 10, 2023 from https://github.com/google/AFL.

Received 16 June 2022; revised 1 October 2022; accepted 14 December 2022