A Framework of High-speed Network Protocol Fuzzing based on Shared Memory

Junsong Fu, Shuai Xiong, Na Wang, Ruiping Ren, Ang Zhou, and Bharat K. Bhargava, *Life Fellow, IEEE*

Abstract—In recent years, security test of network protocols based on fuzzing has been attracting more and more attentions. This is very challenging compared with the stateless software fuzzing and most early network protocol fuzzers are of low speed and poor test effect. Since the first greybox and stateful fuzzer named AFLNET was proposed, several new schemes have been designed to improve its performance from different aspects. During the research, a great challenge is how to greatly improve the fuzzing efficiency. Based on the basic analysis in SNPSFuzzer, this paper provides a more thorough analysis about the time consumption in a fuzzing iteration for 13 network protocols and then we design a High-speed Network Protocol Fuzzer named HNPFuzzer. In HNPFuzzer, the test cases and response messages between the client and server are transmitted through the shared memory, guided by a precise synchronizer, rather than the socket interfaces. This greatly shorten the period of an iteration. Moreover, we design a persistent mode attempting to fuzz the service instances in the memory more than one time based on analyzing the side effect information. This mode further improves the speed of fuzzing. Experiment results illustrate that our scheme can improve the fuzzing throughput by about 39.66 times in average and triggers a large number of crashes including 2 new vulnerabilities which cannot discovered by existing fuzzers. Note that, the existing network protocol fuzzing schemes proposed in different directions do not compete with each other and on the contrary, they can collaborate with each other to improve the overall fuzzing effect and efficiency. Consequently, more existing tools can be integrated into our framework to get better network protocol fuzzing effect.

Index Terms—Network protocol fuzzing, high-speed, shared memory.

1 Introduction

With the proliferation of various network services, it is imperative to thoroughly analyze the security of the network protocols. Considering that the protocols are of great number and complexity, it is unwise to inspect the protocol codes line by line. An alternative approach is testing protocol security in an automatic manner such as fuzzing. In recent years, most fuzzing schemes are proposed for stateless software and they have been achieving a great success in both academia [1]–[6] and industry [7]–[10]. For example, the Google's fuzzer named OSS-Fuzz has discovered over 30,000 bugs in 500 open source projects [11], [12] and meanwhile more and more open-source projects have been being targeted by this community [13].

Different to the prosperity of stateless software fuzzers, the fuzzing schemes particularly designed for stateful network protocol implementations are still in their early stage [14], [15]. Considering that the network protocol implementations are also software, we usually employ stateless software and stateful network protocol implementations to distinguish them from each other. For the sake of simplicity, we use software and network protocols to represent the stateless software and stateful network protocol implementations respectively in the rest of this paper. Recently, researchers begin to introduce the idea of software fuzzing into the network protocol security testing field [14], [16], [17]. However, the transfer from software fuzzing to network protocol fuzzing is not straightforward and five main challenges are summarized in the following:

- The network protocols are more complex involving a set of different participants. As a consequence, the framework to fuzz protocols is more complex and it is very different to software fuzzing frameworks.
- In network protocol fuzzing, the initial seeds are more difficult to generate. In general, the technical specification (TS) of a protocol is quite complicated and usually accurately defines the structure of the input messages. Randomly generated messages based on mutation as the manner in software fuzzing has a poor performance.
- The network protocols have a new concept compared with the common stateless software, i.e., the protocol state. The states of a protocol greatly improve the difficulty of finding the new program paths. Moreover, to reach a particular state, we need to walk through all the pre-states which makes it more inefficient.
- The flow path of information in the fuzzing process is stretched. To communicate with the server under test (SUT), the messages usually need to be implanted into sockets and then transmitted to the SUT. In the server side, the messages also need to be recovered from the sockets. Compared with software fuzzing, this is of low efficiency.
- Last but not least, the SUT needs to be restarted again and again in the fuzzing process. In the initial phase of restart, the SUT needs to read and process configuration files, initialize variables and prepare to receive the service requests. This is also of a great burden in terms of computing resource and time consumption.

Facing the above challenges, several network protocol fuzzers have been proposed. In the initial, the stateful black-

- *J. Fu, S. Xiong, R. Ren, and A. Zhou are with the School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China.*
  E-mail: {fujs, xiongshuai, rrp, r1mao}@bupt.edu.cn.
- *N. Wang is with the school of cyber science and technology, Beihang University, Beijing 100191, China.*
  E-mail: nawang@buaa.edu.cn.
- *Bharat K. Bhargava is with the Department of Computer Science, Purdue University, West Lafayette, IN 47906, USA.*
  E-mail: bbshail@purdue.edu.

box fuzzing approaches, such as Peach [18] and BooFuzz [19], are directly employed for network protocols and they generate test cases in a random or semi-random manner. A large number of messages are just discarded by the server and quite a fair amount of time is wasted in useless fuzzing process. As a consequence, both the coverage rates of codes and states of the network protocols are very low and it is severe to design more practical network protocol fuzzers.

AFLNET [14] is the first greybox and stateful fuzzer for network protocol implementations and it successfully causes a wave of additional research. To test the SUT, AFLNET assumes that the client side is totally under control and any carefully constructed messages can be sent to the server through the network. In general, the messages are implanted into sockets as the payload in transmission process and AFLNET needs to create new connections for each test case queue. Consequently, fuzzing the network protocols is much slower than fuzzing the software in which process the messages are directly fed to the software in memory.

For each protocol state, AFLNET construct a message corpus based on the captured network traffic. When generating new messages, AFLNET provides a mutation-based scheme. The input messages are divided into different sets based on the related protocol state. In fuzzing process, the messages in different sets comprise a sequence which are fed to the SUT in order. In this way, AFLNET can fuzz a complex protocol with a large number of states.

Another challenge is how to automatically construct the protocol state machine of a protocol. In AFLNET, the State Machine Learning (SML) module monitors the responses from SUT when feeding an input message M in a state annotated as S_i. If a new response is captured, SML constructs a new state S_j and insert it to the state machine. Considering that state S_j can be triggered by M directly from S_i, S_i and S_j need to be connected in the state machine. However, the process of state transfer between states is not considered.

It can be observed that AFLNET provides a grace framework to solve most challenges for network protocol fuzzing and however its performance can be further improved in several aspects. Though several network protocol fuzzing approaches are proposed as discussed in Section 2, how to decrease time consumption of a fuzzing iteration has not been thoroughly researched. In this paper, we provide a detailed analysis about the time consumption distribution in a fuzzing iteration and find that the fuzzing efficiency can be greatly improved in several aspects. Motivated by these interesting findings, we designed a high-speed network protocol fuzzing framework named HNPFuzzer.

First, transmitting the messages through the socket interface between the client under control (CUC) and the server under test (SUT) is very time-consuming. This is the greatest factor that decreases the network protocol fuzzing efficiency. In HNPFuzzer, the CUC and SUT do not communicate with each other through the socket interfaces. Instead, they transmit messages based on shared memory. To achieve this goal, we analyzed the client/server architecture and collected the library functions related to socket interfaces. Particularly, functions that can either write or read a file descriptor like $write()$ or $read()$ are also included because a socket is a file descriptor. Then these related functions were hooked via LD_PRELOAD mechanism and a connection controller was designed to control the data transmission between client and server based on shared memory.

Second, the time delay caused by synchronization process in protocol fuzzing cannot be ignored and this is totally different to software fuzzing. The system is prone to be asynchronous without meticulously designed synchronizer. Therefore, we carefully designed a synchronizer based on shared memory in HNPFuzzer. The synchronization processes between CUC and SUT, parent and child, a set of event loops are all discussed, respectively.

Third, the forked service instance in existing fuzzers are directly killed after a fuzzing iteration and the restart of the instance is time-wasting. If we can fuzz the instance more than one time, the fuzzing process can be more efficient. In HNPFuzzer, we design a mechanism to monitor the environments of the instance and checks its status after an iteration. If the instance is not contaminated, we execute another fuzzing iteration. In this way, the fuzzing efficiency is further improved.

We analyze the relations between our scheme with existing network protocol fuzzing schemes in different directions. An interesting observation is that the above approaches in different directions do not compete with each other and instead they can cooperate together to form a better network protocol fuzzing tool. Therefore, our scheme can be also treated as a framework.

This paper thoroughly analyzes how to improve the fuzzing speed based on shared memory and implements a harness for network protocol fuzzing. The main contributions of this paper are summarized as follows:

- We provide a thorough analysis about the time consumption distribution of different parts in a fuzzing iteration. Motivated by the analysis results, we designed a high-speed network protocol fuzzing framework named HNPFuzzer based on shared memory.
- A connection controller is proposed to exchange messages between the CUC and SUT with the help of two blocks of shared memory. Compared with the socket-based message sharing approach, the new scheme is much more efficient. Moreover, a persistent mode is implemented by the connection controller, which can skip the initialization phase in the fuzzing.
- We also design a sophisticated synchronizer based on the shared memory to synchronize the states of different components in the fuzzing process. This is extremely important for the stability of the whole fuzzing process.
- A series of experiments are conducted to evaluate the performance of HNPFuzzer. Simulation results illustrate that our scheme is of good performance in terms of fuzzing throughout, branch coverage and the capability of vulnerability discovery.

The rest of this paper is organized as follows. We summarize the related work of network protocol fuzzing in Section 2. The motivation of our scheme is thoroughly discussed in Section 3. Then, we present the overall system framework of HNPFuzzer in Section 4 and the core modules of the system are discussed in Section 5 and 6, respectively. We evaluate the performance of HNPFuzzer in Section 7. In Section 8, we provide a discussion to illustrate some important properties of HNPFuzzer. At last, Section 9 concludes this paper. To foster further research, we have released HNPFuzzer under an open source license at https://github.com/veltavid/HNPFuzzer.

2 Related Work

AFLNET [14] is a fundamental network protocol fuzzer. Inspired by AFLNET, a number of protocol fuzzing schemes have been being proposed to improve fuzzing performance. Overall, the related work mainly optimize AFLNET in three directions, i.e., constructing good test cases sent to the SUT, extracting fine-grained state machine to proper lead the fuzzing process and accelerating fuzzing iterations.

2.1 Improving coverage rate of fuzzing by constructing good test cases of network protocols

Test cases in network protocol fuzzing are much more difficult to generate than those in common terminal software fuzzers. It is quite common for the client to interact with the SUT several times to reach the deep logic paths. Any single message in the test case generated by fuzzer incompatible with the protocol specification leads to an early termination of the protocol. Generation and mutation based message construction approaches are the most two important schemes in existing schemes. Generation based schemes use the TS files to research the meaning and range of a field in an entry. In this way, the test cases are of great quality and it highly depends on the professional skills. On the contrary, the mutation based schemes are more automatic and they construct the test cases based on a set of seeds in a random manner which are used to test the SUT.

NAFuzzer [20] applies the symbolic execution technique to generate the protocol format template as well as exploring the possible values of each message field to heighten the validity of test cases. This scheme is designed based on the key insight that similar message fields being parsed at similar locations in the program. However, symbolic execution has an apparent difficulty in analyzing complicated software and this also suits the complex protocols such as RRC and NAS protocols in 5G [21].

To improve NAFuzzer, Atomic [22] introduces the natural-language processing and machine learning techniques to network protocol test field. Particularly, they propose a method that extracts protocol states and event message structures corresponding to each state by analyzing grammatical information acquired from the protocol documentation. Based on the extracted message templates, they can generate the test cases for the SUT. The generation-based fuzzing schemes suits only the protocols that of detailed technical specifications.

In real life, some protocols, such as the private protocols, usually have no public specifications and a number of smart mutation based fuzzing schemes are proposed. SGPFuzz [23] introduces a series of stacked intelligent mutation strategies to ensure the validity of mutated messages. It also modifies the length fields, which lessens the influence of broken field dependency introduced by mutation.

PAVFuzz [24] proposes a relationship learning strategy to resolve the message fields dependency problem and utilizes the maintained relation table to assist the mutation process. The key elements in each state are automatically identified and assigned with different weights dynamically to avoid wasting time and computing resources.

2.2 Improving coverage rate of fuzzing by extracting fine-grained state machines of network protocols

A protocol state machine is an abstraction of state transition rules that can be used to guide the fuzzer when search-ing an enormous state space. Plenty of works aimed at building better state machine in terms of accuracy and overhead are proposed since AFLNET first introduced the state to guide the protocol fuzzing process.

StateAFL [17] calculates the fuzzy hash value of long-lived allocated memory to represent a certain state. This strategy provides StateAFL with chances to distinguish the different states with same response codes while AFLNET cannot, but requires heavy workloads of state calculation as well as comparison.

StateFuzz [25] and NSFuzz [26] are aware of above drawbacks of StateAFL and AFLNet. Both of them turn to identifying the critical variables in the target application with static analysis technique. The former also utilizes symbolic execution to achieve this goal. Mapping the critical variables to protocol states, they achieve a balance between accuracy and efficiency in the process of building state machine.

IJON [27] points out the significance of analysts engaging in the fuzzing process and extends AFL-based fuzzers to be directed by manually added annotations. With the help of annotations, fuzzers can easily reach the states that they cannot discover on their own.

Liu et al. [28] indicates another flaw in AFLNet that it lacks a reasonable way to evaluate the exploration-exploitation trade-off of states. Therefore, they propose a new state selection algorithm named AFLNetLegion and achieve slightly better performance compared to AFLNet.

2.3 Accelerating fuzzing iterations

Another bottleneck of network protocol fuzzing is its low speed due to the cumbersome steps of restarting the services of network protocols. Moreover, we need to feed the server a well-defined sequence of messages to lead the network protocol state to the state under test after restarting the server. The deeper of the state under test, the more time we need to do the preparatory work. This is unbearable for the fuzzing process especially for a complicated protocol.

The low speed has a bad effect on searching the state space of a protocol, because the fuzzer cannot locate the important states containing protocol vulnerabilities [28]. This can be explained by the fact that if the throughput of fuzzing process is low, all the states trend to perform equally bad and no valuable indicator can lead the fuzzer to the important paths that have more vulnerabilities in potential. Consequently, the performance of fuzzers decreases.

To tackle this challenge, an intuitive consideration is whether can we decrease the counts of restarting the network services in the fuzzing process. Inspired by this idea, the snapshot-based network protocol fuzzing schemes are proposed. The snapshot technologies based on virtual machine [29]–[31] have been widely researched. However, considering that a virtual machine works on the server and the network service works on the virtual machine, the execution efficiency greatly decreases compared with running the service directly on the server.

NYX-Net [32] is an incremental snapshot based fuzzer for network protocols. It employs a modified version of QEMU and KVM to design the hypervisor-based snapshot approach. For each target state, a corresponding snapshot is created and when the target state changes, another snapshot is recreated based on the root snapshot. With this method, they eliminate the time cost derived from sending same

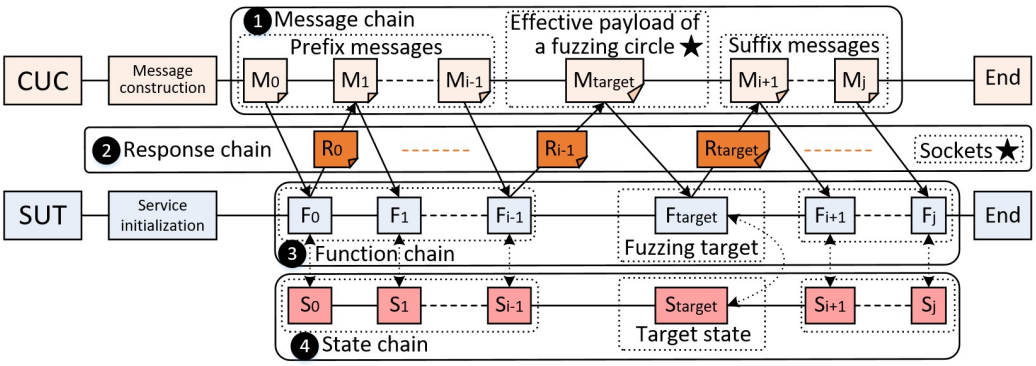


Fig. 1: Workflow of a network protocol fuzzing iteration.

prefix messages repeatedly. Besides, NYX-Net emulates the network transfer with the assistance of virtualization, which gain another acceleration despite the overhead of running in the virtual machine.

SNPSFuzz [16] employs a new snapshot tool named CRIU [33] to dump the context information when the network protocol program is in a specific state and this is more lightweight and efficient than the virtual-machine based approach. Then, the fuzzer restores the corresponding snapshot when the target state needs to be fuzzed. Apparently, compared with restarting the whole program in the initial phase and leading it to the target state, restoring the target environment and state based on snapshot is much faster. SNPSFuzz also designs a message chain analysis algorithm to explore deeper network protocol states.

Except for the snapshot-based schemes, there are some other works that can speed the fuzzing process. The approach in [26] implement a new I/O synchronization mechanism to avoid manually added waiting time, replacing the INET network socket with UNIX socket [34]. [35] changes the location of forkserver to reduce the amount of initialization work and using a custom in-memory file system to bring down the cost of file operations. Each of these improvements can make a slight speed-up, and the work [35] finds their performance are able to accumulate when aggregating such components.

Establishing a connection and reading data from it is far slower than reading from a file. A similar work to our idea is *libpreeny* [36] which attempt to entirely hook the network interfaces and returns the file descriptor of *stdin* instead of network sockets. Though *libpreeny* improves the speed of information transmission, the fidelity is low and many socket operations are not support by *stdin*. Therefore, it cannot properly support network protocol fuzzing.

3 Motivation of HNPFuzzer

3.1 Workflow of a network protocol fuzzing iteration

In a network protocol, at least a server and a set of clients are involved and they are in general connected by the network. In fuzzing process, we usually put both the server program and client program together in a local machine for the sake of convenience. Fig. 1 presents the workflow of a network protocol fuzzing iteration and the process is executed by the cooperation of the client under control (CUC) and the server under test (SUT). In this paper, CUC is assumed to be totally controlled by the fuzzer and we can send any message to SUT. There are four chains involved in the process, i.e., the message chain of CUC, response chain from SUT, function chain and state chain of SUT, and we denote them as M, R, F and S, respectively.

In the initial phase, the CUC constructs the test sequence of messages for this test iteration, i.e., the message chain M. The prefix messages, i.e., $M_0, M_1, \cdots, M_{i-1}$, are employed to lead the protocol state of SUT to the target state S_{target} which is focused by the fuzzer. Once the network protocol reaches the target state, the CUC sends the effective payload of this fuzzing iteration M_{target} to SUT. In general, the prefix messages keep unchanged for a constant protocol state S_{target}. Meanwhile, the message M_{target} which is fed to the function related to S_{target}, i.e., F_{target}, needs to be updated in each test iteration. In real life, both the mutation and generation based message construction schemes can be employed to form M_{target}. At last, the suffix messages, i.e., $M_{i+1}, \cdots, M_j$, are also sent to SUT for the full test in this iteration. Usually, the suffix messages cannot be ignored, because the effects of a request may do not appear immediately and however they can affect the behaviors of processing the suffix messages.

On the SUT side, existing fuzzers usually need to restart the service instance when a new test iteration begins. This is reasonable considering that the program environment may be modified after a test iteration and hence we need to reset the program environment. In this way, the results of the next test iteration are avoided to be contaminated. After initialization, the SUT is ready to receive and process the requests from CUC.

In fuzzing process, the CUC sends the messages in M to the SUT in order. Though the server side and client side are close in physical space, they are separated with each other in logic considering that they need to communicate by socket interfaces. To synchronize the behaviors of CUC and SUT, the timing of sending the next message is often decided based on the response from the SUT.

When SUT receives a message from the CUC, it employs the function corresponding to the current protocol state to process the message. If the message does not match the protocol state, the function corresponding to the current state cannot resolve and process the received message. As a result, the SUT directly drops the message and respond the CUC with an *ERROR* message. On the contrary, the SUT processes the message and sends the response to CUC. Meanwhile, the protocol state is changed to the next state in state chain S. The CUC and SUT iterate the above process until the fuzzing process is completed. Similar to the test cases, the response messages are also transmitted through

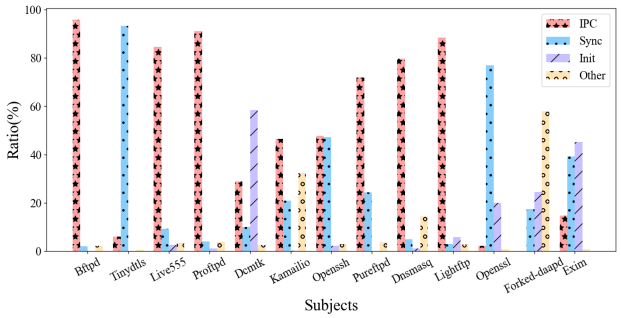


Fig. 2: Time consumption distribution in a fuzzing iteration.

the socket interfaces.

3.2 Analysis of time consumption distribution in a network protocol fuzzing iteration

Based on the workflow of a fuzzing iteration, this section analyzes the constitution of time consumption in an iteration. We first conducted an experiment to get the proportion of time cost derived from socket-based interprocess communication (IPC), synchronization, initialization and other behaviors during a fuzzing iteration. To measure the time of different behaviors, we patched a set of functions including $net\_send()$, $net\_recv()$, $run\_target()$, $send\_over\_network()$ and $write\_stats\_file()$ in AFLNet and ran each service in ProFuzzBench for 10000 times. The distribution of average time consumption for different behaviors in network protocols are presented in Fig. 2. Specifically, it can be inferred that "other" behavior is the most time-consuming part in *Forked-daapd* (57.8%). We manually ran *Forked-daapd* and found it took a long time to terminate the server after it received a SIGTERM, which is probably the most time-consuming part in "other" behavior.

3.2.1 Time consumption of socket based data transmission between CUC and SUT

From the Fig. 2, we can know that socket-based IPC takes the majority of time (from 44.3% to 95.8%) in 8 out of 13 targets. Therefore, we can infer that the latency caused by socket interfaces is one of the most important reasons for the low efficiency of network protocol fuzzing.

High-speed IPC means a higher throughput in protocol fuzzing and it is essential for fuzzers to discover more illegal behaviors and trigger more crashes. However, most existing protocol fuzzers leverage socket based IPC to connect CUC and SUT for the sake of convenience. Though some of them [34], [35] choose UNIX domain socket rather than network socket to speed up IPC, the data transmission speed is still not fast enough according to [37]. Therefore, replacing the socket based IPC with more efficient mechanism, such as shared memory, is promising to further accelerate existing network protocol fuzzers.

3.2.2 Time delay caused by synchronization

In software fuzzing process, the test cases can be directly fed to the program any time without time delay. However, the synchronization problem in network protocols cannot be ignored. Fig. 2 shows the significant latency of synchronization in Tinydtls (93.2%) and *Openssl* (76.9%). This type of latency is introduced by $poll()$ invocations of clients, which is also related to the socket interfaces.

```
./s_server.c
1009 int s_server_main(int argc, char *argv[])
1010 {
...      // initialize objects, parse command-line options and check the correctness of options
1730     if (nocert == 0) {
1731         s_key = load_key(s_key_file, s_key_format, 0, pass, engine,
1732                          "server certificate private key file"); // load a private key from a file
...          // error check
1738         s_cert = load_cert(s_cert_file, s_cert_format,
1739                            "server certificate file"); // load a certificate from a file
...          // branches that are not executed during fuzzing
1767     }
...      // initialize a SSL_CTX object
2036     if (dhfile != NULL)
2037         dh = load_dh_param(dhfile);
2038     else if (s_cert_file != NULL)
2039         dh = load_dh_param(s_cert_file); // loads Diffie-Hellman parameters from a file
...      // set members in the SSL_CTX object
2233     do_server(&accept_socket, host, port, socket_family, socket_type, protocol,
2234               server_cb, context, naccept, bio_s_out); // listen for incoming connections
...      // cleanup
2276 }

./s_socket.c
206 int do_server(int *accept_sock, const char *host, const char *port,
207                int family, int type, int protocol, do_server_cb cb,
208                unsigned char *context, int naccept, BIO *bio_s_out)
209 {
...      // initialize a server socket "asock"
323     for (;;) { // accept() loop
...          // check socket type and initialize "ourpeer"
336         do {
337             sock = BIO_accept_ex(asock, ourpeer, 0); // create the connection socket
338         } while (sock < 0 && BIO_sock_should_retry(sock));
...         // error handling
344         BIO_set_tcp_ndelay(sock, 1);
345         i = (*cb)(sock, type, protocol, context); // Messages exchange
...         // Update naccept and stop accept() loop if naccept reaches 0
384     }
...     // Cleanup
393 }
```

Fig. 3: A code snippet of *openssl*.

Specifically, the time delay of synchronization is minimized when the CUC and SUT send messages one by one in order. However, sometimes one side may need to send a series of messages before it receives the next message. For example, a TLS client may send a ChangeCipherSpec message after completing a TLS handshake. Then the client should continue sending application data to notify the server that the handshake has been completed. Therefore, each time the client (AFLNET) calls $net\_recv()$, it does not know whether it can receive a response or not. To solve this problem, AFLNET introduces the $poll()$ and set a timeout. In the ideal situation, a timeout occurs in $poll()$ implies that the client cannot receive a response this time, while no timeout in $poll()$ indicates there is a response can be read at present. Unfortunately, it is necessary to take the time of processing message in the server into account in the practical situations. On the one hand, even one-to-one correspondence of requests and responses may cause a timeout in $poll()$ if the timeout argument passed to $poll()$ is too small. On the other hand, a large timeout argument of $poll()$ is a waste of time when the client needs to send multiple messages before receiving one response. In summary, the manually selected $timeout$ of $poll()$ can be either too short or too long and it is hard to calibrate due to the uncertainties in the system. The root cause of this problem is that the client does not know the proper time instance to send or receive messages which depends on the action of the server. This motivates us to add a notification mechanism between the client and the server. Note that, this mechanism is of great burden to be realized in socket based data transmission process and fortunately it can be realized with the assistance of shared memory in HNPFuzzer.

3.2.3 Time cost of server initialization

At last, we discuss the time consumption of server initialization process. With regard to *Dcmtk* and *Exim*, although the socket-based IPC and synchronization together take up a considerable ratio (39.0% and 54.0%) of the whole time con-

```
1  while true do // accept loop entry
2      Do some preparations
3      ACCEPT incoming socket connection // main event loop entry
4      Do some preparations
5      while connection is open do // network event loop entry
6          WAIT for incoming request
7          PARSE request
8          Do something
9          SEND response to client
10     end while
11     CLOSE connection socket
12 end while
```

Fig. 4: The pseudocode used to clarify main event loop.

sumption, initialization takes the largest percentage (58.5% and 45.2%) among four behaviors. Besides, it also costs a considerable time to finish initialization in *Openssl* (20.1%) and *Forked-daapd* (24.6%). Unlike the terminal software, servers need to do a quantity of initialization behaviors, such as reading configurations and establishing a socket, before they start to accept connections. These behaviors would result in considerable time consumption in fuzzing process, because existing protocol fuzzers [14], [17] tend to directly kill the server process after a fuzzing iteration. In other words, the fork-server in existing fuzzers needs to fork a child process every time the fuzzer starts a new session and the new child process has to do all the initialization behaviors again.

We use a simplified code snippet from *Openssl* [38] as an example to illustrate this problem. As shown in Fig.3, the function $do\_server()$ contains a typical accept loop (line 323-384), which calls $BIO\_accept\_ex()$ to wait for incoming connections and conducts a session with the client (line 345). However, before entering the main event loop, the server needs to do quite a few preparations like initializing a SSL_CTX object (line 1848-2036 and line 2040-2220) and loading cryptographic key material from files (line 1730-1767 and 2036-2039). Note that there are many other lines of code that are not executed due to the lack of corresponding command line options. Therefore, the time cost can be even larger when more options are provided.

The main event loop we mentioned above is a part of the accept loop and it is not a specific loop in code as the network event loop in [26]. Actually, the main event loop represents one connection between the client and the server. This is why we can consider the server has entered the main event loop after it creates the connection socket through $accept()$. To make it clearer, we use a piece of pseudocode to facilitate comprehension. As shown in Fig. 4, line 1-12 is the accept loop; line 3-11 is the main event loop, which contains the network event loop (line 5-10). We define the main event loop in this way because NSFuzz has proved that it difficult to locate the network event loop automatically. Moreover, the extra lines (line 3, 4 and 11) should not affect the protocol state as they have nothing to do with messages from the client. Therefore, if the state in line 3 is the same as that of the last connection, the state in line 5 should also be the same. Notably, NSFuzz cannot use the main event loop because they must insert a raise (SIGSTOP) statement between line 5 and line 6, otherwise the server cannot notify the client right before it invokes $recv()$.

To decrease the time consumption of server initialization, we introduce a persistent mode into HNPFuzzer to prevent server processes from terminating in certain conditions. Different to the persistent mode in AFL [7] which requires manual implementation, the persistent mode in HNPFuzzer is totally automated via the connection controller. More details about the termination of server processes can be found in Section 5.3.2. The challenge is that we must ensure the consistency of states at the entry of main event loop, or the following states may change and thus the validity of fuzzing cannot be guaranteed.

Except for the time consumption of the above phases, the time cost of message processing by functions in SUT is not our consideration which is out of scope for this paper.

3.3 The main idea of HNPFuzzer

By analyzing the process of a fuzzing iteration and its time consumption constitution, we can conclude that the period of a test iteration can be greatly shortened in the following several aspects.

- The socket based message transmission between CUC and SUT is quite time consuming. We shorten the time consumption in HNPFuzzer by emulating network functions based on shared memory.
- We can carefully redesign the synchronization mechanism based on shared memory without the influence of network situation uncertainties to decrease the time cost of synchronization.
- We can send several test cases rather than only one in a server instance to decrease the time cost of server initialization under some situations. Consequently, we introduce the persistent mode into HNPFuzzer.

HNPFuzzer is realized via hooking and it is totally transparent for the CUC and SUT. This means that both the CUC and SUT think that they indeed communication with each other by socket interfaces rather than the shared memory. Though the goal of snapshot based network protocol fuzzing schemes is also shortening the time consumption of a test iteration, the main idea between them and HNPFuzzer are totally different with each other. Therefore, they do not compete with our scheme and on the contrary, they can cooperate with HNPFuzzer to improve the efficiency of network protocol fuzzing.

4 Overall Framework of HNPFuzzer

As shown in Fig. 5, the workflow of HNPFuzzer mainly comprises two phases, i.e., the initialization phase and the fuzzing loop phase. This section discusses about the initialization phase and provides an introduction about the fuzzing loop in the high level. The connection controller and synchronizer in the fuzzing loop is the core of HNPFuzzer and they will be discussed in Section 5 and 6, respectively.

4.1 Initialization phase

In most existing network protocol fuzzing schemes, the service instance will be directly killed after a fuzzing iteration. This is reasonable considering that the running of a fuzzing iteration may contaminate the environment of the instance. To improve the fuzzing efficiency, we introduce the persistent mode into HNPFuzzer. The persistent mode was first proposed by AFL which tries to fuzz the target software several times without the need to fork a new process.

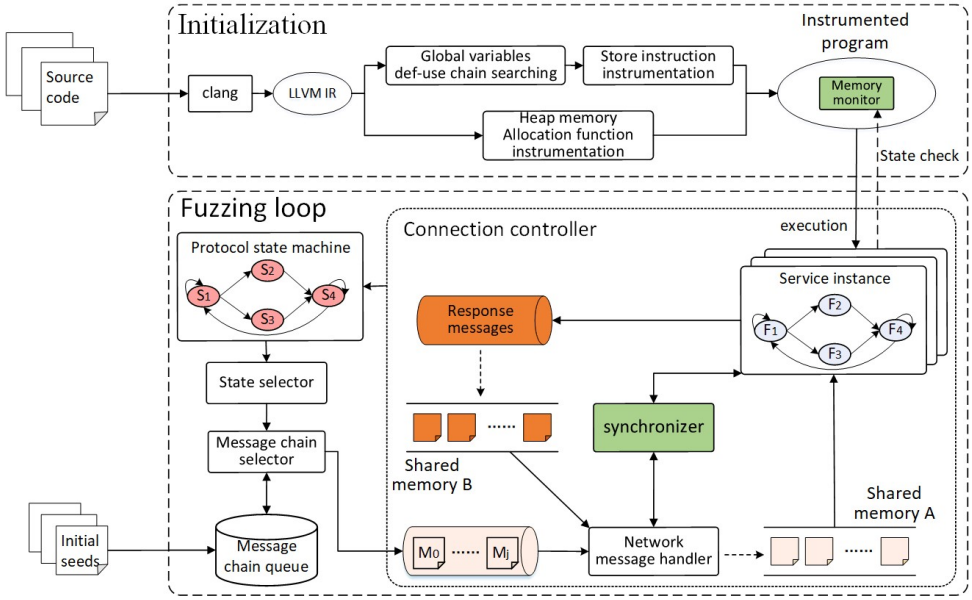


Fig. 5: Overall framework of HNPFuzzer.

The challenge to adapt this idea to the stateful network protocol fuzzing is to keep its initial state at the entry of the main event loop unchanged. In other words, the operations occurred in one connection should not have effects on other connections. One possible solution is to watch the stability value and compare its value to the non-persistent fuzzer. However, this solution requires the fuzzer to loop several times before realizing it is doing an invalid fuzzing.

To ensure that the state of program is not changed at the entry of main event loop in the persistent mode, we need to check if any long-lived data is modified, because long-lived data is widely accepted to strongly correlated to program states. Besides the state modification due to long-lived data modification, memory leaks in the main event loop can affect later connections in terms of speed when persistent mode is on. Generally speaking, the areas to be monitored can be divided into two categories as below. To implement the tracing, we extended the afl-clang-fast tool to instrument the target application based on LLVM framework. Specifically, we instrumented specific store instructions (i.e., whose destination operand is a global variable or a memory area pointed to by a global variable) for the long-lived data tracing and specific call instructions (i.e., whose destination operand is a memory management library function like $malloc()$ or $free()$) for the heap memory tracing. Notably, the instrumented code about the heap memory tracing does not come into effect until the main event loop is entered to meet the second category definition. The instrumented probes are denoted as memory monitor.

1) Long-lived memory areas

The long-lived memory areas we define consist of global variables and the memory areas that can be accessed through global pointers. Since it is possible for any functions to access these areas to change the program state and the change may be preserved to the next loop. To trace this type of memory areas, HNPFuzzer searches and instruments store instructions according to Algorithm 1.

As Algorithm 1 shows, we search the instructions that are direct or indirect users of any global variable. Each search starting with a single global variable is based on

Algorithm 1 Long-lived data related store instructions searching

Input: The list of global variables, G;
Output: The instructions that should be instrumented, I;

```
1:  visited ← [];
2:  I ← [];
3:  for each g ∈ G do
4:      queue ← [g];
5:      while queue is not empty do
6:          ci ← queue.pop();
7:          for each user ∈ userlist(ci) do
8:              if user ∈ visited then
9:                  continue;
10:             end if
11:             visited.insert(user);
12:             if isStoreInst(user) then
13:                 po ← getPointerOperand(user)
14:                 if po == ci then
15:                     I.insert(user)
16:                 end if
17:             else
18:                 queue.push(user)
19:             end if
20:         end for
21:     end while
22: end for
23: return I;
```

Breadth-First Search (Line 5-21). The direct def-use relationship can be obtained through the userlist of a value provided by llvm framework (Line 7). And in the process of searching, we collect the store instructions that may modify the memory area that may be accessed through a global variable (Line 12-16). Notably, the global variables introduced by afl, i.e., $\_\_afl\_area\_ptr$ and $\_\_afl\_prev\_loc$, should be excluded from the input global variables list.

2) Memory areas allocated in the main event loop

The heap memory areas are always manually allocated by the users and the OS would not reclaim them until they are freed by the user. So memory leaks occur if these areas are not released when they are no longer needed. These memory leaks may pile up and neutralize the speed boost brought by persistent mode. We only trace the areas allocated in the main event loop because the persistent mode does not exacerbate the memory leaks out of the loop. To trace this type of memory areas, HNPFuzzer

instruments call instructions whose destination operand is dynamic memory management library functions, e.g., $malloc()$, $calloc()$ and $free()$. And the further check is placed in the memory monitor, since it is difficult to locate the main event loop during the compilation.

4.2 Fuzzing loop phase

After the initialization phase, HNPFuzzer enters the fuzzing loop which is the main component in the whole fuzzing process. In the fuzzing loop, we first need to analyze the target protocol under test, implement a set of protocol-specific message parsers and construct the test cases for fuzzing loop.

The fuzzing loop creates a connection through $connect()$ between the CUC and SUT per iteration. Particularly, the connection controller takes responsibility for the communication between the client and the server. However, HNPFuzzer does not transmit messages through the socket interface, which is only used to bypass the numerous socket-related consistence validations in the server, such as the validations after functions like $getsockname()$. After building the connection, the connection controller starts to transform the specific socket I/O operations to shared memory reading and writing operations. In this way, the fuzzing efficiency is greatly improved. In the controller, the shared-memory-based synchronizer is used to ensure the connection controller transmits the messages at an appropriate time which shorten the time delay of messages.

The fork server is responsible for preparing the service instances which are the targets for test. Usually, an instance can be used for only one test iteration after which it would be killed and a new instance is generated. However, the persistent mode of HNPFuzzer allows an instance to be tested with multiple test iterations which further improves the fuzzing efficiency.

5 Connection Controller

The connection controller is the most important component in our system. Its first function is to implement the persistent mode. The second function is to support the messages exchange through shared memory between the client and the server. We divide a connection between CUC and SUT into three stages, i.e., connection creation, messages exchange and connection termination. The whole process is controlled by the connection controller as shown in Fig. 5.

The first function is related to the connection creation and connection termination stages. After the connection is created, we believe the server has entered the main event loop. Then the instrumented probes are enabled to collect information. The first function of collected information is to detect long-lived data modification representing the global state modification. The second function is to track the allocations in order to protect the persistent mode server from memory leaks. When the connection is terminated, the connection controller will examine if the process violates two rules to decide whether to terminate the server process. The details about how the connection controller detects the connection creation and termination are illustrated in Section 5.1 and 5.3.

The second function is implemented in the messages exchange stage, which is based on LD_PRELOAD hooking. Considering that the network protocols usually transmit messages through sockets, the connection controller intervenes in the messages exchange between the client and the server by hooking specific library functions related to the socket interface. For example, the hooked $send()$ writes the message into shared memory instead of calling the true $send()$ to send it. More implementation details are discussed in Section 5.2.

5.1 Creating the connection

To identify the socket which is used to communicate with the client, we first analyze the client/server model. Generally speaking, the server invokes $socket()$, $bind()$, $listen()$ and $accept()$ successively to create a connection to the client. We notice that $bind()$ and $accept()$ are very crucial to build the connection socket. In HNPFuzzer, the connection controller automatically extracts the port number from the second argument of $bind()$ and compare it to our target port provided by "-N" option to get the server socket. Then the connection socket can be obtained by comparing the server socket with the first argument of $accept()$. However, it is inadequate to just save the server socket and the connection socket due to the fact that file descriptors (including the socket connections) can be duplicated as the result of invoking $fork()$, $dup()$ or $dup2()$. In fact, it is a common design pattern that the server forks a child and duplicates the connection socket to standard POSIX file descriptors.

We introduce a reference count to represent the number of processes with connection sockets and a array named fd_table to track the duplication caused by $dup()$ and $dup2()$. The former is maintained in shared memory to detect the connection termination, while the latter resides within the process address space to assist checking if a connection socket exists in current process. It is clearly that any socket duplication taking place in $dup()$ and $dup2()$ does not affect the reference count but triggers a copy operation in fd_table. For example, a $dup2(fd_1, fd_2)$ invocation will result in the operation of $fd\_table[fd_2] = fd\_table[fd_1]$. The reference count is decreased by 1 when the connection socket and its duplication are all closed in a process. And it is increased by 1 when $accept()$ creating the connection socket or $fork()$ after the connection socket has been created. We believe the connection is closed when the reference count drops to 0.

After obtaining the connection socket, the connection controller will change the connection state to *opening*. The connection state, denoted as ρ, represents the status of connection created between the client and the server each fuzzing iteration. There are three possible assignments for it, i.e., *uninitialized* before the creation of the connection socket; *opening* after the connection socket is created; *closed* after the connection is closed (the reference count is 0). The utilization of it is to support operations that occur in certain stage. For example, the connection controller should not invoke next $accept()$ until the current connection is closed in order to do persistent mode related examinations at the entry of the main event loop.

In asynchronous servers, library functions used to implement I/O multiplexing like $select()$, $poll()$ and $epoll()$ are likely to be called before invoking $accept()$. This is different to the traditional synchronous servers. In both situations, the synchronizer takes the responsibility for arranging the workflow of the loops that invoke these functions. More details of the synchronizer will be discussed in Section 6.

5.2 Exchanging messages based on shared memory

After the connection between CUC and SUT, they begin to transmit the messages, i.e., the test cases and responses, with each other. There are two directions of the message flow between the CUC and SUT. In one direction, the test cases are sent from the CUC to the SUT. In the other direction, the responses are sent from the SUT to the CUC. Particularly, we employ two fix-sized buffers in data transmission process considering that the SUT may need to receive and send messages simultaneously. The fix-sized buffers is allocated by function $shmget()$. As far as we know, there is no way to expand a shared memory area other than to recreate a larger one and remap to it. Hence the message larger than the buffer must be split into pieces to transmit. The algorithms of transmitting messages from the CUC to the SUT is given below.

Algorithm 2 demonstrates how the CUC sends a single message to the SUT. The CUC first blocks itself until the SUT allows it to send message (Line 1). Then $Size_m$ in SA is initialized as $Size_M$ (Line 2) for the purpose of telling the SUT the length of the message. In the main loop (Line 4-17), the CUC writes the message into the shared memory buffer. In the inner loop, if there is remainder message in the shared memory buffer (Line 10), the CUC will leverage the synchronizer (Line 11) to inform the SUT that it should continue to read the message. The root cause is that we cannot manipulate the size of receiving buffer in the SUT. For example, if the SUT invokes $recv(fd.buf, buf, sizeof(buf), 0)$ and the size of buf is smaller than $SA.buf$, it is not able to read all of the message in $SA.buf$. So the SUT needs to invoke $recv()$ multiple times and each time it finishes $recv()$ the session state will be changed by the Synchronizer. Therefore, the CUC must notify the Synchronizer to restore the session state (Line 11) to allow the SUT to continue its $recv()$ invocation. More details of synchronization will be discussed in Section 6. If the connection is closed by the SUT in the process of message exchange, the CUC will return -1 to symbolize the occurrence of unexpected behaviors (Line 12-15). The structure SA is shared between the CUC and SUT so that the two sides can exchange messages. It comprises three elements including $Size_m$, buf and $Size_b$. $Size_m$ represents the length of the message to be sent at present; buf is a fixed-length buffer which holds the whole message or a part of the message depending on whether the length of message is smaller than the buffer size; $Size_b$ represents the length of unread message in buf. The CUC has to wait until the buf in SA is cleaned up by the SUT before entering a new loop iteration, because we do not want the data written in the following iterations to overwrite the data that the SUT has not read.

Algorithm 3 depicts how to read the messages in the SUT. The SUT would not start to receive message until the CUC finishes sending (Line 1). In the main loop, the SUT reads the message as much as possible if the buffer in the SUT is smaller than the shared memory buffer (Line 5-10). Otherwise, after the SUT finishes the message reading (Line 12-17), it waits for the remainder message from the CUC with the help of synchronizer if the message has not been completely transmitted (Line 18-19). In this case, the blocked CUC will resume and send the rest messages to the SUT.

In the process of sending response messages from the SUT to the CUC, we introduce an additional buffer denoted as buf_c. When the SUT invokes $send()$, it will first write message to buf_c instead of putting the message into the shared memory buffer directly. The messages kept in the buffer buf_c are not sent to CUC until the SUT starts waiting for new messages from the CUC or closes the connection socket. More details about the implementation of buf_c will be described in Section 6.3. The purpose of this design is to tackle the situation that the SUT sends multiple messages in response to one CUC message. The algorithm of sending messages from the SUT to the CUC is similar to the Algorithm 2 and 3, except that the receive buffer size controlled by the CUC is always larger than the shared memory buffer size. For the sake of length limitation, sending messages from SUT to CUC will not be discussed in this paper.

Algorithm 2 Message sending in the CUC

Input: The structure, SA, in shared memory used to exchange messages; the size, $Size_s$, of the buffer in SA; message, M, to be sent; the size, $Size_M$, of M;
Output: The number of bytes, l, successfully sent this time;
1: Synchronize with the server;
2: $SA.Size_m \leftarrow Size_M$;
3: $l \leftarrow 0$;
4: **while** $Size_M > 0$ **do**
5: $Size_t \leftarrow min(Size_M, Size_s)$;
6: $memcpy(\&SA.buf, \&M[l], Size_t)$;
7: $SA.Size_b \leftarrow Size_t$;
8: $Size_M -= Size_t$;
9: $l += Size_t$;
10: **while** $SA.Size_b > 0$ **do**
11: Synchronize with the server;
12: **if** the connection is closed **then**
13: $l \leftarrow (-1)$;
14: **return** l;
15: **end if**
16: **end while**
17: **end while**
18: **return** l;

Algorithm 3 Message receiving in the SUT

Input: The structure, SA, in shared memory used to exchange messages; the total bytes, $Size_r$, of received message; the buffer, buf, used to store the received message; the size, $Size_b$, of buf;
Output: The number of bytes, l, successfully received this time;
1: Synchronize with the client;
2: $l \leftarrow 0$;
3: **while** $Size_b > 0$ **do**
4: **if** $SA.Size_b > Size_b$ **then**
5: $memcpy(\&buf[l], \&SA.buf[Size_r], Size_b)$;
6: $SA.Size_b -= Size_b$;
7: $Size_m -= Size_b$;
8: $l += Size_b$;
9: $Size_r += Size_b$;
10: $Size_b \leftarrow 0$;
11: **else**
12: $memcpy(\&buf[l], \&SA.buf[Size_r], Size_b)$;
13: $l += SA.Size_b$;
14: $SA.Size_m -= Size_b$;
15: $Size_b -= SA.Size_b$;
16: $SA.Size_b \leftarrow 0$;
17: $Size_r \leftarrow 0$;
18: **if** $SA.Size_m > 0$ **then**
19: Synchronize with the client;
20: **else**
21: **break;**
22: **end if**
23: **end if**
24: **end while**
25: **return** l;

5.3 Terminating the connection based on memory monitor

5.3.1 Memory monitor

In memory check process, the memory monitor initially determines if the SUT is in the main event loop currently through checking the connection state ρ which is set by

the connection controller, and then traces the memory areas allocated in the main event loop by maintaining their start addresses in a set of chunks denoted as θ. If any of these memory areas is freed, the memory monitor will remove the corresponding address from θ. With regard to long-lived memory, the invocation of the memory monitor implies that there is a store instruction writing a certain long-lived memory area. In this case, the memory monitor assigns true to a variable denoted as G, representing the global state has probably changed. It should be noted that we do not do any comparison or calculation about the contents of these memory areas for two reasons. First, the purpose of the persistent mode is to accelerate the fuzzing process, so we must minimize the overhead introduced by itself. Second, we expect the memory areas allocated during a connection are all freed after the connection ends, or they may affect the following connections regardless of their contents.

5.3.2 Connection termination

The workflow of the connection termination is shown in Fig. 6. In HNPFuzzer, we assume that the SUT is responsible for maintaining and closing the sockets that it creates. As a consequence, the connection is considered to have been closed when the reference count of the connection socket drops to 0. The connection state ρ is also marked as *closed*. Once the connection is closed, the connection controller will verify if either of the two rules, namely the chunk set θ should be empty and the variable G should not be modified, has been violated. If θ is empty and G is false (the global state remains unchanged), the connection controller will raise a SIGSTOP to notify the forkserver of connection termination. Otherwise, it will exit directly. Then, the forkserver delivers status code to the fuzzer through pipe.

Before the fuzzer creates a new connection, it will interact with the forkserver and need to make a decision, i.e., let it resume the stopped process (persistent mode enabled) or fork a new process. The decision can be made based on the last process state. To check the state, we mainly focus on the trace of specific memory areas as discussed in Section 4.1. This is reasonable considering that some operations with inter-connection effects have been tackled in AFLNET. For example, the effects of file operations are eliminated by the cleanup script; the exceptions are handled by signal handler.

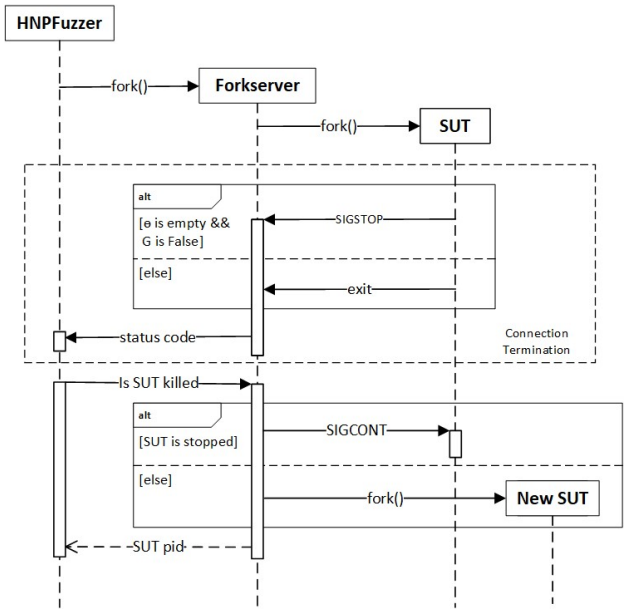


Fig. 6: The workflow of connection termination.

6 Shared-memory based synchronizer

The synchronizer is an important component of the connection controller. It is used to synchronize the state of different components during the fuzzing process.

6.1 Implementation of the two primitives

The synchronizer is implemented based on operating a variable called session state, and any operations on the session state should be atomic. We implement two functions, $shm\_wait()$ and $shm\_notify()$. The semantic of $shm\_wait()$ is to block the thread until it is notified that a specific session state has been reached. The semantic of $shm\_notify()$ is to set the session state in order to resume blocking threads. There are two alternative methods to implement the primitives, i.e., Futex and busy waiting. The semantics of FUTEX_WAIT and FUTEX_WAKE are the same as our primitives. As any other synchronization primitives like Mutex or Condition Variable use Futex as their building block, we do not take them into account. On the other hand, busy waiting, which does not invoke any syscall, holds the possibility of outperforming Futex. We conducted an experiment to compare their efficiencies.

To find out the better implementation, we performed an experiment based on our modified version of IPC-benchmark [37] and the results are shown in Table 1. The results suggest that the cases using shared memory make increases of 1.9x and 61.7x in the amount of throughput respectively compared to the Internet sockets. It can be observed that the *busy waiting* based mechanism is much faster than that based on the *Futex*. This might due to the overhead of context switching brought by the *Futex* system call. Therefore, we use *busy waiting* to implement the two primitives in HNPFuzzer.

6.2 Synchronization between the client and the server

In stateful network protocol fuzzing process, we need to mutate the messages corresponding to the target state and it is necessary to synchronize the client and the server. However, different SUTs are of distinct workflows depending on their protocols and implementations which increases the difficulty of synchronization. In real life, some of network protocols may violate the pattern of one response to one request. For example, the client can send a "change cipher spec" message and another encrypted message consecutively in TLS; the server can send two status codes of 150 and 451 in response to one "list" command message in *Lightftp*, which is an implementation of FTP. The challenge is how to match the requests and responses correctly in the context of fuzzing different protocols. Generally speaking, traditional stateful network fuzzers use $poll()$ with manually set timeout before $send()/recv()$ to handle this problem and unfortunately this pattern introduces an extra time delay to the fuzzing iteration. Particularly, if we set a short timeout parameter for function $poll()$, the result can be even worse as the stateful fuzzer may desynchronize from its state machine.

TABLE 1: Data transmission speed with different mechanisms

Method	Speed (msg/s)					
	1	2	3	4	5	Average
Internet sockets (TCP)	14028	14579	14311	14802	14628	**14469.6**
Shared memory (Futex)	27597	27909	28151	28273	26996	**27785.2**
Shared memory (Busy waiting)	869332	904737	880781	905049	901112	**892202.2**

The root cause of the above problem is that both of the two sides, i.e., the client and server, cannot predict what the other side is planning to do in the next step. To solve this problem, we introduce the session state into our scheme which is also stored in shared memory. The session state can be either "C" or "S". The state of "C" represents the client is sending or receiving a message and the server should block if it is about to send or receive a message. Similarly, the state of "S" represents the server is sending or receiving a message and the client should block if it is about to send or receive a message. Therefore, once any side finishes its current task or is unable to continue running unless the other side finishes its next task, it should use $shm\_notify()$ to change the session state to notify the other side to perform its next task. This action is literally the state transition. In this way, the synchronization between two sides can be achieved by modifying a variable in the shared memory rather than waiting blindly. Fig. 7 shows the possible session state transitions in the client and server and we introduce each of them in detail as follows. Considering the session state can only be "S" or "C" so that either the client or the server can run normally at any moment. It is worth mentioning that the wake-up signal lost cannot happen when using busy waiting, so the situation that both sides are being blocked due to one side missing the wake-up signal from the other side cannot occur. The situation that one side is waiting for a certain state while the other side is waiting for the message should not happen either, because hooked $recv()$ does not block in this case and the session state is bound to be switched.

a) $S \rightarrow S$: The client is blocked by $shm\_wait()$ (e.g., line 1 in Algorithm 2).
b) $C \rightarrow C$: The server is blocked by $shm\_wait()$ (e.g., line 1 in Algorithm 3).
c) $S \rightarrow C$: This transition occurs in following five conditions:
   i. The server finishes sending a message;
   ii. The message is longer than the shared memory buffer. The server tells the client to send the remains;
   iii. The server starts receiving messages but finds the shared memory buffer empty;
   iv. The client receives the SIGALRM;
   v. The server exits or stops.
d) $C \rightarrow S$: This transition occurs in the following four conditions:
   i. The client finishes sending a message;
   ii. The message is longer than the shared memory buffer. The client tells the server to send the remains;
   iii. At the beginning of each session;
   iv. The client is about to end the session. For example, the client finishes reading messages and has no more messages to send.

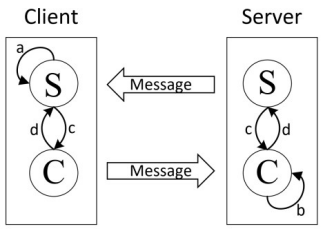


Fig. 7: Session state transitions in client and server.

In data transmission process, we employ a shared memory area to transmit the state of one side to the other side. In this way, both the CUC and SUT can make the best choice in proper time instance. Compared with the socket interface based data transmission, our scheme sensibly shorten the time delay caused by synchronization process.

Moreover, another challenge is how to track the events in state transitions. In HNPFuzzer, they are tracked in different ways and the most important two ways are presented as follows. First, some events can be detected by monitoring the inner variables such as $SA.Size_b$ and the length of $SA.buf$. Second, we can also track the events by hooking particular system functions such as $send()$. More details of can also be found in Section 4 and 5.

6.3 Synchronization among the parent process and child processes

Many SUTs need to invoke $fork()$ when communicating with the CUC. This brings another challenge that the response messages stored in buf_c can be sent repeatedly by multiple processes, because buf_c is duplicated as the $fork()$ duplicates the process. Therefore, we create buf_c as a shared memory object and map it to process address space with $mmap()$ so as to avoid duplication. This means that we must synchronize the size of buf_c among processes instead of using $realloc()$ to expand the buf_c directly. In detail, we introduce two variables, i.e., the size of buf_c denoted as $size_c$ and $size_{max}$. The former is shared among processes, and the latter is kept in each process address space. When a process needs to expand the buf_c, it will update both of the two variables. If a process finds its $size_{max}$ is not equal to $size_c$, this indicates that other processes have expanded buf_c. Then this process needs to update its $size_{max}$ and remap buf_c. Note that, these operations are atomic with the help of the lock implemented by *Futex*. However, locks alone cannot guarantee the atomicity of operations. Because deadlock may occur if a thread is interrupted by a signal after obtaining the lock. Therefore, our synchronizer leverages $pthread\_sigmask()$ to block the signals in current thread before it obtains the lock.

6.4 Synchronization among event loops

There are quantities of event loops in the server, some of which are crucial to the communication between client and server (e.g., accept loop and select loop). These loops can run concurrently in different threads or processes. Take *Openssh* as an example, the parent process runs accept loop in $server\_accept\_loop()$ while the child process runs select loop in $client\_loop()$. We must ensure that the accept loop does not invoke $accept()$ until the client loop ends. This is because the fuzzer only creates one connection at a time and the connection controller needs to check the θ and G in the memory monitor before opening a new connection. Hence the challenge is how to synchronize the execution order of these functions.

Our synchronizer will inspect the file descriptor set passed to these functions (e.g., $sockfd$ in $accept()$, $readfds$ and $writefds$ in $select()$, fds in $poll()$). If the server socket exists in the file descriptor set, the synchronizer will subsequently examine the ρ to see if there has been a connection opened. Then it either blocks the thread until the opened connection is finished or does nothing if there is no connection. Notably, the synchronizer will remove the connection socket from the file descriptor set if it is passed to the I/O multiplexing functions. This is because the connection controller never sends messages through sockets except for building the connection. The thread invoking the I/O multiplexing functions may get stuck if the timeout is infinite after removing the connection socket from the file descriptor set. In this case, we change timeout to 0 and add connection socket to the result because the connection socket is always logically readable/writeable with the help of high speed shared memory.

7 Performance Evaluation of HNPFuzzer

In this section, we evaluate the performance of HNPFuzzer and conduct a series of experiments to answer the following two questions.

1) *Can HNPFuzzer achieve improved fuzzing efficiency compared with the baseline network protocol fuzzers?*
2) *Can HNPFuzzer discover more vulnerabilities than other existing works in widely used network protocols?*

We have implemented HNPFuzzer based on the AFLNET project. In experiments, we compared HNPFuzzer with three mainstream open source network protocol fuzzers, i.e., AFLNET [14], AFLNWE [39] and StateAFL [17]. We first introduce the experiment setup in Section 7.1. To evaluate the performance of HNPFuzzer, we compared it with existing fuzzers in terms of fuzzing throughput and code branch coverage in a constant period as presented in Section 7.2, 7.3 and 7.4. At last, the discovered vulnerabilities are analyzed in Section 7.5.

7.1 Experiment setup

For the sake of fairness, all the experiments are conducted on a 64-bit Ubuntu 20.04 LTS (kernel version 5.15.0-56) with an Intel Core i7-11700 CPU (2.50GHz) and 16GB RAM. We chose *ProfuzzBench* [40] as the benchmark in simulation which is mainstream for stateful protocol fuzzing. The benchmark contains 13 network service applications including 10 widely used protocols such as *FTP*, *HTTP*, *SSH* and *SMTP*. Additionally, it includes not only protocols based on *TCP*, but also covers the protocols based on *UDP*. Considering that the *UDP* based network protocols have no conception of connections, the controller in HNPFuzzer cannot detect the termination of these protocols. Therefore, the persistent mode of HNPFuzzer will be disabled when fuzzing these protocols. In order to enhance the fuzzing reliability, *ProFuzzBench* applies many patches for these network service applications. They eliminate the randomness of applications and remove certain $fork()$ invocations to let the application exit after one connection is closed. However, the elimination of these $fork()$ invocations will disable the persistent mode of HNPFuzzer, so we remove these patches in the fuzzing process of HNPFuzzer.

Moreover, AFLNWE is a special version of AFL which sends the input through socket instead of writing to a file while AFLNET and StateAFL send a series of messages in a fuzzing iteration. Each message is possible to cause a state transition of the server, which means that the effectiveness of two executions is likely to be variable due to different number of processed messages even if both of them are one execution during fuzzing. In this case, it is inappropriate to judge the throughput of fuzzers by their execution speeds. Therefore, we use the message processing speed proposed in [16] as a metric. The second metric we employed for effectiveness comparison is the code branch coverage, which is also a widely used metric to evaluate the performance of fuzzers. We ran each target protocol service in a docker container for 24 hours and repeated each experiment for 10 times to reduce the impact of uncertainty in the process of protocol fuzzing. The average simulation results are presented and analyzed in detail.

7.2 Fuzzing throughput

The fuzzing throughput is a dominating factor for the efficiency of stateful fuzzing. On the one hand, the throughput is significant to the suitable target state selection. This can be explained by the fact that the score of a state is negatively correlated with the state occurrence frequency in response chains and is positively correlated with the number of new paths discovered in this state. The high fuzzing throughput reduces the possibility that a state discovering few paths gains a high score via its low occurrence frequency. On the other hand, the throughput determines whether the fuzzer is capable to generate enough test cases to explore the target state thoroughly. Therefore, the fuzzers with high fuzzing throughput is less likely to leave out the important states. In our experiments, we use the message processing speed, a more fine-grained metric than the execution speed, to represent the throughput of fuzzers.

The detailed results are shown in Table 2 and it can be observed that HNPFuzzer achieves a much higher throughput compared with the other three fuzzers on most network services. Specifically, compared with AFLNET, HNPFuzzer increases the throughput up to 44259.38% and on average the throughput is increased by 3966.09%. The high speed of HNPFuzzer should be credited to the fast shared memory with the help of synchronizer and the persistent mode which cuts the costs in initialization.

AFLNWE is the slowest fuzzer on average among four fuzzers. This is reasonable considering that it only sends one message in each testing iteration. Hence the server

TABLE 2: Fuzzing throughput of various fuzzers on ProFuzzBench

Service	Fuzzing Throughput (msg/s)			
	AFLNET	AFLNWE	StateAFL	HNPFuzzer
Bftpd	116.45	28.47(-75.55%)	50.2(-56.89%)	**580.68(+398.64%)**
Lightftp	124.79	36.09(-71.08%)	49.10(-60.65%)	**2129.93(+1606.84%)**
Proftpd	98.74	15.35(-84.46%)	86.21(-12.70%)	**108.29(+9.67%)**
Pureftpd	187.58	32.32(-82.77%)	146.76(-21.76%)	**906.79(+383.41%)**
Dcmtk	122.53	50.28(-58.96%)	25.28(-79.37%)	**76.97(-37.18%)**
Live555	153.71	43.58(-71.64%)	48.05(-68.74%)	**1146.90(+646.17%)**
Exim	24.57	6.04(-75.41%)	12.85(-47.68%)	**99.24(+303.99%)**
Forked-daapd	6.62	0.78(-88.19%)	5.14(-22.36%)	**104.65(+1480.79%)**
Openssh	78.57	41.98(-46.57%)	84.19(+7.15%)	**269.29(+242.75%)**
Openssl	41.72	6.91(-83.45%)	15.23(-63.50%)	**449.82(+978.08%)**
Dnsmasq	247.11	37.15(-84.97%)	69.74(-71.78%)	**2748.03(+1012.05%)**
Kamailio	49.19	6.26(-87.27%)	12.55(-74.49%)	**184.24(274.54%)**
Tinydtls	31.31	12.69(-59.48%)	24.91(-20.44%)	**13888.03(+44259.38%)**
Average		**-74.60%**	**-45.63%**	**+3966.09%**

initialization cost in AFLNWE takes a larger proportion than other fuzzers. StateAFL is also slower compared with AFLNET and HNPFuzzer due to its quantities of instrumentation operations as well as the other heavy overhead operations such as hash value computations. Another interesting observation is that the throughputs of HNPFuzzer for *Proftpd* and *Dcmtk* are not greatly improved. To analyze the possible reasons for *Proftpd*, we run *Proftpd* in HNPFuzzer for 10000 times and recording the time consumption of various operations. We find that the time consumption of IPC in HNPFuzzer on *Proftpd* only takes up 11.76% of the total time and 76.01% time was used to wait for the status code from the fork-server. Hence we can infer that the root cause is the latency from the post operations in the *Proftpd* server, which is avoided by killing the server process in other fuzzers. Moreover, the instrumentation in HNPFuzzer also slows the execution of service programs. Through the ablation experiments, we also obtain the reason for *Dcmtk* which is related to the persistent mode and this will be discussed in Section 7.3.

7.3 Performance breakdown

In order to explore the contribution of each component to the overall performance of HNPFuzzer, we have conducted a set of ablation experiments which can be divided into three groups, i.e., persistent mode, synchronization and shared memory IPC. In our ablation experiments, the patches of $fork()$ were preserved in the second and the third group where the persistent mode was disabled. Therefore, the results of these groups would not be lower than the actual level even if targets were equipped with the modified fork mechanism provided by Profuzzbench.

The experiments results are presented in Table 3. As can be seen, each component in HNPFuzzer is able to accelerate the fuzzing in varying degrees, and their combined effect results in the highest average acceleration. Note that, the last column in Table 3 is strictly the same with that in Table 2. It is worth mentioning that shared memory IPC cannot work without the synchronization mechanism. Otherwise a message may be overwritten before it is received, and the state machine cannot be built correctly. Therefore, in the third group of experiments, only persistent mode is disabled. Additionally, as we mentioned in Section 7.1, the persistent mode of HNPFuzzer will be disabled when fuzzing udp-based protocols.

It can be observed from the first column in Table 3 that the lone persistent mode can achieve a higher average throughput (+29.13%), though it slows down four services, i.e., *Proftpd*, *Pureftpd*, *Live555* and *Dcmtk*. In particular, these results are strongly related with the time consumption experiments in Section 3.2. Because the initialization time consumption occupies a minimal amount of time in *Proftpd*, *Pureftpd* as well as *Live555*, the persistent mode has no chance to improve their performance, but yields a decelerating impact due to its own overhead. On the contrary, in the services of *Exim*, *Forked-daapd*, *Openssl*, their initialization takes up considerable time and therefore they achieve more apparent speedup compared with those services with a fast initialization. The service of *Dcmtk* is one anomaly considering that it is decelerated by the persistent mode though it has a cumbersome initialization. Then we ran *Dcmtk* with memory monitors deactivated and persistent mode rules verification disabled separately. We found *Dcmtk* reached a higher throughput in both of cases, which implied 1) the instrumentation brought huge overhead to *Dcmtk*; and 2) the two rules were often failed to be met and the server could not actually ran persistently. Therefore, it can be inferred that the persistent mode should be deactivated when the delay deriving from instrumented memory monitors exceeds the time can be saved from the persistent mode.

The synchronizer in HNPFuzzer can reduce the time waste due to manually-configured time delays in $poll()$. The experiment results in the second column show that the synchronizer gains a higher average throughput (+127.48%) as well. However, some services with the sole synchronizer experience performance degradation. There are two potential reasons for this problem. On the one hand, the manually-configured time does not introduce much time waste in the first place and the synchronizer does not save time but increases overhead due to redundant notification mechanism. On the other hand, the synchronizer itself cannot detect some session state transition events without shared memory based IPC, which introduces extra overhead. For example, the event c.iii requires the server to check if there is any incoming message. This can be effortlessly

TABLE 3: Results of ablation experiments on ProFuzzBench

Service	Fuzzing Throughput (msg/s) improvement rate relative to AFLNET			
	Persistent mode only	Sync only	Shared memory IPC and sync	Together
Bftpd	+1.52%	-20.84%	+261.40%	**+398.64%**
Lightftp	+0.4%	-30.4%	+1896.56%	**+1606.84%**
Proftpd	-27.22%	-30.00%	+69.43%	**+9.67%**
Pureftpd	-13.20%	-50.07%	+295.58%	**+383.41%**
Dcmtk	-44.88%	-39.31%	+14.49%	**-37.18%**
Live555	-24.73%	-41.89%	+598.65%	**+646.17%**
Exim	+24.55%	+100.77%	+178.23%	**+303.99%**
Forked-daapd	+291.24%	+220.24%	+47.7%	**+1480.79%**
Openssh	+18.99%	+7.59%	+150.57%	**+242.75%**
Openssl	+64.67%	+265.98%	+946.77%	**+978.08%**
Dnsmasq	\	-16.39%	+1012.05%	**+1012.05%**
Kamailio	\	+6.57%	+274.54%	**+274.54%**
Tinydtls	\	+1284.93%	+44259.38%	**+44259.38%**
Average	**+29.13%**	**+127.48%**	**+3846.57%**	**+3966.09%**

achieved by examining the shared memory buffer content length if shared memory based IPC is enabled, while the synchronizer must employ $poll()$ to detect this event when it is the only component in effect.

The shared memory based IPC is the most powerful component in our system, which brings an average speedup of 3846.57% as shown in the third column in Table 3. It is worth mentioning that this improvement is achieved by cooperating with the synchronizer, without which the correspondence between requests and responses will become disordered, leading to further desynchonization in the state machine. Therefore, the effect of shared memory IPC cannot be measured when the synchronizer is deactivated. It can be inferred from Table 3 that shared memory IPC can accelerate all of the services. However, although *Forked-daapd* gains a higher throughput (+47.7%), it is lower than employing the synchronizer alone (+220.24%), which implies the potential adverse impact brought by the shared memory IPC. The reason for these negative performances can be attributed to the overhead deriving from the locks and signal masks which are introduced alone with shared memory IPC. Both of them are employed to guarantee the atomicity of shared memory read-write operations.

7.4 Code branch coverage

Code coverage measures the degree of the application has been tested. Generally, more vulnerabilities will be discovered if more code is executed. We select branch coverage to represent the code coverage considering that *aflclang* instruments the program in basic blocks. The code branch coverage is displayed in Table 4. It can be found that HNPFuzzer outperforms other three fuzzers on most services and increases the coverage rate by 6.53% in average.

Fig. 8 illustrates the number of code branches with the growing of fuzzing time in 24 hours. We can conclude that HNPFuzzer outperforms other fuzzers all the time in most network services. Moreover, the curve of HNPFuzzer is smoother, which indicates the appreciable number of code branches discovered by HNPFuzzer is less likely to rely on certain random generated test cases. However, similar to the throughput, HNPFuzzer also performs less well than AFLNET on *Dcmtk*. This is reasonable considering that HNPFuzzer does not improve the fuzzing throughput

TABLE 4: Code branch coverage of various fuzzers on ProFuzzBench

Service	Code Branch Coverage			
	AFLNET	AFLNWE	StateAFL	HNPFuzzer
Bftpd	485.40	483.00(-0.49%)	445.20(-8.28%)	**499.60(+2.93%)**
Lightftp	351.00	162.20(-53.79%)	266.00(-24.22%)	**355.60(+1.31%)**
Proftpd	5077.40	5419.60(+6.74%)	4907.00(-3.36%)	**5226.20(+2.93%)**
Pureftpd	1154.60	422.00(-63.45%)	975.60(-15.50%)	**1302.60(+12.82%)**
Dcmtk	3123.40	3115.60(-0.25%)	3013.80(-3.51%)	**3108.60(-0.47%)**
Live555	2801.60	2840.40(+1.38%)	2750.00(-1.84%)	**2846.80(+1.61%)**
Exim	2578.20	1083.40(-57.98%)	4039.80(+56.69%)	**3017.60(+17.04%)**
Forked-daapd	2218.20	1132.80(-48.93%)	2118.40(-4.50%)	**2493.20(+12.40%)**
Openssh	3366.20	3170.00(-5.83%)	3169.00(-5.86%)	**3404.60(+1.14%)**
Openssl	10079.40	10105.80(+0.26%)	9950.60(-1.28%)	**10261.60(+1.81%)**
Dnsmasq	1220.00	804.00(-34.10%)	1164.00(-4.59%)	**1224.00(+0.33%)**
Kamailio	9744.60	8253.40(-15.30%)	9360.40(-3.94%)	**10181.80(+4.49%)**
Tinydtls	509.80	376.60(-26.13%)	538.40(+5.61%)	**645.40(+26.60%)**
Average		**-22.91%**	**-1.12%**	**+6.53%**

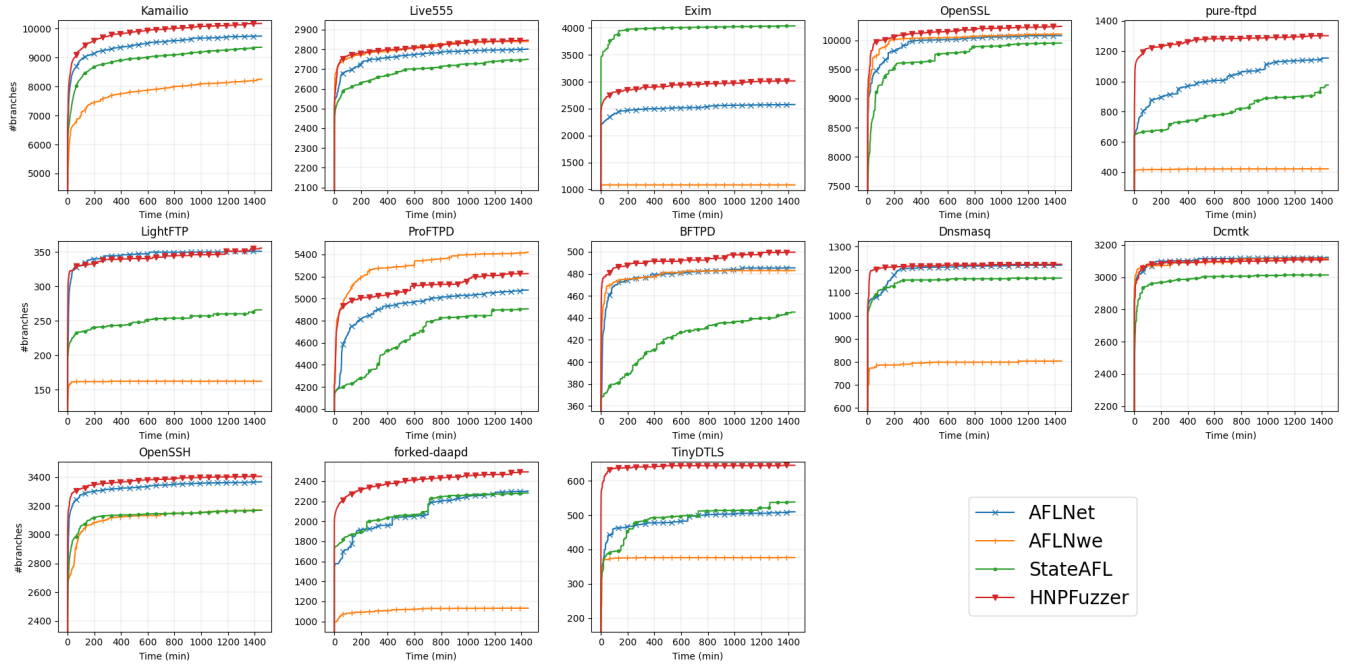


Fig. 8: Number of code branches achieved by each fuzzer within 24 hours.

of *Dcmtk*. In this case, the sophisticated mechanisms in HNPFuzzer may have a bad effect on fuzzing efficiency.

It can be also observed from Fig. 8 that quite a large number of branches are found in the initial fuzzing phases. This is reasonable considering that the superficial paths can be always found easily and it becomes harder and harder to find the deep-seated paths with the execution of fuzzing process. Moreover, to reduce the uncertainty of experiments, the simulation results are presented in average. However, an interesting observation of Fig. 8 is that there are still some small sudden changes in the simulation results of some services, such as *Openssl*, *Forked-daapd* and *Tinydtls*. In our opinion, the small changes are acceptable considering that it is possible that a set of new branches are found in a small set of runs.

7.5 Vulnerability discovery

To evaluate the ability of different fuzzers in vulnerability digging, we replayed all crash testcases to servers which were in their original versions but with ASAN enabled and dropped those cannot reproduce the crash. Therefore, the influence of false positive crashes was eliminated via this step. Then we manually analyzed the information provided by ASAN to deduplicate crashes. Consistent with our predictions, HNPFuzzer triggered more unique crashes on the basis of AFLNET. Table 5 illustrates the number of deduplicated crashes triggered by four different fuzzers within 24 hours. It can be observed that HNPFuzzer triggers the most crashes for most protocols. Moreover, we find that HNPFuzzer cannot only trigger more crashes in *Dnsmasq* and *Tinydtls*, but also have capability of triggering crashes in *Forked-daapd* which never crashes under other fuzzers testing. Overall, HNPFuzzer is of the best performance of all the four protocol fuzzers in terms of vulnerability discovery.

We further analyze the crashes and find that HNPFuzzer is able to reveal the same bugs as AFLNET did in *Live555* and *Dnsmasq*. For *Tinydtls*, HNPFuzzer found a new Denial of Service (DoS) vulnerability which is not discovered by other fuzzers. As Fig. 9 shows, the bug is triggered in line 2854 of the file *dtls.c*. Once the server receives a certificate request, it crashes when the cipher suite of peer does not match TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 (line 2854). This logic can only be triggered when state is DTLS_STATE_WAIT_SERVERHELLODONE (line 3262), in other words, the fourth flight. This vulnerability has also been manually verified by a *Tinydtls* user [41]. We analyze other crashes triggered in *Tinydtls* and find most of them occur in the process of creating the cookie, which takes place in the second flight. In conclusion, HNPFuzzer is more likely to reach deeper states to discover new vulnerabilities with the help of its high throughput.

The crashes found in *Forked-daapd* are all caused by SIGKILL due to out of memory (OOM). The reason of OOM is the endless $mmap()$ invocation loop caused by the fuzzer generated malformed SMARTPL queries. This OOM vulnerability has been manually confirmed and mitigated by the researchers in [42]. Besides, we observe that the huge delay from SIGTERM receiving to the server termination should be to blame for the low efficiencies of other fuzzers on *Forked-daapd*. Therefore, the persistent mode of HNPFuzzer that avoids sending SIGTERM should be credited with the discovery of this bug.

TABLE 5: Deduplicated crashes triggered by various fuzzers

Service	Deduplicated Crashes			
	AFLNET	AFLNWE	StateAFL	HNPFuzzer
Live555	4	4	4	**4**
Forked-daapd	0	0	0	**1**
Dnsmasq	5	8	3	**9**
Tinydtls	3	3	1	**4**

```
./dtls.c
2838  static int
2839  check_certificate_request(dtls_context_t *ctx,
2840       dtls_peer_t *peer,
2841       uint8_t *data, size_t data_length)
...
2852   update_hs_hash(peer, data, data_length);
2853
2854   assert(is_tls_ecdhe_ecdsa_with_aes_128_ccm_8(
        peer->handshake_params->cipher));
...
3118  static int
3119  handle_handshake_msg(dtls_context_t *ctx, dtls_peer_t *peer,
      session_t *session,
3120  const dtls_peer_type role, const dtls_state_t state,
3121       uint8_t *data, size_t data_length)
...
3260   case DTLS_HT_CERTIFICATE_REQUEST:
3261
3262     if (state != DTLS_STATE_WAIT_SERVERHELLODONE) {
3263       return dtls_alert_fatal_create(DTLS_ALERT_UNEXPECTED_MESSAGE);
3264     }
3265
3266     err = check_certificate_request(ctx, peer, data, data_length);
```

Fig. 9: The DoS vulnerability exposed by HNPFuzzer in *Tinydtls*.

8 Discussion

Based on the experiments in Section 7, we have proved that HNPFuzzer outperforms the baseline fuzzers, i.e., AFLNET, AFLNWE and StateAFL, in terms of fuzzing throughput, code branch coverage and vulnerability discovery. However, there are still some other important properties of HNPFuzzer need to be illustrated. We first provide a comparison between HNPFuzzer and existing distinguished fuzzers. Then, the possibility and challenges of integrating HNPFuzzer and existing fuzzers will be discussed.

8.1 Comparison with other network protocol fuzzers

Though a set of network protocol fuzzers with high performance have been proposed since AFLNET, it is unwise to directly quantitatively compare all the fuzzers in a particularly scenario. On the contrary, most designed schemes in the literatures are compared with the baseline fuzzers and this paper also follows the convention. This is reasonable considering that the existing fuzzers are of totally different underlying principals. We take NSFuzz, Nyx-Net and SnapFuzz as examples to illustrate the differences to HNPFuzzer.

8.1.1 NSFuzz

NSFuzz improves the performance through building a more fine-grained protocol state machine as well as accelerating the fuzzing based on a new synchronization mechanism. The former is achieved by extracting state variables to represent protocol states, which shares no similarities with our work. While the latter holds a similar idea to our synchronizer, i.e., avoid the manually set waiting time in synchronization by providing a notification mechanism between the server and the client. In detail, NSFuzz lets the server raise a SIGSTOP before it starts $recv()$ so that the forkserver can return from $waitpid()$ and notify the fuzzer through pipe to send a message. Different from the server-forkserver-fuzzer pattern, our synchronization mechanism allows the server to notify the fuzzer directly through shared memory. Therefore, we do not need to identify the network event loop which may require manual work in NSFuzz, but we has to ensure any access to the shared memory areas is atomic. In other words, our synchronizer is more automated but introduces extra overhead. Besides, it is important to note that our synchronizer can not only accelerate the fuzzing, but is also an indispensable component for shared memory based IPC, which is a totally different function from that of NSFuzz synchronization mechanism.

8.1.2 Nyx-Net

Nyx-Net is one of the most advanced network protocol fuzzers that are quite different from AFL based fuzzers. Its main idea is to cut the heavy cost derived from network traffic handling through hypervisor-based snapshot and network behaviors emulation. The second solution shares a similar essential idea with our shared memory based IPC, i.e, avoiding the heavy overhead of socket I/O. Nyx-Net achieves this by running a bytecode virtual machine to directly generate data in the server process. The bytecodes are obtained from the fuzzer through hypervisor.

On the other hand, HNPFuzzer is builded on the top of AFLNET. It mutates messages and sends them to the server through shared memory with the help of the synchronizer, which is also located in shared memory. It does not suffer from the limitations of generative fuzzing, such as requiring manipulates and lacking of context. Though Nyx-Net mitigates these limitations by adopting a customized VM instead of providing manipulates and obtaining traces of the communication during network functionality emulation, it cannot completely eliminate the influence. This might be the reason why Nyx-Net gained lower improvement rates on average branch coverage in our replications in 5 targets (i.e., +1.36% in *Bftpd*, +0.87% in *Live555*, +4.37% in *Forked-daapd*, -8.56% in *Dnsmasq*, +17.22% in *Tinydtls*) than HNPFuzzer (i.e., +2.93% in *Bftpd*, +1.61% in *Live555*, +12.4% in *Forked-daapd*, +0.33% in *Dnsmasq*, +26.6% in *Tinydtls*).

8.1.3 SnapFuzz

SnapFuzz is another advanced network protocol fuzzer. It integrates numerous components to accelerate the fuzzing, and most of them are not in conflict with HNPFuzzer. The only two components intersecting with our fuzzer are SnapFuzz protocol and Smart Deferred Forkserver. The SnapFuzz protocol is adopted to eliminate the need for custom delays in synchronization and replace Internet sockets with UNIX domain sockets, which shares a similar idea with our shared memory based synchronizer and IPC. In relation to the Smart Deferred Forkserver, which is introduced to skip the initialization in server, shares a same essential idea with the persistent mode in HNPFuzzer.

According to the results of our ablation experiment and replications of SnapFuzz, we compile Table 6, which suggests the shared memory based synchronizer and IPC gained a higher average throughput improvement rate (+2096.47%) than the SnapFuzz protocol component (+707.31%). Moreover, the ablation experiment showed that the persistent mode in HNPFuzzer also make a great contribution to speeding up *Forked-daapd* fuzzing (+47.7% when disabled, +1480.79% when enabled), which was not tested by SnapFuzz. According to our analysis, this speedup is not only correlated with initialization elimination, but is also primarily achieved via reducing the time consumption of the server termination. As the high latency of SIGTERM handling is probably the most time-consuming part in "other" behaviors in *Forked-daapd*, the persistent mode can eliminate this and provide a speedup. In conclusion, we can conclude the persistent mode has the potential to further accelerate the fuzzing when terminating the server via signal is slower than doing the cleanup after the connection.

TABLE 6: Improvement rate of fuzzing throughput (exec/s) relative to AFLNET

Service	Fuzzing Throughput (exec/s)	
	SnapFuzz-protocol	HNPFuzzer-shmIPC
Dcmtk	-52.75%	+17.30%
Live555	+399.35%	+400.08%
LightFTP	+75.71%	+924.52%
Dnsmasq	+533.72%	+554.97%
Tinydtls	+2580.51%	+8585.50%
Average	+707.31%	+2096.47%

8.2 Further enhancement of HNPFuzzer

HNPFuzzer improves fuzzing throughput mainly by decreasing the time consuming of IPC and synchronization. An interesting attempt is integrating the snapshot technique into HNPFuzzer. In network protocol fuzzing, the distinguished effect of snapshot has been fully proved in SNPSFuzzer [16] and Nyx-Net [32]. Considering the different underlying principles, we believe that snapshot is a promising technique to further boost the speed of our system, though there are some challenges to combine it with our implementation.

First, the persistent mode (process recycling) in our scheme cannot be employed in conjunction with snapshot at the same time, because their functions are duplicated (i.e., our persistent mode skips initializations, snapshot skips initializations as well as common prefix messages). However, it makes sense that we can combine them and enable the faster one adaptively based on the time consumption obtained during fuzzing. In real life, we need to dynamically determine which strategy is faster in different situations, because they are of different efficiency bottlenecks (e.g., the size of snapshot affects the restoration speed, the instrumentations affect the persistent mode speed).

Second, different snapshot strategies has different influences on the synchronization mechanism in our scheme and we discuss it in the following. If the snapshot is only used to eliminate the initialization (root snapshot in [32]), then it does not affect the synchronization as the location of snapshot is ahead of message exchanges. However, it is possible that the snapshots taken in the middle of message exchange processes like [16] and incremental snapshots in [32]. In this case, the fuzzer needs to set up the synchronizer as well as the connection controller ahead of snapshot restoration to make sure the restored server runs smoothly. For example, if the server is going to be restored to the state right before receiving a message M_x, the fuzzer should run Algorithm 2 with M_x and its size as inputs. The extra steps are that the fuzzer should set the session state to S as well as restore the server between line 9 and line 10 in Algorithm 2.

9 Conclusion

In this paper, we designed a high-speed network protocol fuzzing approach named HNPFuzzer by shortening the time consumption of IPC and time delays caused by synchronization and initialization process. A sophisticated connection controller is designed and implemented based on shared memory to manage the high-speed information transmission between CUC and SUT. To manage the synchronization for CUC and SUT, we implement a practical strategy to arrange the messages in order and allows both sides know the state of each other. This manner is much more efficient than manually setting the timeout before sending and receiving messages.

Moreover, to thoroughly use the program environment constructed by a series of initialization and preparation behaviors, we add the persistent mode to HNPFuzzer. It determines whether to exit or just stop the server according to the memory state in order to save time in initialization. In our experiments, we also noticed that the step to kill the SUT is slow in certain services, e.g., *Forked-daapd*. This is because the signal handlers can also be very time consuming. In this case, the efficiency of HNPFuzzer is further improved as the SUT is terminated by itself rather than by SIGTERM from the CUC.

Simulation results illustrate that HNPFuzzer has a good performance in terms of both fuzzing efficiency and vulnerabilities discovery compared with the baseline network protocol fuzzers. As for the future work, an interesting and promising direction is combining our scheme with the snapshot based network protocol fuzzing schemes such as SNPSFuzzer and Nyx-Net. Intuitively, they can further improve the fuzzing efficiency in a collaborative manner.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China (2022YFB2702700), National Natural Science Foundation of China under Grants 62102017, 62001055 and in part by the Fundamental Research Funds for the Central Universities under Grant YWF-22-L-1273.

References

[1] M. Boehme, C. Cadar, and A. ROYCHOUDHURY, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021.

[2] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[3] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1613–1627, 2020.

[4] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie : Fuzzing windows applications with harness synthesis and fast cloning," in *Network and Distributed System Security Symposium*, 2021.

[5] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *USENIX Security Symposium*, 2021, pp. 2597–2614.

[6] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 659–676, 2021.

[7] M. Zalewski. American Fuzzy Lop. Accessed: Feb. 2023. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020, pp. 10–10.

[9] Lafintel. Circumventing fuzzing roadblocks with compiler transformations. Accessed: Feb. 2023. [Online]. Available: https://lafintel.wordpress.com/

[10] L. Project. libfuzzer. Accessed: Feb. 2023. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[11] I. Google. Oss-fuzz: Continuous fuzzing for open source software. Accessed: Feb. 2023. [Online]. Available: https://github.com/google/oss-fuzz

[12] K. Serebryany, "Oss-fuzz-google's continuous fuzzing service for open source software," in *USENIX Security symposium*. USENIX Association, 2017.

[13] D. Korczynski and A. Korczynski. Fuzzing 100+ open source projects with oss-fuzz - lessons learned. Accessed: Feb. 2023. [Online]. Available: https://adalogics.com/blog/fuzzing-100-open-source-projects-with-oss-fuzz

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

[15] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 337–350.

[16] J. Li, S. Li, G. Sun, T. Chen, and H. Yu, "Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 2673–2687, 2022.

[17] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, pp. 1–31, 2021.

[18] P. F. Platform. Gitlab.org / security-products / protocol-fuzzer-ce · gitlab. Accessed: Feb. 2023. [Online]. Available: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce

[19] J. Pereyda. boofuzz: A fork and successor of the sulley fuzzing framework. Accessed: Feb. 2023. [Online]. Available: https://github.com/jtpereyda/boofuzz

[20] K. Yan, B. Yu, Y. Tang, X. Kong, C. Chen, and J. Lei, "Nafuzzer: Augmenting network protocol fuzzers via automatic template and seed generation," *2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*, pp. 391–398, 2022.

[21] C. Park, S. Bae, B. Oh, J. Lee, E. Lee, I. Yun, and Y. Kim, "Doltest: In-depth downlink negative testing framework for lte devices," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1325–1342.

[22] Y. Chen, Y. Yao, X. Wang, D. Xu, C. Yue, X. Liu, K. Chen, H. Tang, and B. Liu, "Bookworm game: Automatic discovery of lte vulnerabilities through documentation analysis," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1197–1214.

[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] F. Zuo, Z. Luo, J. Yu, Z. Liu, and Y. Jiang, "Pavfuzz: State-sensitive fuzz testing of protocols in autonomous vehicles," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 823–828.

[25] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System Call-Based State-Aware linux driver fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022, pp. 3273–3289.

[26] S. Qin, F. Hu, B. Zhao, T. Yin, and C. Zhang, "Nsfuzz: Towards efficient and state-aware network service fuzzing," in *process of 1st International Fuzzing Workshop*, 2022, pp. 1–9.

[27] 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)*, 2020, pp. 1597–1612.

[28] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 720–730.

[29] L. Cui, B. Li, Y. Zhang, and J. Li, "Hotsnap: A hot distributed snapshot system for virtual machine cluster." in *LISA*, 2013, pp. 59–74.

[30] L. Cui, Z. Hao, L. Li, and X. Yun, "Snapfiner: A page-aware snapshot system for virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2613–2626, 2018.

[31] B. K. Raju and G. Geethakumari, "Snaps: Towards building snapshot based provenance system for virtual machines in the cloud environment," *Comput. Secur.*, vol. 86, no. C, p. 92–111, sep 2019.

[32] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 166–180.

[33] Criu.org. Comparison to other cr projects. Accessed: Feb. 2023. [Online]. Available: https://criu.org/Comparison_to_other_CR_projects

[34] Y. Zeng, M. Lin, S. Guo, Y. Shen, T. Cui, T. Wu, Q. Zheng, and Q. Wang, "Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols," *Sensors*, vol. 20, no. 18, p. 5194, Sep 2020.

[35] A. Andronidis and C. Cadar, "Snapfuzz: High-throughput fuzzing of network applications," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 340–351.

[36] Y. Shoshitaishvili. Preeny. Accessed: Feb. 2023. [Online]. Available: https://github.com/zardus/preeny

[37] P. Goldsborough. ipc-bench. Accessed: Feb. 2023. [Online]. Available: https://github.com/goldsborough/ipc-bench

[38] B. K. et al. Openssl. Accessed: Jun. 2023. [Online]. Available: https://github.com/openssl/openssl/tree/0437435a960123be1ced766d18d715f939698345

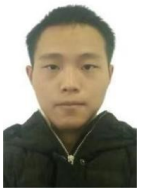
[39] V.-T. Pham. Aflnwe. Accessed: Feb. 2023. [Online]. Available: https://github.com/aflnet/aflnwe

[40] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 662–665.

[41] jerrytesting. An assertion failure. Accessed: Feb. 2023. [Online]. Available: https://github.com/contiki-ng/tinydtls/issues/26

[42] whatdoineed2do. Add memorymax to service. Accessed: Feb. 2023. [Online]. Available: https://github.com/owntone/owntone-server/pull/700

Junsong Fu received his Ph.D. degree in communication and information system from Beijing Jiaotong University in 2018. He is now serving as an associate professor in the school of cyberspace security in Beijing University of Posts and Telecommunications. His research interests include network security and information privacy issues in the Inter-net, mobile networks and Internet of Things.



Shuai Xiong received his Bachelor degree from the School of Cyberspace Security at Beijing University of Posts and Telecommunications (BUPT) in 2022. He is now pursuing the Master degree in School of Cyberspace Security at BUPT. His research interests include the security issues in terms of network protocol, software and mobile networks.

Na Wang received her Ph.D. degree in the School of Mathematical Sciences from Xiamen University in 2018. She is an assistant professor and doctoral supervisor in the School of Cyber Science and Technology, Beihang University. Her research interests include cryptography, message sharing, and information security issues in distributed and cloud systems.

Ruiping Ren received his Bachelor degree from the School of Cyberspace Security at Beijing University of Posts and Telecommunications (BUPT) in 2022. He is now pursuing the Master degree in School of Cyberspace Security at BUPT. His research interests include the security issues in terms of mobile communications and networks protocol.

Ang Zhou is currently pursuing the Bachelor degree at Bei-jing University of Posts and Telecommunications,and will continue to pursue a postgraduate degree in Beijing Univer-sity of Posts and Telecommunications.His main research directions include program reverse analysis, firmware secu-rity, mobile terminal security, mobile Internet security and program analysis.

Bharat K. Bhargava is a Professor of Computer Science at Purdue University. Prof. Bhargava is the founder of the IEEE Symposium on Reliable and Distributed Systems, IEEE conference on Digital Library, and the ACM Conference on Information and Knowledge Management. He is the editor-in-chief of four journals and serves on over ten editorial boards of international journals. Prof. Bhargava has pub-lished hundreds of research papers and has won five best paper awards in addition to the technical achievement award and golden core award from IEEE. He is a life fellow of IEEE.