HSPFuzzer: High-Speed Network Protocol Fuzzing with Connection Reuse

Zhewei Xia, Yingpei Zeng, Xiangpu Song, Shanqing Guo, and Ting Wu

***Abstract*—Fuzzing is a fundamental technique for detecting vulnerabilities in network protocols. However, existing approaches suffer from low fuzzing throughput caused by the overhead associated with server under test (SUT) restarts and connection setup. In this paper, we present HSPFuzzer, a High-Speed Protocol Fuzzer that leverages connection reuse to reduce SUT restarts and connection re-establishments. To enable efficient connection reuse, it incorporates a prefix message identification algorithm to determine the essential packets required within a connection and a coverage monitoring mechanism to detect abnormal execution states. Additionally, HSPFuzzer employs an innovative message provision method that ensures input messages are delivered to the SUT with minimal delay within the same connection. HSPFuzzer also eliminates the need for manually implementing message-splitting logic by connection reuse. We evaluate HSPFuzzer on 12 widely used servers and experimental results show that HSPFuzzer achieves fuzzing throughput 1062× faster than AFLNet, whereas other state-of-the-art fuzzers, including AFLNet, SnapFuzz, HNPFuzzer, and AFL++, achieve, at most, a 12× speedup over AFLNet. Furthermore, HSPFuzzer attains an average code coverage increase of 25.1% compared to AFLNet, while competing fuzzers achieve, at most, 2.13% more coverage. Notably, HSPFuzzer also discovers more vulnerabilities, which further proves its effectiveness.**

***Index Terms*—Fuzzing, Network protocol, Connection reuse, Vulnerability detection**

I. Introduction

NETWORK security has become increasingly critical in both the Internet and the Internet of Things (IoT). The growing complexity of network protocols has introduced significant security challenges for protocol servers, and new attack techniques continue to emerge, such as Heartbleed [1], EternalBlue [2], and regreSSHion [3]. Traditional approaches, including manual code review and security analysis, have become insufficient in addressing these threats. As a result, fuzz testing, especially grey-box fuzzing, has emerged as the preferred method for security assessment due to its low cost and high efficiency [4], [5], [6], [7], [8], [9].

Since mainstream fuzzers like AFL (American Fuzzy Lop) are primarily designed for programs that process inputs from files or the console [4], several specialized fuzzers have been developed to target network protocols [10], [11], [12], [13], [14], [15], [16], [17]. These fuzzers can identify messages in fuzzing inputs and transmit them to the server under test (SUT) via network sockets. For instance, AFLNet [10] has demonstrated superior performance over AFLNwe [10] and BooFuzz [18] in terms of code coverage and the number of detected vulnerabilities when fuzzing servers such as LightFTP and Live555. SnapFuzz [12] enhances AFLNet by introducing a smart deferred forkserver, while HNPFuzzer [17] further optimizes AFLNet by leveraging shared memory and persistent mode.

TABLE I
THE FUZZING THROUGHPUT (EXECS/S) AND REQUIREMENT OF FUZZERS.

	AFLNET	HNPFuzzer	HSPFuzzer (ours)
Dnsmasq	6.82	32.55	10359.15
LightFTP	5.12	31.43	12058.14 [1]
Live555	12.16	14.07	824.40
Need Coding	Yes	Yes	No
Need Source Code	No	Yes	No

[1] HSPFuzzer uses different (split) seeds in LightFTP.

However, existing network protocol fuzzers exhibit low fuzzing throughput, limiting their ability to effectively explore the SUT (Server Under Test). For example, when fuzzing Dnsmasq, LightFTP, and Live555, AFLNet [10] achieves a throughput of only 5∼12 executions per second as shown in Table I. The latest fuzzer, HNPFuzzer [17], improves throughput by up to six times but remains constrained to approximately 30 executions per second. This throughput is significantly lower than that of fuzzing conventional programs. For example, based on our experience, AFL++ can reach over 10,000 executions per second when fuzzing over half of the 23 FuzzBench programs [19]. In addition, AFLNet and all its variations require users to manually add code to split messages within inputs [10], and some may require access to the SUT's source code [15], [16], [17].

Manuscript received March xx, 2025; revised xxxx. The work of Y. Zeng was supported in part by the Zhejiang Provincial Natural Science Foundation of China (No. LY22F020022), in part by the National Natural Science Foundation of China (No. 61902098). The work of S. Guo was supported by Major Scientific and Technological Innovation Projects of Shandong Province, China (No. 2024CXGC010114), National Natural Science Foundation of China (No. 62372268), Shandong Provincial Natural Science Foundation, China (No. ZR2022LZH013, No. ZR2021LZH007), Jinan City "20 New Universities" Funding Project (2021GXRC084). The work of T. Wu was supported in part by the Key Research Project of Zhejiang Province, China (No. 2024C01212). *(Corresponding author: Yingpei Zeng and Shanqing Guo.)*

Zhewei Xia is with the Zhuoyue Honors College and School of Cyberspace, Hangzhou Dianzi University, Hangzhou 310000, China (e-mail: xiazwie@hdu.edu.cn)

Yingpei Zeng is with School of Cyberspace and Zhejiang Provincial Key Laboratory of Sensitive Data Security and Confidentiality Governance, Hangzhou Dianzi University, Hangzhou 310000, China (email: yzeng@hdu.edu.cn).

Xiangpu Song and Shanqing Guo are with School of Cyber Science and Technology, Shandong University, Jinan 250000, China (email: songxiangpu@mail.sdu.edu.cn, guoshanqing@sdu.edu.cn).

Ting Wu is with Hangzhou Innovation Institute, Beihang University, Hangzhou 310000, China (email: wutingbh@163.com).

The low fuzzing throughput of network protocols is primarily attributed to the significant overhead involved in providing test inputs to the SUT, including server initialization, connection establishment, and message transmission [10]. While recent fuzzers [12], [17] have made progress in reducing the overhead associated with server initialization and message transmission, they largely overlook the cost of connection establishment, which, according to our measurements, can be as high as 779.45% of the actual SUT processing time. However, we argue that reconnecting to the SUT for every input is unnecessary. For instance, in the case of an FTP (File Transfer Protocol) server, once authentication is successfully completed, subsequent commands such as `LIST`, `PWD`, and `STAT` can be executed within the same connection without requiring re-establishment [20].

In this paper, we present HSPFuzzer, a high-speed protocol fuzzer based on the novel concept of connection reuse. Connection reuse enables multiple fuzzing inputs to be transmitted within the same connection to the SUT, thereby eliminating the unnecessary overhead associated with server restarts and connection establishment. This approach can be considered a more advanced form of persistent mode, which persists the connection rather than merely persisting the SUT [12], [17]. Moreover, connection reuse does not introduce false positives in vulnerability detection, as real-world attackers can also reuse connections. Connection reuse eliminates the need for writing code for message splitting as well with special seed preparation. To enable effective connection reuse, HSPFuzzer integrates a prefix message identification algorithm to determine and transmit the necessary messages for establishing a new connection, along with a coverage monitoring mechanism to detect abnormal execution states. Additionally, HSPFuzzer employs a message provision method to timely deliver messages to the SUT while discarding unnecessary reply messages, further optimizing fuzzing efficiency.

We implement a prototype of HSPFuzzer and compare it against state-of-the-art fuzzers, including AFLNet, AFLNwe, SnapFuzz, HNPFuzzer, and AFL++ (with preeny/desock), across 12 widely used servers. Experimental results demonstrate that HSPFuzzer achieves a 1062.06× speedup over AFLNet, whereas AFLNwe, SnapFuzz, HNPFuzzer, and AFL++ only achieve 1.68×, 7.82×, 12.13×, and 4.06× improvements over AFLNet, respectively. Furthermore, HSPFuzzer enhances AFLNet's code coverage by 25.10%, while the other fuzzers exhibit improvements of -12.57%, 0.58%, 2.13%, and -32.46%, respectively. Additional experiments confirm that connections are effectively reused for multiple inputs in SUTs. Interestingly, we find that connection reuse not only preserves the fuzzing capability of inputs but may even enhance it. Moreover, our evaluation validates the effectiveness of prefix message identification and coverage monitoring. During testing, HSPFuzzer successfully identified 14 vulnerabilities, surpassing the second-best fuzzer by four additional findings. The prototype of HSPFuzzer will be open-sourced for research after paper publication at https://github.com/hspfuzzer.

The remainder of this paper is structured as follows. Section II discusses the limitations of existing approaches and provides an overview of our methodology. Section III details the design and implementation of HSPFuzzer. In Section IV, we present and analyze our experimental results. Section V further examines our approach and its limitations. Section VI reviews related work. Finally, we conclude in Section VII.

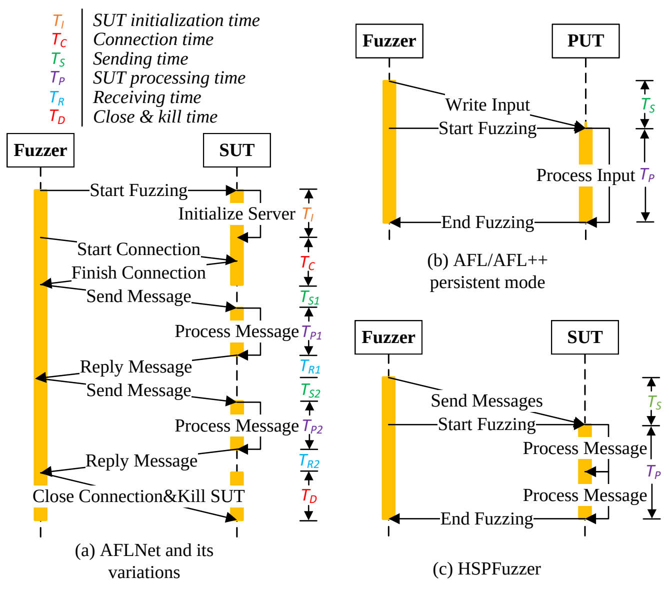

Fig. 1. The breakdown of fuzzing time for processing a single input across different fuzzers (assuming the input contains two messages).

II. Problem and Approach Overview

A. Network protocol fuzzing

Network protocol implementations (i.e., servers) exhibit distinct interaction patterns compared to conventional (ordinary) programs or libraries, such as tcpdump and libpng [7]. First, servers receive inputs via network sockets rather than from files or the console, as is typical for ordinary programs and libraries. Second, servers are designed to run continuously and often undergo an initialization phase to process configurations upon startup. In contrast, ordinary programs and libraries generally execute for a short duration and may not require such initialization. Third, servers can accept multiple new network connections and process multiple messages within each connection before termination, whereas ordinary programs and libraries typically handle a single input before exiting.

These distinctions significantly impact the fuzzing of servers compared to ordinary programs. Fig. 1 (a) illustrates the typical execution flow of existing network fuzzers such as AFLNet [10] and its variations [11], [12], [13], [14], [15], [17] when processing an input. The process begins with the forking of an SUT instance from the forkserver, followed by its initialization, which takes time T_I (subsequent time representations follow the same notation). Next, a network connection is established between the fuzzer and the SUT (T_C). If the input contains multiple messages, the fuzzer sequentially extracts each message, transmits it to the SUT (T_{Si}), waits for the SUT to process the message (T_{Pi}), and receives the response (T_{Ri}), extracting relevant response codes to construct a state machine. Once all messages have been processed, the fuzzer closes the connection and terminates the SUT process (T_D). Notably, the connection closure and SUT termination often occur simultaneously, making them indistinguishable in practice.

B. Dissection of fuzzing time

TABLE II
THE BREAKDOWN OF FUZZING TIME WHEN AFLNET AND HNPFUZZER PROCESS A SINGLE SEED IN FOUR SERVERS (IN MICROSECONDS).

Subject	Fuzzer	T_I (Init. SUT)	T_C (Conn. Setup)	T_S (Send)	T_R (Reply)	T_P (SUT Process)	T_D (Close & Kill)
LightFTP	HNPFuzzer	15,678	873	35	27	112	1,573
	AFLNET	19,409	1,945	305	183	109	734
Live555	HNPFuzzer	2,400	225	56	29	783	623
	AFLNET	9,247	1,162	1,653	1,754	691	192
OpenSSL	HNPFuzzer	84,906	528	45	15	12,056	6,939
	AFLNET	70,667	10,381	718	333	11,436	5,893
DCMTK	HNPFuzzer	81,956	244	62	11	3,158	21,100
	AFLNET	45,045	1,176	567	341	2,758	1,940

To further investigate the overhead incurred by AFLNet when processing an input, we analyze both AFLNet and the most recent fuzzer, HNPFuzzer, by measuring their execution times for handling a single seed, using the time components defined in the previous subsection. The results are presented in Table II. HNPFuzzer may restart the SUT for certain inputs, and in such cases, its initialization time (T_I) and termination time (T_D) are similar to those of AFLNet. Here, HNPFuzzer indeed restarts the SUTs for these seeds. However, for inputs where HNPFuzzer does not restart the SUT, T_I is eliminated entirely, and T_D is reduced to include only the overhead associated with closing the connection. HNPFuzzer also effectively reduces T_C by eliminating unnecessary waiting time and decreases both T_S and T_R through the use of shared memory for message transmission. However, T_P is slightly higher in HNPFuzzer due to additional instrumentation that monitors memory operations within the SUT. Notably, both fuzzers exhibit higher T_P values for OpenSSL because, when a seed passes the prerequisite checks, the subsequent cryptographic operations require substantial processing time. Conversely, for inputs that fail the prerequisite checks, T_P remains minimal (typically under 100 µs). Additionally, HNPFuzzer exhibits a slightly higher T_D due to its modified killing procedure, which includes closing the connection earlier.

The results indicate that maintaining a network connection constitutes a significant portion of the execution time even in HNPFuzzer. If the SUT is not restarted, T_I is eliminated, and T_D is reduced to only the connection termination overhead. However, the core processing time (T_P) is irreducible. Even with HNPFuzzer's optimized connection setup time (T_C), this overhead still accounts for 779.46% and 28.73% of the processing time (T_P) in LightFTP and Live555, respectively. A similar ratio is expected in OpenSSL and DCMTK (Digital Imaging and Communications in Medicine Toolkit, i.e., DICOM Toolkit) when processing fuzzing inputs that fail early validation checks. Additionally, connection termination introduces delays as well. For instance, closing a Transport Layer Security (TLS) connection requires sending a termination notification before deallocating resources. In comparison to the persistent mode of AFL [4] and AFL++ [21] for fuzzing ordinary programs under test (PUT) (as illustrated in Fig. 1 (b)), the connection overhead T_C and T_D is extra introduced. Furthermore, while message reception time (T_R) is reduced in HNPFuzzer, it remains a non-negligible overhead.

Other approaches have similar overheads. SnapFuzz and NSFuzz also restart the SUT for every input [10], [12], [15]. While some approaches mitigate this issue by using snapshot-based techniques to avoid full restarts [13], [14], the process of restoring snapshots itself introduces non-negligible latency, e.g., in the range of 10–20 ms [13]. EQUAFL [22] persists the SUT but reconnects the connection across multiple inputs similar to HNPFuzzer.

C. Core ideas and reasonability of HSPFuzzer

The core ideas of our approach are as follows. Given that network connection setup and teardown constitute a significant portion of fuzzing time, HSPFuzzer employs **connection reuse** to transmit multiple fuzzing inputs over a single connection. Additionally, HSPFuzzer introduces an optimized **message provision** mechanism that schedules message transmission within the connection to minimize delays and discards unnecessary response messages. As a result, HSPFuzzer achieves a fuzzing time breakdown for the SUT (as illustrated in Fig. 1 (c)) that closely resembles the persistent mode of AFL/AFL++ for ordinary programs.

We justify the reasonability of connection reuse, which is a core innovation, as follows.

First, a connection is typically not closed by the server after processing multiple messages in standard server implementations. We manually examine the source code of several network protocols, with results summarized in Table III. Our analysis identifies three types of messages: *prefix messages*, which must be sent at the beginning of a connection (e.g., for authentication); *ordinary messages*, which can be sent an unlimited number of times after authentication (e.g., for serving a client); and *quit messages*, which explicitly terminate the connection (e.g., for ending the session or handling severe errors). For instance, in FTP servers, once a client successfully authenticates using `USER` and `PASS` messages, subsequent commands such as `LIST`, `PWD`, `MKD`, and `STAT` can be transmitted within the same network connection and processed normally by the server [20], as illustrated by the code snippet of the Pure-FTPd server shown in Listing 1. The connection remains open unless a `QUIT` message is sent or a severe error occurs (e.g., an over-length message or an unexpected HTTP `GET` request), both of which immediately terminate the session. For many protocols, the quit messages occur infrequently during fuzzing, provided they are excluded from the seed inputs. We further validate this observation experimentally in Section IV.

Second, connection reuse does not introduce false positives in vulnerability detection. This is because server implementations generally do not impose restrictions on the number of messages a client can send within a single connection. If

TABLE III
THE MESSAGES THAT CAN BE SENT OVER A NETWORK CONNECTION IN DIFFERENT PROTOCOL IMPLEMENTATIONS. "{}" DENOTES CONCATENATION. "[]" INDICATES THE MESSAGES ARE OPTIONAL DUE TO SETTINGS. SERVERS ARE USING THEIR VERSIONS OR COMMIT IDS.

Protocol	Prefix Message	Ordinary Message	Quit Message	Server
FTP	{USER,PASS}	LIST,PWD,ABOR, ...	QUIT, over-length message	LightFTP v2.3.1, Pure-FTPd v1.0.52
			QUIT, messages of other protocols like GET, over-length messages	vsftpd v3.0.5, ProFTPD v1.3.8c
DTLS	{ClientHello, ClientHello[1]}	ClientKeyExchange, ChangeCipherSpec, Finished, ...	ALERT (fatal)[2], error messages (e.g., wrong fields)	TinyDTLS (8a9e048)
SIP	[REGISTER],[INVITE]	CREATE, JOIN, INVITE, ...	BYE, fatal error messages (i.e., wrong critical fields)	Kamailio v6.0.0
TLS	ClientHello	ClientKeyExchange, ChangeCipherSpec, ...	ALERT (fatal), error messages (e.g., wrong fields)	OpenSSL v3.4.0
SSH	{SSH_MSG_KEXINIT, SSH_MSG_KEX_ECDH_INIT, SSH_MSG_USERAUTH _REQUEST}, ...	SSH_MSG_CHANNEL_OPEN, SSH_MSG _CHANNEL_REQUEST, ...	SSH_MSG_DISCONNECT, unknown message types, error messages if in strict mode	OpenSSH v9_9_P1
DICOM	[A-ASSOCIATE-RQ]	C-STORE, C-FIND, ...	A-RELEASE-RQ, A-ABORT, fatal error messages (i.e., wrong critical fields)	DCMTK v3.6.9
RTSP	[DESCRIBE]	OPTIONS, SETUP, PLAY, ...	TEARDOWN, fatal error messages (i.e., wrong critical fields)	Live555 v2025.01.17
MQTT	CONNECT	PUBLISH, SUBSCRIBE, UNSUBSCRIBE, ...	DISCONNECT, fatal error messages (i.e., wrong critical fields)	Mosquitto v2.0.20
IPP	[WWW-Authenticate]	Get-Jobs, Create-Job, Send-Document, ...	fatal error messages (i.e., wrong critical fields)	ippsample (65a3759)
RESP	[AUTH]	GET, SET, LPOP, ...	QUIT, fatal error messages (i.e., wrong critical fields)	Redis v7.4.2
SMTP	{EHLO,AUTH}	MAIL, RCPT, DATA, ...	QUIT, too many fatal error messages	Exim v4.98

[1] The retransmission contains the cookie.
[2] Quit message causes the termination of the session but not the connection in UDP-based protocols.

```
} else if (loggedin == 0) {
  /* from this point , all commands need
       authentication */
  addreply_noformat(530, MSG_NOT_LOGGED_IN);
  goto wayout;
} else {
```

Listing 1. The code snippet in Pure-FTPd shows that the ordinary messages are processed after the user logs in.

a vulnerability can be triggered by sending multiple inputs (i.e., messages) within the same connection, an attacker could exploit the server using the same message sequence. It is important to note that this situation differs from ordinary programs or libraries, which typically process one input at a time. For example, the tcpdump program is commonly used to process a single pcap file per execution. If modified to process multiple inputs during fuzzing (e.g., using AFL persistent mode), discovered vulnerabilities may not manifest in real-world usage. It is also worth noting that HSPFuzzer does not attempt to maintain a clean environment for each input transmitted over the same connection. This is because we consider it unnecessary, as the inputs may not interfere with one another. Even if they do, such interference could potentially be beneficial for exploring diverse execution paths in certain cases (Section IV-D3).

Last but not least, connection reuse removes the need for additional code to split messages within inputs. Typically, implementing message splitting requires a non-trivial coding effort when using AFLNet and its variants [10], [12], [13], [15], [17], particularly for fuzzing servers that do not support processing packets containing multiple messages. For example, the LightFTP server terminates its `for` loop after matching a single command in the received packet `rcvbuf`, thereby ignoring any additional commands present in the same packet. As of March 2025, only 17 protocols are officially supported in the AFLNet repository [10]. While limiting each input to a single message eliminates the need for message-splitting logic, it also restricts the SUT to processing only one message per fuzzing execution. This limitation reduces the ability to discover vulnerabilities that require processing multiple messages within the same connection (e.g., CVE-2023-24042). However, with connection reuse, using single-message inputs avoids this problem without imposing constraints, as multiple inputs can be transmitted within a single connection. Additionally, AFLNet and its variants require extra code to parse reply messages and extract response codes, a task that can instead be handled by code-free state collection techniques such as StateAFL and SGFuzz [11], [16].

We provide a brief theoretical comparison as follows. Assuming that the time required to process each input is uniform, the time to process a single input in AFLNet can be represented as $(T_I+T_C+T_S+T_P+T_R+T_D)$ (Fig. 1 (a)). In contrast, assuming that HSPFuzzer restarts the PUT operation N times and reestablishes the connection M times when processing n inputs, the average time to process one input is given by $\frac{NT_I+MT_C+nT_S+nT_P+MT_D}{n} = \frac{N}{n}T_I + \frac{M}{n}(T_C+T_D) + T_S+T_P$. When connection reuse is effective, i.e., when N and M are small relative to n, this average time approximates (T_S+T_P) (Fig. 1 (c)).

D. Challenges and solutions

There are several challenges that must be addressed to ensure the effectiveness of connection reuse and message provision.

(C1) First, the SUT may not properly process messages received within a connection unless the correct prefix messages are first provided, and in some cases, it may close the connection or even enter an abnormal state. Additionally, the SUT may not support inputs containing multiple messages.

Our solution: HSPFuzzer employs an algorithm to automatically identify potential prefix messages from source pcap files and determine whether the SUT can process multiple messages within a single input. This ensures proper message preparation during seed generation. To monitor connection status, HSPFuzzer hooks functions responsible for closing connections and related operations to detect when the SUT attempts to close a connection. It also leverages coverage monitoring to identify cases where the connection does not behave as expected. When such anomalies are detected, HSPFuzzer proactively re-establishes a new connection to maintain stability.

(C2) Second, to minimize delays, the fuzzer must automatically track the message processing status of the SUT, enabling timely delivery of the next message or input as soon as processing of the current message or input is complete.

Our solution: HSPFuzzer introduces an optimized message provision mechanism by hooking the network I/O functions of the SUT, allowing it to monitor the SUT's activity and detect the completion of message processing using two conditions identified in our study. To maximize efficiency, message passing is also handled via shared memory, ensuring minimal latency. Whenever feasible, HSPFuzzer transmits all messages in a single batch to reduce turnaround time and discards unnecessary response messages to further optimize performance. While HNPFuzzer also hooks network I/O functions and utilizes shared memory, it lacks support for connection reuse and UDP-based protocols. Additionally, it employs a more complex synchronization method [17].

III. Design and implementation

A. Overview

Compared with conventional fuzzers such as AFL/AFL++, HSPFuzzer introduces key differences in both preprocessing and fuzzing, as illustrated in Fig. 2, where the differences are highlighted in yellow. During preprocessing, HSPFuzzer employs a prefix message recognition algorithm to analyze and classify messages into prefix messages and ordinary messages, based on whether they must be sent at the start of a connection.

During fuzzing, HSPFuzzer consists of two main components: the primary fuzzer and a fuzzer stub, the latter residing within the same process as the SUT. The primary fuzzer includes modules that enable connection reuse, in addition to the standard functionalities of a fuzzer. The connection reuse mechanism is primarily implemented within the *connection management* module, which maintains the network connection used during fuzzing, and the *coverage monitoring* module, which assesses the stability and validity of the connection. The fuzzing loop of the HSPFuzzer is shown in Algorithm 1, with modifications highlighted in grey [4], [5], [6].

The fuzzer stub is responsible for message provision, ensuring that fuzzing inputs (i.e., messages) are delivered to the SUT in a timely manner. This is achieved through the *network I/O hook* module, which intercepts the SUT's network I/O functions, and the *provisioning loop* module, which continuously supplies messages to the SUT. Synchronization between the primary fuzzer and the fuzzer stub is handled through pipes while fuzzing inputs and coverage information are transmitted through shared memory for greater throughput [17].

The following subsections provide a detailed explanation of these modules.

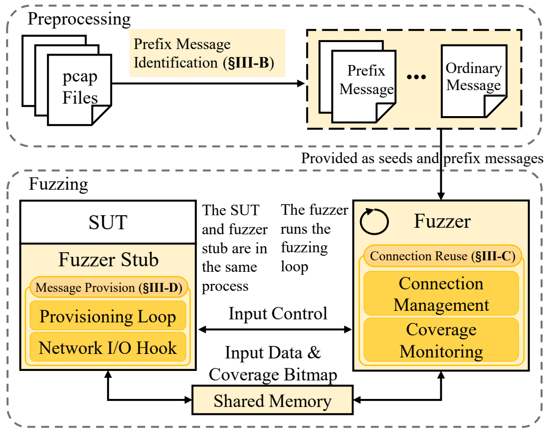
Preprocessing
pcap Files
Prefix Message Identification (§III-B)
Prefix Message
...
Ordinary Message
Provided as seeds and prefix messages
Fuzzing
SUT
Fuzzer Stub
Message Provision (§III-D)
Provisioning Loop
Network I/O Hook
The SUT and fuzzer stub are in the same process
The fuzzer runs the fuzzing loop
Fuzzer
Connection Reuse (§III-C)
Connection Management
Coverage Monitoring
Input Control
Input Data & Coverage Bitmap
Shared Memory

Fig. 2. Overall architecture of HSPFuzzer (differences to AFL++ highlighted in yellow).

Algorithm 1 The fuzzing loop.

```
Input: $Seeds$, $Prefixes$
Output: $Crashes$
 1: repeat
 2:     $s$ = ChooseNext($Seeds$)
 3:     $p$ = AssignEnergy($s$)
 4:     for $i$ from 1 to $p$ do
 5:         $s'$ = MutateInput($s$)
 6:         PrepareConnection($Prefixes$)
 7:         InputEvaluation($s'$)
 8:         if $s'$ crashes then
 9:             add $s'$ to $Crashes$
10:         else if IsInteresting($s'$) then
11:             add $s'$ to $Seeds$
12:         end if
13:         MonitorCoverage()
14:     end for
15: until timeout reached or user aborts
```

B. Prefix message identification

As explained in Section II-C, prefix messages may be required to establish a network connection to the SUT with specific configurations that allow the majority of messages to be processed. We formalize this concept in Definition III.1. In this definition, τ represents the scenario where messages such as quit messages are included, which are typically processed unconditionally (τ is usually a small value, e.g., 1).

Definition III.1 (Prefix Messages)**.** Prefix messages in a typical message sequence are those that must be present; otherwise, at most τ of the remaining messages can be processed.

HSPFuzzer introduces a prefix message recognition algorithm to automatically identify prefix messages from given pcap files and determine whether the SUT can process multiple messages within a single input. Each pcap file contains packets exchanged by a client and server during an interaction, which can be obtained by capturing network traffic between them [23], [10].

The algorithm is detailed in Algorithm 2. The core principle is that since prefix messages significantly impact the processing of subsequent messages, the latter cannot achieve the same code coverage in the SUT without them. To assess this, we use a SUT instance with identical startup settings and collect branch coverage using llvm-cov (other coverage metrics are also viable). The algorithm takes a pcap file as input and produces three outputs: the identified prefix messages ($PrefixMsgs$), and two boolean values ($NeedSplit$ and $AllSplit$) which indicate whether the SUT requires message splitting for at least the prefix messages and whether message splitting is necessary for non-prefix messages, respectively. If multiple pcap files are provided, the algorithm is executed separately for each.

The first phase identifies prefix messages. It first get the coverage of each message by sending all N messages from the pcap file sequentially (line 3). Assuming that $N \geq 2 + \tau$, the algorithm iterates through messages from M_0 to $M_{N-2-\tau}$ to determine whether each is a prefix message (lines 4-16). If M_x is a prefix message, the algorithm calculates after removing M_x the number of messages after M_x that still achieve similar coverage as before, i.e., within a threshold δ, accounting for factors such as network fluctuations or multithreading effects (lines 9-11). If this number exceeds τ, M_x is not a prefix message, and the process terminates (lines 13-15). Otherwise, the next message is analyzed in the same manner. Ultimately, all messages preceding the last non-prefix message are classified as prefix messages (line 17).

The second phase of the algorithm first determines $NeedSplit$ by comparing the coverage differences between two scenarios: (1) sending all N messages from the pcap file sequentially ($CovA$) and (2) concatenating all messages into a single input and sending them at once (lines 19-20). If the SUT can parse multiple messages within a single packet, the coverage values in both cases should differ only slightly, i.e., within $N\delta$, and no splitting is needed. Otherwise, at least prefix messages need to be split from other messages. Next, the algorithm determines $AllSplit$ using a similar approach by comparing $CovA$ with the coverage obtained from sending the prefix messages separately while concatenating the non-prefix messages. If the difference is significant, $AllSplit$ is set to `true`, indicating that the non-prefix messages also require splitting.

HSPFuzzer utilizes these results for seed preparation. The identified prefix messages are integrated into the network connection setup phase. Within the fuzzer, a dedicated prefix message queue is maintained alongside the original seed queue to facilitate efficient management. If $NeedSplit$ is set to `true`, the prefix messages must be explicitly provided. Otherwise, if the seeds already contain prefix messages, they may not need to be provided separately, as they will be sent to the SUT when the seeds are transmitted. If both $NeedSplit$ and $AllSplit$ are `true`, all messages are further separated into individual seeds. In rare cases where a pcap file contains quit messages during seed preparation, these messages are removed to ensure connection reuse.

Algorithm 2 Prefix Message Identification

```
Input: $PcapFile$
Output: $PrefixMsgs$, $NeedSplit$, $AllSplit$
 1: $Msgs \leftarrow PcapFile$ // Get a list of messages $M_0 \dots M_{N-1}$
 2: $N \leftarrow Msgs.\text{LENGTH}$ // Assume $N \geq 2 + \tau$
 3: $\{Cov(M_i) \mid i \in [0, N-1]\} \leftarrow \text{GETCOVERAGE}(Msgs)$
    // Determine PrefixMsgs
 4: for $x \leftarrow 0$ to $N - 2 - \tau$ do
 5:     $LeftM \leftarrow Msgs.\text{REMOVE}(Msgs.\text{GET}(x))$
 6:     $\{CovT(M_i) \mid i \in [0, N-1] \wedge i \neq x\} \leftarrow \text{GETCOVERAGE}(LeftM)$
 7:     $Num \leftarrow 0$
 8:     for $y \leftarrow x + 1$ to $N - 1$ do
 9:         if $\text{DIFF}(Cov(M_y), CovT(M_y)) \leq \delta$ then
10:             $Num \leftarrow Num + 1$
11:         end if
12:     end for
13:     if $Num > \tau$ then
14:         break
15:     end if
16: end for
17: $PrefixMsgs \leftarrow Msgs.\text{SUBLIST}(0, x)$
    // Determine NeedSplit and AllSplit
18: $NeedSplit \leftarrow$ false, $AllSplit \leftarrow$ false
19: $MsgsInOne \leftarrow \text{CONCATENATE}(Msgs)$
20: $CovO \leftarrow \text{GETCOVERAGE}(MsgsInOne)$
21: $CovA \leftarrow \bigcup_0^{N-1} Cov(M_i)$
22: if $\text{DIFF}(CovA, CovO) > N\delta$ then
23:     $NeedSplit \leftarrow$ true
24:     $LeftMsgsInOne \leftarrow \text{CONCATENATE}(Msgs.\text{SUBLIST}(x, N))$
25:     $CovOL \leftarrow \text{GETCOVERAGE}(PrefixMsgs, LeftMsgsInOne)$
26:     if $\text{DIFF}(CovA, CovOL) > N\delta$ then
27:         $AllSplit \leftarrow$ true
28:     end if
29: end if
```

C. Connection reuse

Connection reuse in HSPFuzzer is primarily implemented through a connection management module and a coverage monitoring module. Note that HSPFuzzer now does not reuse connection in UDP-based servers, since they are either connection-less (session-less) or implement protocol-specific connection management functions incompatible with general hooking techniques.

Connection Management. The connection management module is responsible for coordinating the establishment and termination of network connections, transmitting prefix packets, and restarting the SUT when necessary. When a new input is generated by mutating a seed and is ready for execution, the module first checks whether an active network connection is available. If no connection is ready, it instructs the fuzzer stub to initiate a new connection to the SUT. Otherwise, the available connection is immediately used to process the input.

If a new network connection is established, the module selects a sequence of prefix packets, as identified in the previous subsection, and transmits them to stabilize the connection, ensuring it can correctly process subsequent ordinary messages. If the fuzzer stub detects that the connection has been closed or if input processing repeatedly times out (e.g., more than five consecutive timeouts), the module marks the current connection as closed and initiates a new one. Additionally, if frequent connection restarts occur due to timeouts, the module restarts the SUT, as persistent failures may indicate that the SUT has encountered an issue requiring a clean restart.

To eliminate network overhead when restarting a connection, the fuzzer stub employs a specialized mechanism. First, it creates a *dummy socket*. Since network protocol servers invoke `bind` to associate with a specific port and `listen` to accept incoming connections, the fuzzer stub hooks these functions. Within `bind`, it verifies whether the specified port and protocol type match those configured in the fuzzer settings and records the file descriptor of the server socket. Subsequently, in `listen`, it checks whether the provided parameter matches the recorded socket descriptor. If so, it creates a new socket and connects it to the recorded server socket, establishing a *dummy socket* (`dummy_fd`).

Then, whenever the fuzzer stub is instructed to start a new network connection, it simulates network I/O operations, ensuring that when the SUT calls `accept`, it always receives the dummy socket `dummy_fd`. This approach prevents the establishment of actual network connections for subsequent interactions. Moreover, since the dummy socket is a real Internet socket rather than a UNIX socket, as used in [12], [24], [25], the SUT can invoke any network functions on it, ensuring maximum compatibility.

Coverage Monitoring. While the connection management module and fuzzer stub continuously monitor the connection status and initiate restarts when necessary, certain cases arise where the connection remains open but fails to process messages correctly. For example, Mosquitto [26], an MQTT (Message Queuing Telemetry Transport) broker server, may encounter issues when the *Remaining Length* field [27] of a message is excessively large. In such scenarios, the server continuously reads incoming data into memory but does not process it until the entire specified length has been received. Consequently, a large number of fuzzing inputs remain unprocessed, rendering fuzzing ineffective.

To mitigate this issue, HSPFuzzer employs a coverage monitoring module based on an Exponentially Weighted Moving Average (EWMA) algorithm. EWMA dynamically adjusts its weighting, prioritizing recent data while gradually diminishing the influence of older data. The module maintains an exponential moving average value, E_{avg}. After each input execution, the coverage metrics of the current and previous inputs are recorded as $Curr_{cov}$ and $Prev_{cov}$ (initialized to 0). The coverage difference, D_{cov}, is computed as:

$$D_{cov} = \textsc{Diff}(Curr_{cov}, Prev_{cov}), \tag{1}$$

and updates E_{avg} as follows:

$$E_{avg} = \alpha D_{cov} + (1 - \alpha) E_{avg}. \tag{2}$$

Finally, the module evaluates whether E_{avg} has dropped abnormally below a predefined threshold γ (set to 1 in our experiments). A significant drop suggests that the server is in a degraded state, repeatedly executing the same operations without discovering new coverage. In such cases, the module instructs the connection management module to restart the connection.

D. Message provision

The message provision module ensures that fuzzing inputs (i.e., messages) received from the fuzzer are promptly delivered to the SUT for processing without unnecessary delays. It comprises a network I/O hook module that intercepts and manipulates the SUT's network I/O functions, and a control module, which operates as a provisioning loop complementing the fuzzing loop that continuously generates inputs [4], [21].

Network I/O Hook. Specifically, the network I/O hook provides the following functionalities to the provisioning loop through `LD_PRELOAD`-based dynamic hooking: (a) initialization (including creating the dummy socket and launching the message provisioning loop), (b) manipulation of socket read and write operations (including redirecting read requests to shared memory and dropping unnecessary writes), (c) acceptance of new connections, (d) connection closure, and (e) monitoring of connection closures.

Provisioning Loop. The provisioning loop is started as a separate thread within the network I/O hook module. It needs to determine when the SUT has finished processing all messages in an input, allowing it to send the next input without delay. Other fuzzers such as AFLNet [10], [15], [12] terminate the SUT (e.g., via a SIGTERM signal) after sending the last message, using the termination event as an indicator that processing has ended. HNPFuzzer [17] does not always terminate the SUT after each input but does close the connection, using connection closure as an indicator. HSPFuzzer differs in that it may retain both the SUT and the connection after processing an input.

To address this, we identify two reliable conditions that signal processing completion and apply to both TCP- and UDP-based SUTs. The first condition is when the SUT attempts to read the next input, implying that it has finished processing the current input. Notably, the point at which the SUT completes reading an input (e.g., when `recv` returns) does not necessarily mark the completion of processing, as further computation may be required. The second condition is connection closure. Since HSPFuzzer supports connection reuse, an explicit connection termination usually indicates that the SUT has received a quit message and intends to terminate the session. Additionally, a timeout condition is incorporated to handle corner cases where neither of the primary conditions is met.

Algorithm 3 outlines the provisioning loop. The loop first retrieves an input and a command from the fuzzer via shared memory and a dedicated pipe, respectively (line 5). Based on the received command, the loop triggers the connection closure functionality of the network I/O hook module if necessary (line 7), followed by invoking acceptance of new connections functionality if required (line 10). Subsequently, it signals the network I/O hook module to allow the SUT to start reading input via the manipulation of socket read operations (line 12). The loop then waits for the SUT to complete processing and obtain a result (line 13). The result corresponds to one of the three aforementioned possible conditions. Finally, the result is sent back to the fuzzer to update the status of the fuzzer stub.

IV. Evaluation

In our experiments, we aim to address the following research questions:

- **RQ1.** Does HSPFuzzer achieve higher fuzzing throughput?
- **RQ2.** Does HSPFuzzer attain greater code coverage?
- **RQ3.** How does connection reuse enhance the fuzzing efficiency of HSPFuzzer?
- **RQ4.** Can HSPFuzzer detect more vulnerabilities?

Algorithm 3 Provisioning Loop

```
1:  global $HasConn$ ← false // Also be set to false when monitoring the connection closure
2:  global $SharedMem$ ← $\emptyset$
3:  while true do
4:      $NeedClose$ ← false
5:      RecvCommand($SharedMem$, &$NeedClose$) // receive an input and an command (i.e., closing connection or not) from the fuzzer side
6:      if $NeedClose$ is true and $HasConn$ is true then
7:          CloseConnection(&$HasConn$)
8:      end if
9:      if $HasConn$ is false then
10:         CreateConnection(&$HasConn$)
11:     end if
12:     StartToReadMessages($SharedMem$)
13:     $Res$ ← WaitForFinishing() // Result could be Normal|Closed|Timeout
14:     SendResults($Res$)
15: end while
```

A. Experiment setup

We evaluated HSPFuzzer on 12 servers, each implementing a distinct network protocol. Our selection includes nine servers from ProFuzzBench [23], along with three widely used servers: Mosquitto, ippsample, and Redis. We used LLVM version 12, the latest version supported by AFLNet. Similar to HNPFuzzer [17], HSPFuzzer does not require patching servers to terminate after processing each fuzzing input.

The HSPFuzzer prototype is implemented based on AFL++ (v4.09c). For comparison, we selected five state-of-the-art fuzzers: AFLNet [10], AFLNwe [10], SnapFuzz [12], HNPFuzzer [17], and AFL++ [21]. AFL++ with desock (i.e., preeny) [25] is one of the recommended methods for fuzzing network protocols using AFL++ [21]. All fuzzers used the same initial seed set to ensure a fair comparison [23]. However, for SUTs where $NeedSplit$ is `true` (LightFTP and TinyDTLS), HSPFuzzer first extracted prefix messages from the pcap files as previously described. Additionally, if $AllSplit$ was also `true` (LightFTP), all messages were further split into separate seeds.

All experiments were conducted on a 64-bit Ubuntu 20.04 LTS system (kernel version 5.4.0-196-generic) running on an Intel Xeon Platinum 8124M CPU (3.00GHz) with 16 cores and 15GB of RAM. Each experiment was executed within a Docker container, with a runtime of 24 hours per trial, and repeated six times to mitigate randomness [28].

B. Fuzzing throughput (RQ1)

Fuzzing throughput is a critical factor influencing a fuzzer's ability to identify vulnerabilities within a given timeframe, as higher throughput allows the generation of more inputs for testing the SUT. We evaluated the performance of various fuzzers, with the results presented in Table IV. Some fuzzers could not be configured to support certain servers, which we denote as "-". In particular, OpenSSH and Exim posed challenges for fuzzing while maintaining their normal operation after serving a client. For instance, OpenSSH must be executed with the "-d" option to remain in the foreground but not detach as a daemon during fuzzing, but with the option it terminates after handling one client. Since we aim to avoid modifying the source code to alter default behavior, HSPFuzzer instead restarts these two SUTs after processing each input.

The results demonstrate that HSPFuzzer significantly outperforms other fuzzers in fuzzing throughput. Note that the comparison is generally fair since they use the same seeds (i.e., sending the similar messages in one execution), except LightFTP whereas HSPFuzzer uses seeds that contain one message only, as we explained in Section IV-A. On average, HSPFuzzer achieves a throughput 106,206.32% (1062×) higher than AFLNet, executing thousands or even tens of thousands of test cases per second across multiple servers, with an average throughput of 4,473.05 executions per second. This remarkable performance is attributed to connection reuse and efficient message provision. Connection reuse reduces much overhead as shown in Fig. 1 (c), and our efficient message provision also reduces waiting time. AFLNwe, which sends an entire input to the SUT at once, reduces some waiting time and achieves an average throughput 202.70% that of AFLNet. AFL++&desock, which utilizes a UNIX socket for communication, further increases throughput to 678.76% of AFLNet. SNAPFuzz, by deferring the fork point, improves throughput to 781.9% of AFLNet. HNPFuzzer introduces a persistent SUT execution mode but achieves only a 1,213.41% improvement over AFLNet. Our analysis indicates that this is due to its frequent SUT restarts, which occur whenever changes in global variables or heap memory are detected. However, this strategy results in unnecessary restarts, approaching 100% in certain programs, since many servers modify global variables or allocate additional memory for each new connection. For example, LightFTP assigns a session ID as a global variable that increments with each new connection. Such modifications have minimal impact on fuzz testing and do not necessitate frequent restarts.

C. Code coverage (RQ2)

Code coverage is a key metric for evaluating fuzzers [28], [29], as it directly reflects the extent of code explored and correlates moderately with vulnerability discovery ability [29]. To unify measurement, we measured coverage with llvm-cov [30], instead of gcov as used in ProFuzzBench. Since HSPFuzzer does not restart the SUT even when its coverage changes, we collected coverage data at runtime by setting the "%m" parameter in `LLVM_PROFILE_FILE` and periodically sending a signal to instruct the SUT to write its profile, with "%m" merging the output. Table V presents the final code coverage results, while Fig. 3 illustrates the coverage growth achieved by each fuzzer over 24 hours.

The results indicate that HSPFuzzer consistently outperforms other fuzzers in terms of code coverage across all tested servers. On average, its branch coverage exceeds that of AFLNet by 25.1%. The p-values from Mann-Whitney U test [28] comparing HSPFuzzer and AFLNet are all below 0.05, confirming statistical significance in their differences across all servers. The coverage growth curves reveal that HSPFuzzer achieves significantly higher coverage early in the fuzzing process, aligning with its superior fuzzing throughput. This suggests that the inputs executed by HSPFuzzer contribute

TABLE IV
AVERAGE FUZZING THROUGHPUT (EXECS/S) OF VARIOUS FUZZERS (THE CHANGES IN PARENTHESES ARE COMPARED TO AFLNET).

Server	AFLNET	AFLNWE	SNAPFuzz	HNPFuzzer	AFL++&desock	HSPFuzzer
Dnsmasq	6.82	27.49 (+303.08%)	62.39(+814.81%)	32.55 (+377.27%)	46.66 (+584.16%)	10359.15 (+151793.70%)
LightFTP	5.12	32.04 (+525.78%)	31.20 (+509.38%)	31.43 (+513.87%)	22.70 (+343.36%)	12058.14 (+235410.55%)
TinyDTLS	2.12	14.08 (+564.15%)	49.50 (+2234.91%)	219.88 (+10271.70%)	43.01 (+1928.77%)	11154.47 (+526054.25%)
Kamailio	4.61	5.80 (+25.81%)	-	8.43 (+82.86%)	-	6028.45 (+130668.98%)
OpenSSL	3.41	14.96 (+338.71%)	-	8.69 (+154.84%)	-	3668.30 (+107474.78%)
OpenSSH	19.12	22.02 (+15.17%)	-	27.7 (+44.87%)	3.2 (-83.26%)	29.65 (+55.07%)
DCMTK	18.42	15.83 (-14.06%)	24.74 (+34.31%)	6.97 (-62.16%)	17.84 (-41.15%)	792.13 (+4200.38%)
Live555	12.16	30.46 (+150.49%)	50.60 (+316.12%)	14.07 (+15.71%)	-	824.40 (+6679.61%)
Mosquitto	8.45	4.58 (-45.80%)	-	-	31.46 (+272.31%)	3683.51 (+43491.83%)
ippsample	7.31	9.09 (+24.35%)	-	41.52 (+467.99%)	37.20 (+408.89)	4005.61 (+54696.31%)
Redis	7.34	10.25 (+32.43%)	-	-	33.81 (+336.82%)	1065.31 (+13663.70%)
Exim	1.95	3.81 (+95.83%)	-	7.16 (+267.18%)	-	7.54 (+286.67%)
Average		**167.96%**	**781.90%**	**1213.41%**	**405.54%**	**106206.32%**

TABLE V
AVERAGE BRANCH COVERAGE OF VARIOUS FUZZERS WITH P-VALUES COMPARING AFLNET AND HSPFUZZER (THE CHANGES IN PARENTHESES ARE COMPARED TO AFLNET).

Server	AFLNET	AFLNWE	SNAPFuzz	HNPFuzzer	AFL++&desock	HSPFuzzer	p-value
Dnsmasq	1388.00	1407.00 (+1.37%)	1376.20 (-0.85%)	1381.90(-0.44%)	1136.30 (-18.13%)	1553.00 (+11.89%)	0.012
LightFTP	301.70	142.00 (-52.93%)	303.00 (+0.43%)	304.50 (+0.93%)	118.20 (-60.82%)	377.70 (+25.19%)	0.008
TinyDTLS	823.70	201.30 (-75.56%)	824.30 (+0.07%)	827.00 (+0.40%)	131.50 (-84.04%)	854.00 (+3.68%)	0.015
Kamailio	11075.00	9753.30 (-11.93%)	-	11358.80 (+2.56%)	-	14474.30 (+30.69%)	<0.001
OpenSSL	10711.30	9411.00 (-12.14%)	-	11066.70 (+3.32%)	-	12062.10 (+12.61%)	0.004
OpenSSH	3047.30	3052.30 (+0.16%)	-	3081.30 (+1.12%)	2413.30 (-20.81%)	3124.70 (+2.54%)	0.029
DCMTK	3765.50	3658.50 (-2.84%)	3805.00 (+1.05%)	3808.20 (+1.13%)	3521.30 (-6.49%)	4111.00 (+9.18%)	0.011
Live555	2314.00	1941.00 (-16.12%)	2364.70 (+2.19%)	2362.50 (+2.10%)	-	2383.00 (+2.98%)	0.017
Mosquitto	2403.40	2323.00 (-3.35%)	-	-	1564.20 (-34.92%)	3530.40 (+46.89%)	0.003
ippsample	3017.20	2735.00 (-9.35%)	-	3286.40 (+8.92%)	2892.30 (-4.14%)	3619.10 (+19.95%)	0.009
Redis	6537.80	8789.40 (+34.44%)	-	-	4556.00 (-30.31%)	14893.80 (+127.81%)	<0.001
Exim	3184.30	3103.00 (-2.55%)	-	3224.40 (+1.26%)	-	3423.70 (+7.80%)	0.002
Average		**-12.57%**	**+0.58%**	**+2.13%**	**-32.46%**	**+25.10%**	

effectively to exploring the SUTs. Moreover, connection reuse may facilitate the triggering of new execution paths in certain servers. For example, in Redis, executing inputs without restarting the SUT allows the server to accumulate data, potentially activating execution paths that require specific data states to be reached [31]. A similar effect is observed in other servers, such as Mosquitto.

Despite its higher throughput, AFLNwe exhibits lower branch coverage than AFLNet, consistent with prior findings [17]. This is because AFLNwe does not split messages in an input but instead sends them as a single packet, limiting its ability to effectively test servers like LightFTP, which processes only one command from a received buffer while discarding the rest or treating them as parameters. AFL++&desock suffers from the same limitation. Its branch coverage is even lower than that of AFLNwe, and we find that it may be attributed to its seed-trimming process, leading to many seeds being reduced to very short seeds (often only several bytes). HSPFuzzer disables both calibration and seed trimming [31]. SnapFuzz and HNPFuzzer achieve slightly higher branch coverage than AFLNet, with improvements of 0.58% and 2.13%, respectively. However, their overall gains are limited due to relatively low fuzzing throughput.

D. The Effect of connection reuse (RQ3)

1) The effect of prefix messages: Fig. 4 illustrates the coverage performance of HSPFuzzer without prefix messages on LightFTP and TinyDTLS since only the two SUTs use prefix messages separately. In LightFTP, HSPFuzzer without prefix messages requires approximately 10 additional hours to achieve the same coverage as its counterpart with prefix messages. This delay stems from HSPFuzzer's connection reuse mechanism, which facilitates the discovery of seeds that begin with different commands, as command names are utilized as dictionary entries within the fuzzer. Without connection reuse, login attempts are likely to fail, yielding results similar to those of AFLNwe and AFL++&desock. In the case of TinyDTLS, HSPFuzzer without prefix messages fails to establish a session, as only ClientHello messages are parsed while subsequent messages are discarded.

2) The number of times a connection is reused: It is unclear how many inputs can be effectively transmitted within a single connection during real-world fuzzing. To investigate this, we measured the number of connections utilized by different servers to process one million fuzzing inputs and computed the average number of inputs per connection. The results, presented in Table VI, exclude SUTs that do not have connection reuse (i.e., UDP-based servers and OpenSSH and Exim that are restarted for each input). The findings indicate that most servers efficiently handle multiple inputs within a single connection, thereby amortizing the overhead associated with connection setup and teardown. However, OpenSSL and DCMTK have stricter message handling, processing only 1.01 and 3.03 inputs per connection, respectively.

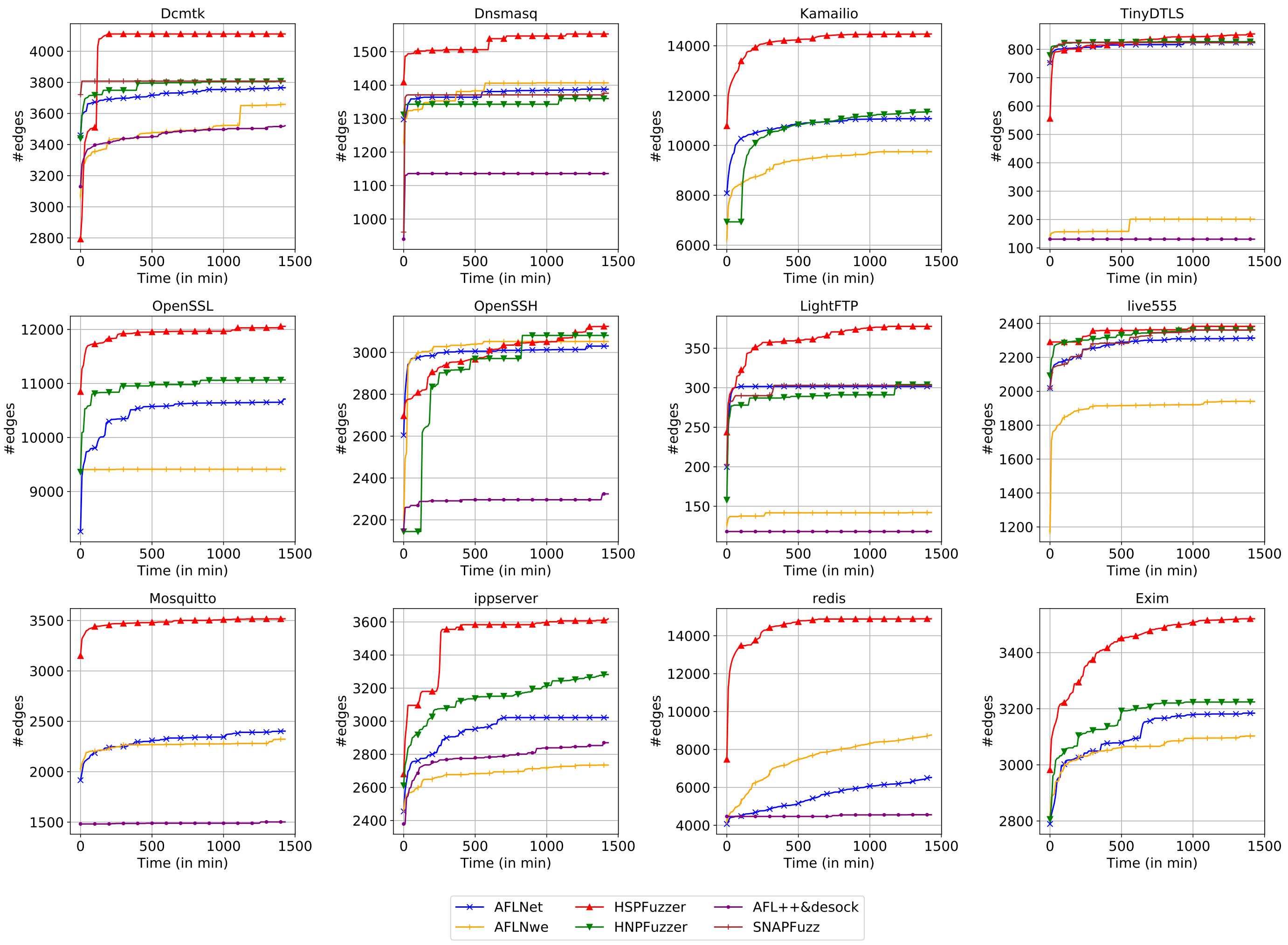

Fig. 3. Average branch coverage growth over 24h of various fuzzers.

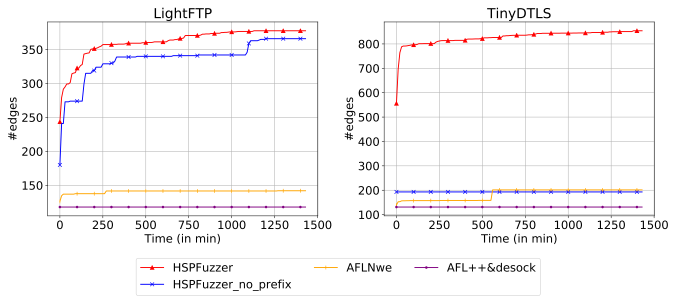

Fig. 4. HSPFuzzer with and without prefix messages.

TABLE VI
THE AVERAGE NUMBER OF INPUTS SENT ON A CONNECTION (K = THOUSAND, M = MILLION). TESTING USING 1M INPUTS.

Program	OpenSSL	LightFTP	Live555	DCMTK	Redis	Mosquitto	ippsample
Reuse Count	1.01	>1M	306.49	3.03	619.73	52.61K	1.21K

3) The impact of connection reuse on the coverage ability of inputs: Section IV-C establishes that HSPFuzzer significantly outperforms other fuzzers in terms of coverage. However, it remains unclear whether this advantage is due to its high fuzzing throughput (Section IV-B) or whether connection reuse actually impairs input effectiveness in coverage. We selected four servers where connection reuse was particularly effective (more than 300 reused inputs per connection). We then executed 10K inputs against these servers using two configurations: one leveraging connection reuse and the other restarting the SUT for each input (i.e., no connection reuse). We employed two coverage metrics provided by AFL++ (and AFL): *hit-count* and *tuple* coverage. A tuple represents a byte in the coverage map and corresponds to a new edge (i.e., SanitizerCoverage) under the afl-clang-fast instrumentation method. Hit-count coverage tracks execution frequency per tuple, categorizing occurrences into eight buckets (i.e., 1, 2, 4, 8, 16, 32, 64, 128). Both metrics contribute to seed selection in AFL++, with tuple coverage also influencing favored seed calculations.

The experiment was repeated five times, and the average results are depicted in Fig. 5. Overall, connection reuse does not degrade coverage performance and, in some cases, enhances it. For tuple coverage, connection reuse achieves significantly higher coverage in Mosquitto, LightFTP, and Redis, while maintaining comparable performance in Live555. In terms of hit-count coverage, connection reuse leads to improved results in Mosquitto and LightFTP, similar coverage in Live555, and slightly reduced coverage in Redis. These findings suggest that connection reuse does not hinder the exploration of SUT execution paths and is recommended for its substantial performance benefits.

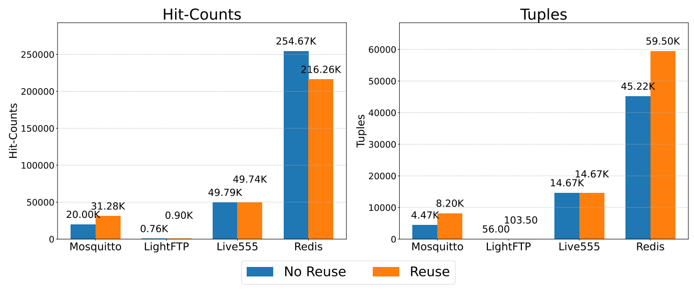

Fig. 5. The coverage ability of 10K inputs with and without connection reuse, which is measured by the number of new hit-counts and tuples.

4) The effect of coverage monitoring: We only enable coverage monitoring in Mosquitto since only the SUT may enter abnormal conditions. This server invokes `loop_handle_reads_writes()` to process socket data before returning control to the upper layer. We configured HSPFuzzer to fuzz Mosquitto with one million inputs and recorded the number of times `loop_handle_reads_writes()` successfully returned to the upper layer. The results indicate that enabling coverage monitoring increased this count from 4.78M to 5.19M, an 8.6% improvement over the configuration without coverage monitoring. This suggests that coverage monitoring can enhance fuzzing effectiveness by mitigating server anomalies.

E. Vulnerability discovery (RQ4)

To evaluate HSPFuzzer's capability in identifying vulnerabilities, we compared its performance against other fuzzers. Detected crash seeds were replayed on SUTs built with AddressSanitizer (ASan), and unique vulnerabilities were manually analyzed based on deduplicated stack traces from the top three stack frames [28], [32], [5]. Table VII shows the average time to discover the vulnerabilities by different fuzzers. All identified vulnerabilities have been publicly disclosed and are absent from the latest software versions. The results demonstrate that HSPFuzzer detects at least 4 more vulnerabilities than other fuzzers, and discovers most issues within the first hour, aligning with its rapid coverage expansion at the start of fuzzing.

Furthermore, the memory leak vulnerability CVE-2021-41690 in DCMTK, discovered exclusively by HSPFuzzer, highlights its unique advantage in detecting vulnerabilities triggered by a high volume of messages. This memory leak results from the missing deallocation of two lists `presentationContextList` and `userInfo` allocated for a client connection in `parseAssociate` in some cases. If the SUT is not instrumented with ASan or leak detection is disabled for performance [10], and vulnerabilities are detected only during crash replay, as in our evaluation and other studies [32], [33], such issues can remain undetected by conventional fuzzers. The leaked memory per connection is minimal, and since AFLNet restarts the SUT for each connection (or HNPFuzzer restarts after a limited number of connections), the total memory leakage remains insufficient to cause a crash. In contrast, HSPFuzzer's connection reuse enables the accumulation of leaked memory, leading to an out-of-memory crash, even in the absence of ASan instrumentation.

TABLE VII
TIME TO DISCOVER SPECIFIC VULNERABILITIES AND TOTAL FOUND NUMBER BY DIFFERENT FUZZERS (M = MINUTE, S = SECOND)

Subject	Vulnerability	AFLNet	AFLNwe	SnapFuzz	HNPFuzzer	AFL++	HSPFuzzer
Live555	CVE-2018-4013	11m50s	20m39s	1m42s	10m55s	✗	1m1s
	CVE-2021-38381	8m57s	34m31s	15m35s	17m0s	✗	7m21s
	CVE-2021-38382	13m5s	26m16s	12m35s	12m52s	✗	12m20s
	CVE-2021-39282	18m45s	44m14s	14m54s	12m15s	✗	19m49s
TinyDTLS	Bug#544819	<1m	<1m	<1m	<1m	✗	<1m
	buffer-overflow2	1m53s	<1m	<1m	<1m	✗	<1m
	buffer-overflow3	5m51s	7m36s	3m11s	<1m	✗	<1m
	assertion fail	✗	✗	✗	14m32s	✗	152m47s
DCMTK	Bug#942	105m57s	156m32s	✗	✗	✗	31m5s
	segfault	181m20s	174m53s	14m32s	10m37s	1000m 30s	37m25s
	SEGV	213m2s	150m43s	✗	✗	✗	51m58s
	CVE-2021-41690	✗	✗	✗	✗	✗	18m
	memory leak	✗	✗	✗	✗	✗	11m34s
Redis	CVE-2024-31227	✗	✗	✗	✗	✗	4m51s
Number	14	10	10	8	9	1	14

Similarly, CVE-2024-31227 in Redis demonstrates the advantage of HSPFuzzer's superior fuzzing throughput. This vulnerability arises from an error when parsing the `ACL` command in the `ACLSetSelector` function, which results in a later crash in the `ACLDescribeSelector` function. It is relatively simple to trigger. However, other fuzzers fail to detect it within 24 hours due to their lower throughput. Given that Redis v7.x includes about 460 commands, an extensive number of fuzzing inputs is required for comprehensive testing. HSPFuzzer's high throughput enables it to uncover this vulnerability efficiently (within five minutes).

V. DISCUSSIONS AND LIMITATIONS

When connection reuse is effective, users might assume that HSPFuzzer primarily tests ordinary messages while rarely exercising prefix messages, as these are typically sent only at the start of a connection. However, this is not a concern. First, prefix messages can often be transmitted multiple times within the same connection, effectively functioning as ordinary messages once the prefix packets have been sent. For instance, in FTP, a client may issue `USER` and `PASS` commands multiple times to log in [20]. Second, even in protocols where prefix messages are restricted to the initial phase of a connection, HSPFuzzer can be configured to intentionally restart connections at fixed time intervals to ensure their inclusion in testing.

HSPFuzzer also features a focus mode, which involves sending multiple random seed messages before executing the current input. This approach may increase the likelihood of reaching diverse states, particularly when the provided seeds lack necessary prior messages. In our evaluation, the seeds were well-formed and did not exhibit this issue, and we observed that enabling focus mode did not yield an increase in branch coverage. As a result, while this feature remains implemented in our prototype, it is not enabled by default and may be enabled by users when seeds are randomly collected.

Similar to fuzzers operating in persistent mode [4], [21], crash-triggering inputs saved by HSPFuzzer may not always contain all the messages necessary to reproduce the crash in a standalone replay. AFL++ provides a configuration option, `AFL_PERSISTENT_RECORD`, to capture additional input data and mitigate this issue [21]. Alternatively, users may attach a debugger such as gdb to the SUT for post-mortem analysis, as demonstrated in [34]. Another viable solution is to employ snapshot-based techniques, like a feature recently integrated into syzkaller [35], to preserve the execution state for debugging.

VI. Related work

Existing grey-box network protocol fuzzing approaches could be roughly classified into three categories.

A. Improving fuzzing throughput

Since servers receive inputs via network sockets, several fuzzers focus on optimizing fuzzing throughput [25], [10], [24], [14], [13], [15], [12], [22], [17], [36], [37]. Desock (preeny) [25] hooks the SUT's socket and redirects it to the console. Desockmulti [24] improves upon desock by eliminating delays associated with interaction support. AFLNet and AFLNwe [10] were among the first to introduce internet socket-based input transmission for SUTs.

To address SUT startup overhead, Nyx-Net [14] and SNPSFuzzer [13] employ KVM (Kernel-based Virtual Machine) and CRIU (Checkpoint and Restore in Userspace) to snapshot the SUT. NSFuzz [15] reduces delays by manually annotating the end of message processing. SnapFuzz [12] introduces a smart deferred forkserver to minimize process forking time. GreenFuzz [36] optimizes message buffering to reduce fuzzer-SUT message-passing overhead. Netfuzzlib [37] intercepts network I/O functions to further decrease latency. However, these approaches restart or restore the SUT for each input, leading to unnecessary performance overhead.

HNPFuzzer [17] and EQUAFL [22] adopt a different approach by eliminating SUT restarts for each input, akin to persistent mode fuzzing in traditional fuzzers [4], [21]. Additionally, HNPFuzzer utilizes shared memory for message transmission, further reducing overhead. In this paper, our findings reveal that connection reuse within inputs can further enhance fuzzing throughput.

B. Improving seed scheduling and mutation

Several approaches incorporate state coverage to enhance seed scheduling and mutation strategies [38], [10], [39], [11], [15], [16]. IJON [38] first introduced state coverage as an extension to traditional branch coverage [4], using manual annotations to collect state information. AFLNet [10] builds a protocol state machine by instrumenting response code parsing in messages and prioritizing seeds that traverse rare states. Alternative methods focus on refining server state extraction. NSFuzz [15] employs manual annotations on specific variables to track state transitions. StateAFL [11] infers server state by analyzing snapshots of persistent memory regions. SGFuzz [16] instruments enum variables in source code to capture state transitions. Black-box techniques also attempt to infer state coverage from packet sequences [40] or even higher-level semantic coverage using log analysis [41].

In terms of seed mutation, recent work has explored innovative strategies [42], [43]. Fuzztruction-Net [42] introduces fault injection by modifying the SUT's peer (e.g., the client when fuzzing a server) to ensure that mutated messages pass integrity and encryption checks. ChatAFL [43] leverages LLMs (Large Language Models) to enhance the seed corpus, generate new seeds, and perform intelligent mutation. These seed scheduling and mutation optimizations are orthogonal to HSPFuzzer.

C. Adapting to specific protocols

Several protocol-specific fuzzing techniques have been proposed, targeting TLS [44], [45], DTLS [46], Bluetooth [47], TCP [48], DNS [49], Matter [50], and MQTT [51]. These approaches typically integrate domain knowledge into fuzzing architectures, input generation, and vulnerability detection. For example, ResolverFuzz [49] utilizes a name server to fuzz DNS resolvers, while MBFuzzer [51] employs two senders to fuzz MQTT brokers. Since well-formed inputs are crucial for effective fuzzing [52], most protocol-specific fuzzers generate inputs based on protocol specifications [44], [46], [47]. LLMs have also been employed for learning protocol semantics and improving input generation [50]. Beyond crash-inducing vulnerabilities, these fuzzers also detect specification violations [44], [46], [50], and LLMs have been utilized in this context as well [51]. Additionally, differential testing has been widely adopted to compare different implementations of the same protocol and identify inconsistencies [45], [48], [49], [51]. HSPFuzzer differs from these approaches as it is designed for general protocol fuzzing rather than being tailored to a specific protocol.

VII. Conclusion

The primary contribution of this work is the development of HSPFuzzer, a high-performance network protocol fuzzer. By reusing connections, HSPFuzzer significantly reduces fuzzing overhead. Additionally, we introduce an efficient message provisioning mechanism via a fuzzer stub. HSPFuzzer also eliminates the need for manually writing message-splitting code. Experimental results demonstrate that HSPFuzzer substantially outperforms state-of-the-art fuzzers such as AFLNet, AFLNwe, SnapFuzz, HNPFuzzer, and AFL++. For instance, it achieves a 1062× increase in fuzzing throughput over AFLNet.

We believe that HSPFuzzer further advances the field of network protocol fuzzing by reducing unnecessary overhead and improving efficiency. Moreover, as it requires no additional code, it provides a more user-friendly fuzzing solution.

References

[1] "The heartbleed bug," 2014. [Online]. Available: https://heartbleed.com/

[2] "CVE-2017-0144 (EternalBlue)," 2017. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-0144

[3] Qualys, "regreSSHion Exploit (PoC)," 2024. [Online]. Available: https://github.com/xonoxitron/regreSSHion

[4] M. Zalewski, "AFL - American Fuzzy Lop," 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[5] 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 Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, 2021.

[6] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Comput. Surv.*, vol. 54, no. 11s, Sep. 2022.

[7] X. Zhang, C. Zhang, X. Li, Z. Du, B. Mao, Y. Li, Y. Zheng, Y. Li, L. Pan, Y. Liu, and R. Deng, "A survey of protocol fuzzing," *ACM Comput. Surv.*, vol. 57, no. 2, Oct. 2024.

[8] H. Elahi and G. Wang, "Forward-porting and its limitations in fuzzer evaluation," *Inf. Sci.*, vol. 662, p. 120142, 2024.

[9] Y. Li, Y. Zeng, X. Song, and S. Guo, "Improving seed quality with historical fuzzing results," *Inf. Softw. Technol.*, vol. 179, p. 107651, 2025.

[10] V. Pham, M. Böhme, and A. Roychoudhury, "AFLNet: A greybox fuzzer for network protocols," in *Proc. 13th IEEE Int. Conf. Softw. Test. Verif. Valid. (ICST '20), Testing Tools Track*, 2020, pp. 460–465.

[11] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 191, 2022.

[12] A. Andronidis and C. Cadar, "Snapfuzz: high-throughput fuzzing of network applications," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA '22)*, 2022, p. 340–351.

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

[14] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: network fuzzing with incremental snapshots," in *Proc. 17th Eur. Conf. Comput. Syst. (EuroSys '22)*, 2022, p. 166–180.

[15] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "Nsfuzz: Towards efficient and state-aware network service fuzzing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, Sep. 2023.

[16] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *Proc. 31th USENIX Secur. Symp. (USENIX Security '22)*, Aug. 2022, pp. 3255–3272.

[17] J. Fu, S. Xiong, N. Wang, R. Ren, A. Zhou, and B. K. Bhargava, "A framework of high-speed network protocol fuzzing based on shared memory," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 4, pp. 2779–2798, 2024.

[18] B. Contributors, "Boofuzz: Network protocol fuzzing for humans," 2024, accessed: 2024-10-15. [Online]. Available: https://github.com/jtpereyda/boofuzz

[19] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proc. 29th ACM Joint Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng. (ESEC/FSE '21)*, 2021, pp. 1393–1403.

[20] J. Postel and J. Reynolds, "File Transfer Protocol," Tech. Rep., 1985.

[21] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *Proc. 14th USENIX Worksh. Offens. Technol. (WOOT '20)*, 2020.

[22] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation," in *Proc. 31th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA '22)*, 2022, pp. 417–428.

[23] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA '21)*, 2021.

[24] 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, 2020.

[25] Y. Shoshitaishvili, "Some helpful preload libraries for pwning stuff," 2016. [Online]. Available: https://github.com/zardus/preeny

[26] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," *Int. J. Open Source Softw. Process.*, vol. 2, no. 13, p. 265, 2017.

[27] OASIS, "Mqtt version 5.0 specification," 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[28] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS '18)*, 2018, pp. 2123–2138.

[29] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proc. 44th Int. Conf. Softw. Eng. (ICSE '22)*, 2022, pp. 1621–1633.

[30] LLVM Compiler Infrastructure Project, *LLVM-Cov: A Code Coverage Tool*, Online, https://llvm.org/docs/CommandGuide/llvm-cov.html.

[31] Y. Zeng, F. Zhu, S. Zhang, Y. Yang, S. Yi, Y. Pan, G. Xie, and T. Wu, "Dafuzz: data-aware fuzzing of in-memory data stores," *PeerJ Comput. Sci.*, vol. 9, p. e1592, 2023.

[32] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, "UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers," in *Proc. 30th USENIX Secur. Symp. (USENIX Security '21)*, 2021, pp. 2777–2794.

[33] Y. Gao, W. Zeng, S. Liu, and Y. Zeng, "Autofuzz: automatic fuzzer-sanitizer scheduling with multi-armed bandit," *Softw. Qual. J.*, vol. 33, no. 1, p. 8, 2025.

[34] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS '21)*, 2021, pp. 2197–2213.

[35] D. Vyukov, "Pull request - add snapshot-based fuzzing," 2024. [Online]. Available: https://github.com/google/syzkaller/pull/5091

[36] S. B. Andarzian, C. Daniele, and E. Poll, "Green-fuzz: Efficient fuzzing for network protocol implementations," in *Proc. 17th Int. Symp. Found. Pract. Secur. (FPS '24)*, 2024, pp. 253–268.

[37] J. Robben and M. Vanhoef, "Netfuzzlib: Adding first-class fuzzing support to network protocol implementations," in *29th Eur. Symp. Res. Comput. Secur.*, 2024, pp. 65–84.

[38] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *Proc. IEEE Symp. Secur. Priv. (SP '20)*, 2020, pp. 1597–1612.

[39] D. Liu, T. Pham, G. Ernst, T. Murray, and B. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER '22)*, 03 2022, pp. 720–730.

[40] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," in *Proc. 32th USENIX Secur. Symp. (USENIX Security '23)*, 2023, pp. 4481–4498.

[41] F. Wu, Z. Luo, Y. Zhao, Q. Du, J. Yu, R. Peng, H. Shi, and Y. Jiang, "Logos: Log guided fuzzing for protocol implementations," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA '24)*, 2024, p. 1720–1732.

[42] N. Bars, M. Schloegel, N. Schiller, L. Bernhard, and T. Holz, "No peer, no cry: Network application fuzzing via fault injection," in *Proc. ACM Conf. Comput. Commun. Secur. (CCS '24)*, 2024, p. 750–764.

[43] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proc. 31st Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS '24)*, vol. 2024, 2024.

[44] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proc. 23rd ACM Conf. Comput. Commun. Secur. (CCS '16)*, 2016, pp. 1492–1504.

[45] Z. Zhao, X. Song, Q. Zhong, Y. Zeng, C. Hu, and S. Guo, "Tls-deepdiffer: message tuples-based deep differential fuzzing for tls protocol implementations," in *IEEE Int. Conf. Softw. Anal., Evol., Reeng. (SANER '24)*, 2024, pp. 918–928.

[46] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. De Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *Proc. 29th USENIX Secur. Symp. (USENIX Security '20)*, 2020, pp. 2523–2540.

[47] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, "BrakTooth: Causing havoc on bluetooth link manager via directed fuzzing," in *Proc. 31th USENIX Secur. Symp. (USENIX Security '22)*, 2022, pp. 1025–1042.

[48] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC '21)*, 2021, pp. 489–502.

[49] Q. Zhang, X. Bai, X. Li, H. Duan, Q. Li, and Z. Li, "ResolverFuzz: Automated discovery of DNS resolver vulnerabilities with Query-Response fuzzing," in *Proc. 33th USENIX Secur. Symp. (USENIX Security '24)*, Philadelphia, PA, Aug. 2024, pp. 4729–4746.

[50] X. Ma, L. Luo, and Q. Zeng, “From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter IoT devices,” in *Proc. 33th USENIX Secur. Symp. (USENIX Security ’24)*, 2024, pp. 4783–4800.
[51] X. Song, J. Wu, Y. Zeng, H. Pan, C. Zuo, Q. Zhao, and S. Guo, “Mbfuzzer: A multi-party protocol fuzzer for mqtt brokers,” in *Proc. 34th USENIX Secur. Symp. (USENIX Security ’25)*, Aug. 2025.
[52] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, “Smart greybox fuzzing,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1980–1997, 2019.

Zhewei Xia is currently pursuing a bachelor’s degree with the Zhuoyue Honors College, Hangzhou Dianzi University, Hangzhou, China. His research interests are in software and system security.

Yingpei Zeng is currently an Associate Professor at Hangzhou Dianzi University. His current research interest is software security. He received his B.S. and Ph.D. degrees from the Department of Computer Science and Technology, Nanjing University, in 2004 and 2010, respectively. He also worked as a Research Assistant in the Department of Computing, Hong Kong Polytechnic University, from May 2008 to May 2009. Before joining Hangzhou Dianzi University, he worked in China Mobile and MicroStrategy as a software engineer and product owner for eight years. He has published more than 30 papers with others in different journals and conferences.

Xiangpu Song received the B.Eng. degree in computer science and technology from Chengdu University of Technology, Chengdu, China, in 2021. He is currently pursuing a Ph.D. degree at the School of Cyber Science and Technology, Shandong University, Qingdao, China. His research focuses on protocol security and fuzzing.

Shanqing Guo is currently a professor with the School of Cyber Science and Technology at Shandong University. He received his M.S. and Ph.D. degrees in computer science from Ocean University, China, in 2003, and Nanjing University, China, in 2006, respectively. He joined the School of Computer Science and Technology at Shandong University as an assistant professor in 2006. His research interests include AI Security, Data-driven Security, Software and System Security. He has published in TSE, TDSC, IEEE S&P, USENIX Security, ACM CCS, and other venues.

Ting Wu received his Ph.D. degree from Shandong University, China. He is currently serving as a Full Professor of Hangzhou Innovation Institute, Beihang University, China. He has actively participated in several nationally funded projects as principal investigator and senior researcher. He is the author of a number of conference and journal publications and served as general co-chair of the IEEE ICCCN conference in 2018. His research interest is information security.