LibAFLstar: Fast and State-Aware Protocol Fuzzing

Cristian Daniele[1],[0000−0001−7435−4176], Timme Bethe[2],[0009−0006−6958−5328]
Marcello Maugeri[3],[0000−0002−6585−5494],
Andrea Continella[2],[0000−0002−0329−1830] and Erik Poll[1][0000−0003−4635−187X]

[1] Radboud University, Nijmegen, The Netherlands
{cristian.daniele,erik.poll}@ru.nl
[2] University of Twente, Enschede, The Netherlands
timme@timmebethe.nl, a.continella@utwente.nl
[3] University of Catania, Catania, Italy
marcello.maugeri@phd.unict.it

Abstract. Fuzzing is arguably one of the most effective software vulnerability discovery techniques. However, despite recent advances, fuzzing stateful software suffers from severe inefficiencies and scalability limitations. This hinders automated testing for software that relies on state models, such as protocol implementations. Unlike stateless approaches, efficient stateful fuzzers need to i) explore the state model of the target system, ii) focus on the most interesting states, iii) track which messages are interesting for each state, and iv) handle expensive restarts and synchronizations of the system. In this paper, we present LibAFLstar, a fast and state-aware protocol fuzzer that addresses the aforementioned challenges leveraging i) partial message sequences, ii) a novel state scheduler, iii) state-aware queues and bitmaps, and iv) persistent mode. We fine-tune our approach by running an extensive ablation study with more than 20 configurations over six protocol implementations. Then, we evaluate LibAFLstar on the same protocol implementations (FTP, RTSP and HTTP) for 24 hours. We compare LibAFLstar's performance with two state-of-the-art fuzzers: AFLNet and ChatAFL. Our experiments show that LibAFLstar is more than 30× faster than competitors and achieves, on average, 1.4× more coverage.

Keywords: Software Security · Software Testing · Fuzzing · Stateful Systems· Network Protocols

1 Introduction

Fuzzing is a well-known technique for finding bugs in systems. The idea behind fuzzing is simple: sending unexpected (e.g., malformed) messages to a System Under Test (SUT) to trigger unexpected behaviour. Despite being introduced more than 30 years ago, fuzzing has become very popular with the advent of AFL [36] (and its re-implementation AFL++ [16]), a *smart mutation-based* fuzzer highly effective at finding vulnerabilities in modern systems. In recent years, a plethora of fuzzers have been devised to target a variety of systems [8, 15, 30, 19], as surveyed in recent work [38, 23, 21, 13]. Nevertheless, AFL++ and its descendant LibAFL [17] remain the best choice when it comes to fuzzing *stateless* targets because of their speed and effectiveness.

Unfortunately, the effectiveness of AFL++ and generic smart mutation-based fuzzers are often confined to *stateless systems*—systems that do not require the implementation of a state model to function properly. Classical examples of stateless systems are PDF readers, MP3 players, or image libraries.

When dealing with *stateful systems*, existing smart mutation-based fuzzers are, unfortunately, not as effective. Stateful systems are characterized by a state model. Typical examples include any client or server implementing network protocols, such as FTP, SSH, TLS, and 5G. In these systems, message sequences (often called traces) play a crucial role, as unexpected traces may also expose vulnerabilities. Stateful fuzzing is much more challenging because of the difficulty in tracking states and dealing with dependencies between messages, adding a further layer of complexity to the typical fuzzing challenges [7]. In fact, stateful fuzzers need to understand and explore the state model of the target system (challenge C1), focus on the most interesting states (challenge C2), consider relations between states and interesting messages (challenge C3), and deal with expensive SUT restarts and synchronization (challenge C4).

Several stateful fuzzers have been proposed in recent years [10, 37, 9], however, none has fully addressed the aforementioned challenges. In fact, most approaches [27, 35, 4] require shutting down and restarting the SUT and require snapshotting to provide a new trace (i.e., a new sequence of messages) [31]. Also, all the above-mentioned tools do not use the actual state model of the SUT, implementing only a shallow notion of statefulness.

In this paper, we present LibAFLstar, a fast and state-aware protocol fuzzer that solves the four above-mentioned challenges by combining:

1. **knowledge about the state model** — solving C1;
2. **state schedulers** to focus on the most interesting states — solving C2;
3. **state-aware queues** to group relevant messages by state — solving C3;
4. **persistent mode** to avoid resets of the SUT after every trace and expensive snapshotting, allowing for fast synchronization between the fuzzer and the SUT — solving C4.

We implement LibAFLstar on top of LibAFL and evaluate it on six targets from ProFuzzBench [26], comparing the performance of our tool with the state-of-the-art stateful fuzzers AFLNet [27] and ChatAFL [25].

In our experiments, we show that LibAFLstar is, on average, 20× faster and achieves up to 3× the code coverage[4] of AFLNet and ChatAFL.

[4] In the paper, we use the term code coverage instead of *edge coverage* to avoid confusion with the edge coverage in the state model.

2 Background on Fuzzing

Stateless fuzzing. According to the technique used to generate messages, fuzzers are divided into *grammar-based* and *mutation-based. Grammar-based* fuzzers leverage the grammar of the inputs (given by the analyst or automatically inferred) to generate inputs that slightly differ from the grammar [34], or to generate grammar-compliant inputs that pass specific checks or constraints in the code [24]. *Mutation-based* fuzzers do not require grammar; instead, they take sample messages as input (often called *seeds*) to start the mutations. *Smart* mutation-based fuzzers like AFL++ (often called *grey-box*) do not perform blind mutations, as brute-force or simple mutation-based tools do, but instead use heuristics (e.g., code coverage) to steer the generation of the messages towards the most promising ones. Specifically, smart mutation-based fuzzers store code coverage information in a matrix (a *bitmap*) and use a *queue* to store messages that trigger new code coverage, allowing for further mutations later.

Stateless fuzzers require little else: input grammar or a few seeds are often sufficient to find numerous bugs as proven by OSS-Fuzz [32], a Google initiative that found 36.000 bugs in 1.000 projects[5] thanks to three stateless fuzzers (AFL++ [16], libFuzzer[6] and Honggfuzz[7]).

Stateful fuzzing. Although the boundary between stateful and stateless systems can sometimes be vague [10], it is clear that for efficient fuzzing of certain systems, the grammar of the messages or seed files alone is insufficient. In fact, stateful systems — i.e. systems that require a state model to enforce specific behaviours — require specific components or heuristics to manage their statefulness. For example, Daniele et al. [10] identified seven different categories of stateful fuzzers that implement such ad-hoc components to deal with the stateful nature of the SUT. Among others, they describe the *evolutionary grammar-based* fuzzers. These fuzzers combine a feedback mechanism (similar to AFL++) with the SUT's state model specification. In fact, they use the feedback information to steer the generation of the messages toward the most interesting ones and the knowledge of the state model to explore different states. LibAFLstar falls in this category.

On a higher level, the main problem with stateful fuzzing is the need to deal with two different layers of inputs. In fact, while stateless systems only have to fuzz the messages; stateful systems need to fuzz both the messages and their order. Mutating the message order introduces additional complexity, as the same messages can be sent from any state.

[5] `https://google.github.io/oss-fuzz/`
[6] `https://llvm.org/docs/LibFuzzer.html`
[7] `https://github.com/google/honggfuzz`

3 Challenges in Stateful Fuzzing

Stateful fuzzing is generally more challenging than stateless fuzzing, and using a stateless fuzzer to fuzz stateful systems is often not the best choice [10]. Specifically, four main challenges arise when fuzzing stateful systems:

C1 — State model exploration. Any well-designed stateful fuzzer must be able to explore the state space. In other words, the fuzzer needs to move from one state to another. This ability requires either i) the actual state model that the SUT implements or ii) the ability of the fuzzer to infer the state model of the SUT during the fuzzing campaign. Unfortunately, protocol specifications usually contain scattered information about the state model through different pages of prose, and stateful fuzzers struggle to infer a good approximation of the SUT state model. For example, AFLNet [27] infers the state model of the SUT at run time by observing *only* the response of the server. Despite this approach giving a good understanding of the state model the SUT implements, it might result in an inaccurate understanding. Different server responses might not trigger any transition state, and two identical server responses can instead cause a state transition.

C2 — States prioritization. Given the large number of states that certain programs implement and the limited time availability, efficient stateful fuzzers need to prioritize the most interesting states to fuzz. While a naive approach that prioritizes states that trigger new coverage may work for some systems, it may perform poorly for others, potentially overlooking interesting states. Advanced stateful fuzzers should be able to schedule states according to their probability of covering a big portion of code, i.e., finding more bugs. For example, during the testing of an FTP implementation, it is not ideal to spend much time fuzzing the first state, as many commands are only available after a successful authentication.

C3 — Dependencies between messages and states. Stateless systems have no notion of previous executions: they just take in input one message at a time. For this reason, stateless fuzzers use a *global queue* to store messages that are interesting for the system as a whole. On the contrary, when fuzzing stateful systems, certain messages are interesting only in some states, i.e., after other messages. Despite fuzzers like AFLNet or ChatAFL [25] partially solving this problem by concatenating multiple messages in a single trace, this approach requires the restart of the SUT after every trace. This causes serious performance limitations, as explained in the next paragraph.

C4 — SUT restarts and synchronizations. SUT restarts represent a challenge for both stateless and stateful fuzzing. The time spent to restart the target system significantly reduces the number of inputs that can be processed in a given time budget. This makes fuzzers less efficient. Moreover, stateful systems often implement more complex behaviours, making the restart overhead even more expensive. Worse, when fuzzing stateful protocols that require communication between a client and a server, the synchronization between the two sides also plays a crucial role. When fuzzing a server, stateful fuzzers cannot send

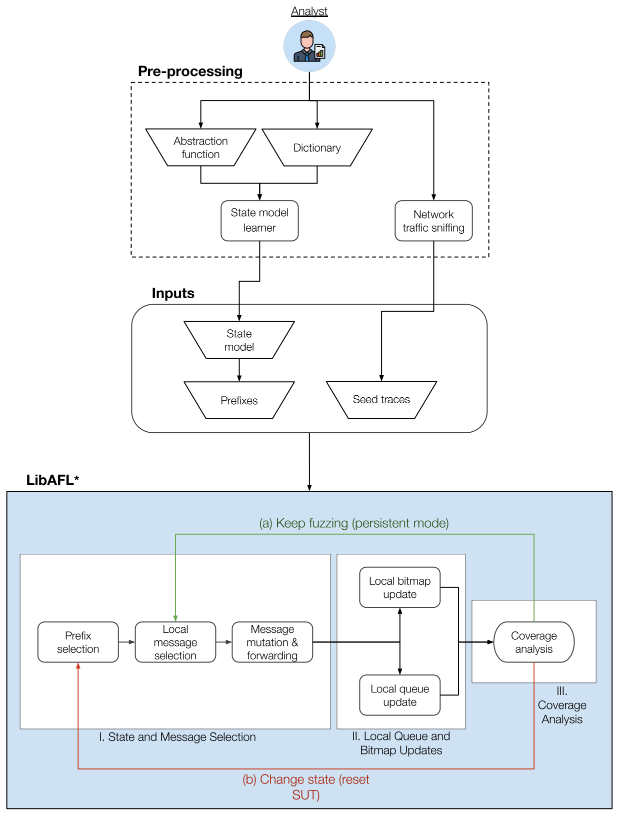

Fig. 1: LibAFLstar workflow. The approach takes in input the state model of the SUT and some seed traces. It then selects a state and keeps fuzzing it by sending malformed messages. It switches state when no more coverage is triggered.

millions of messages sequentially; they must allow the server to process and acknowledge each message.

In Section 4, we address these challenges and propose a set of new techniques to design a fast and state-aware fuzzer for protocol implementations.

4 LibAFLstar: Methodology & Design

In this section, we introduce LIBAFLSTAR, a state-aware fuzzer that addresses the four challenges presented in Section 3.

Our fuzzer takes in input seed traces for the generation of the inputs and the state model of the SUT to create the *prefixes*, i.e., a set of partial sequences of messages.[8]

LibAFLstar first sends a prefix to reach a target state, then appends a mutated message — derived from the sample traces — deemed interesting for that state. For our experiments, we obtain the sample traces by sniffing network traffic (like AFLNet and ChatAFL do) and the prefixes using a black-box state learning technique [28], since the official FTP specification does not provide an exhaustive state model description. More in detail, we leverage the learning techniques to infer the state model of the SUT and then determine prefixes for reaching each state.

The overview of the framework (presented in Figure 1) consists of the following steps:

I. State and Message Selection. The state scheduler chooses a state to fuzz (*prefix selection* in Figure 1), and a message scheduler selects the message to mutate from the *local queue* of the selected state (*local message selection* in Figure 1). In fact, unlike other approaches that keep only one global queue of interesting messages [31, 20], LibAFLstar stores a queue of relevant messages for *each* state. This allows the fuzzer to select messages that are interesting for specific states, as detailed in Section 4.2.

II. Local Queue and Bitmap Updates. While fuzzing the chosen state, the code coverage is collected in the local bitmap, and inputs deemed interesting are inserted in the local message queue, as explained in Section 4.3.

III. Coverage Analysis. LibAFLstar observes the coverage and decides whether to (a) continue fuzzing the same state (green line in Figure 1) or (b) select a new state (red line in Figure 1), as detailed in Section 4.4.

In the next sections, we expand on each of these phases with a running example that simulates the steps that LibAFLstar performs to fuzz LightFTP, an FTP server commonly used to benchmark stateful fuzzers. In Section 5, we describe the implementation details of our prototype.

4.1 Preprocessing

Our approach, as any other mutation-based fuzzer (including AFLNet and ChatAFL), needs seed traces to bootstrap the fuzzing campaign. We collect these traces beforehand by sniffing the communication between the server and a client during legitimate protocol usage. As shown in prior work, the quality of the sniffed traffic strongly influences the generation of the seed traces and, therefore, the fuzzing performance [29]. We limit the risk of collecting incomplete or inaccurate network traffic by monitoring the network traffic between a client (executed by three different analysts) and the server for one hour in total.

[8] Every prefix unambiguously identifies a certain state since stateful systems are usually deterministic.

LibAFLstar also needs the *prefixes* to be able to explore the state model. For instance, when fuzzing LightFTP (state model in Figure 2), the prefixes required to reach all the states are:

P_1: *USER ubuntu* (to reach the state S_1 – allowing the user to insert the password)
P_2: *USER ubuntu, PASS ubuntu* (to reach the state S_2 – allowing the user to login)
P_3: *USER ubuntu, PASS ubuntu, list* (to reach the state S_3 – allowing the user to show the repositories in the server)
P_4: *USER ubuntu, PASS ubuntu, epsv* (to reach the state S_4 – allowing the user to enter in passive mode)

Retrieving the prefixes to reach each state is challenging, as the FTP specification lacks a defined state model. In addition, the actual implementation might adopt a slightly different version of the state model presented in the specification.

For our experiments, we infer the state model of live555 and Lighttpd from the specification and the state models of the different FTP implementations using the active learner tool LearnLib [33]. Other active learner tools are available and actively maintained[9] Interestingly, the four FTP implementations implement slightly different state models.

Active learning tools [33] use an *initial alphabet* to *actively* query the SUT with all the possible combinations of the *commands* in the initial alphabet. They observe (in a black-box fashion) requests and responses and infer the state model by improving their knowledge of the state model via counterexamples. Every time the tool finds an example that does not fit its assumption, it improves its knowledge of the state model. The approach is methodical and precise, although sometimes slow.

Since active learning tools struggle with non-deterministic behaviours, we developed an ad-hoc harness to address them. For example, by handling all the requests that triggered a timeout.

Additionally, LearnLib requires *abstraction functions* (one for requests and one for responses) to map raw messages into learner-compatible inputs. For example, the raw messages "*USER wrong_user1*" and "*USER wrong_user2*" are mapped to the same input "*USER wrong_user*" to limit the state explosion.

Writing the harness and the abstraction functions required around two hours of work.

Figure 2 shows the state model produced by LearnLib, which we use to extract the prefixes to reach every state.

It is worth it to mention that active learning provides the *best* possible list of prefixes. In fact, it provides all the prefixes to reach all the states. Simpler (and faster) approaches may involve extracting prefixes directly from network traffic or manually inferring the state model from the protocol specification. For example, the seed file used by AFLNet[10] {*USER ubuntu, PASS ubuntu,*

[9] `https://des-lab.github.io/AALpy/`
[10] `https://github.com/aflnet/aflnet/blob/master/tutorials/lightftp/in-ftp/ftp_requests_full_normal.raw`

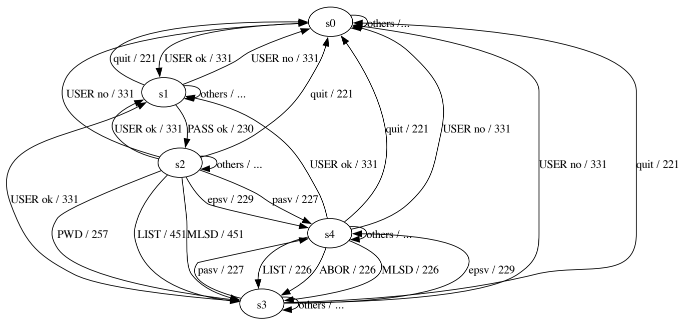

Fig. 2: LightFTP state model, inferred via LearnLib.

SYST, PWD, PORT 127,0,0,1,132,209, LIST, MKD test, QUIT} — obtained by sniffing the network traffic — would already cover 80% of the states. In fact, it would cover all the states but the state $S4$ in Figure 2, reachable only by sending the message $PASV$ or $EPSV$ from the states $S2$ or $S3$.

Using a state model in input allows LibAFLstar to extract the prefixes used to effectively explore the state model of the SUT, addressing the challenge C1.

4.2 State and Message Selection

State Selection LibAFLstar selects promising states – in the sense of having a high probability of covering large portions of code – using the Outgoing Edges (OE) state scheduler. The scheduler prioritizes states with a higher number of outgoing edges. The logic behind this heuristic is that these states likely implement complex functionality, making them ideal fuzzing candidates. For the sake of comparison, we also implement a naive state scheduler, Round Robin (RR), which simply selects the target states in turn. The ability to focus on the most promising states addresses the challenge C2.

Message Selection LibAFLstar inherits LibAFL's message selection strategy. More in detail, LibAFLstar saves the messages that trigger new code coverage in the state's local queue and mutates them until no more coverage is discovered. It is worth mentioning that the queue is never empty, as the messages that do not discover new coverage are not removed from the queue but just moved to the end.

4.3 Local/Global Queues and Bitmaps

As already mentioned in Section 3, a message can be interesting for one state but not interesting for others. Thus, when fuzzing stateful systems, using a global queue for every state is not the best choice, as it fails to capture input-state dependencies.

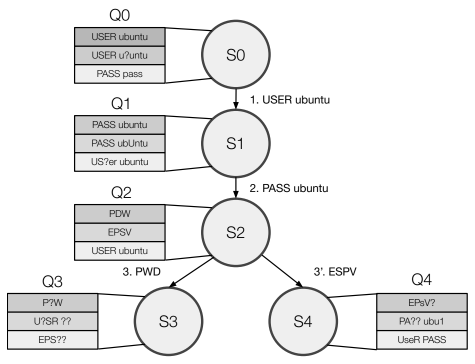

Fig. 3: LibAFLstar implements different queues for different states to be able to mutate only the messages deemed interesting for that specific state.

For instance, while fuzzing LightFTP, sending commands that require prior authentication before the user is authenticated triggers an uninteresting error message. Similarly, when fuzzing stateful systems, the feedback information (recorded in bitmaps) should be associated with individual states rather than the system as a whole. For these reasons, LibAFLstar can also work with local message queues and bitmaps for each state.

Figure 3 shows LibAFLstar knowledge (state model and queues) while fuzzing LightFTP. In this example, the messages *PWD* and *EPSV* are interesting only when sent from state $S2$ and not from state $S0$. On the other hand, the message *USER ubuntu* is interesting in the states $S0$ and $S2$. In fact, despite messages that are interesting in one state might not be in another, a few inputs might be interesting for multiple states. Having the same message in multiple queues is not possible while sharing bitmap information, as, after the first time, interesting messages are not recognized as new. We solved this issue by allowing LibAFLstar to have a bitmap for each state. This approach enables more fine-grained coverage tracking, allowing for precise queue management.

The ability of the fuzzer to keep state-related queues and bitmaps addresses challenge C3.

4.4 Coverage Analysis

After sending a message, LibAFLstar can either (1) continue fuzzing without restarting the SUT (green arrow in Figure 1) or (2) select a new state to fuzz (red arrow in Figure 1).

We efficiently implement these two mechanisms by using the *AFL++ persistent mode*, which allows LibAFLstar to send many messages to the SUT without restarting it. It is worth highlighting that the persistent mode was developed to fuzz state**less** systems exclusively — since there is no point in restarting the SUT if the behaviour does not change after restarts. On the contrary, LibAFLstar uses the persistent mode to fuzz state**ful** systems efficiently.

```
int state=0;
//while (client\_connected()){
while (AFL_LOOP<UINT_MAX}{
  command=receive_command();
  response=execute_command(command);
  send_response(response);
 }
```

Fig. 4: Modification on the SUT code to enable the persistent mode. The example presents the pseudo-code and not the actual LightFTP code. The original code is commented out, the code needed to run LibAFLstar is in blue.

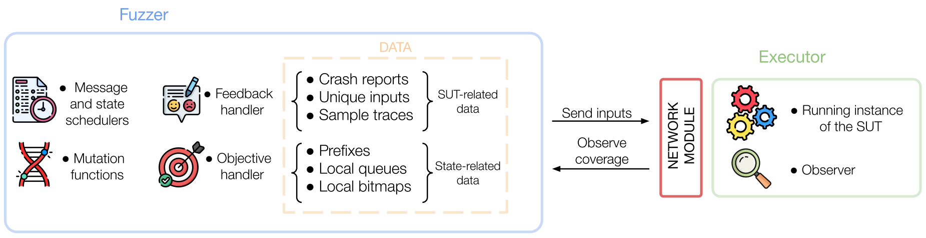

Fig. 5: LibAFLstar architecture.

The persistent mode consists of a special loop that allows the fuzzer to cycle a specific portion of the code without any restart. LibAFLstar leverages this mechanism to cycle the portion of code that handles the commands to send multiple messages to the same instance of the SUT. To enable the persistent mode, it is necessary to modify the code of the SUT to add this loop. For example, patching the LightFTP server requires the modification shown in Figure 4.

Moreover, the persistent mode solves the synchronization overhead. Using the persistent mode allows our fuzzer to know when the SUT has finished its computation — thanks to a pipe message sent by the SUT to the fuzzer. This allows the fuzzer to send hundreds of messages per second, without worrying about overloading the SUT. As we discuss more extensively in Section 8, sending multiple messages without restarting the SUT might cause unexpected state transitions. In fact, some messages sent from a certain state might trigger a transition from the current state to a new one.

The persistent mode addresses challenge C4.

5 Implementation Details

LibAFLstar is written in Rust and built on top of LibAFL. As shown in Figure 5, it consists of three components:

1. the *Fuzzer*: it is the core of LibAFLstar as it incorporates LibAFLstar logic. It contains the standard LibAFL scheduler algorithms for the messages

and the new scheduler algorithms for the states (Round Robin and Outgoing Edges), introduced in Section 4.2. Additionally, it supports standard AFL++ mutation functions, such as havoc mutations. Moreover, it implements the standard *feedback handler*, i.e., a function that determines when an input is interesting and the standard *objective handler*, i.e., a function that determines if an input achieved the pre-determined goal — in our case, the crash of the system. All the non-volatile data used during the fuzzing campaign are stored in the Fuzzer. It stores data about crashes found, input that triggered the crashes, and the sample traces. Moreover, it stores all the data that are state-related, i.e., prefixes, local queues and local bitmaps;

2. the *Executor*: it manages everything regarding the SUT execution. More in detail, it handles the SUT initialization and restart, and contains the *Observer*, which monitors the SUT execution by tracking the code coverage and system behaviour.
3. the *Network Module*: it allows LibAFLstar to communicate over network sockets. This module allows LibAFLstar to operate in both client and server modes, enabling it to fuzz either side as needed.

The main modification we made in LibAFL was to add a loop to the actual LibAFL one. This loop enables LibAFLstar to select a state via the state schedulers, reset the SUT to its initial state, and send the prefix to reach the target state. Once the prefix (and thus the state) is selected, LibAFLstar fetches a message from the local queue of the selected state, mutates and forwards it to the SUT, and observes the coverage.

6 Experimental Results

We evaluated our approach on six real-world protocol implementations, namely LightFTP, BFTPD, ProFTPD, Pure-FTPd, Lighttpd, and live555. Our experiments are divided into two parts: an ablation study to assess the impact of different configuration settings (Section 6.1) and a comparative evaluation against other state-of-the-art stateful fuzzers (Section 6.2). We ran our experiments on a VPS equipped with an Intel Xeon (Ice Lake) processor (16 cores at 2.6 GHz) and 64 GB of DDR4 RAM.

6.1 Ablation Study

To investigate how different configurations affect performance and effectiveness, we conducted 24 experiments for each case study, with each run lasting one hour. To remove any randomness involved, we re-ran all the experiments three times and observed the average. LibAFLstar can be configured with:

1. A state scheduling algorithm: round-robin (RR) or outgoing-edge-based (OE);
2. Either global or local queues and bitmaps;
3. A persistent loop of configurable length, where a length of 1 disables persistent mode.

Table 1: LibAFLstar ablation study for one hour. SS = State Scheduler; LL = loop length (* with 1 meaning no persistent mode enabled) B = Bitmap; Q = Queue; EC = Edge Coverage (in %); E = Number of executions (in thousands); G = Global; L = Local.

SS	LL	B	Q	BFTPD		LightFTP		lighttpd		live555		ProFTPD		PureFTPD	
				EC	**E**	**EC**	**E**	**EC**	**E**	**EC**	**E**	**EC**	**E**	**EC**	**E**
Round robin	1*	G	G	13.64	47	28.79	132	3.86	239	4.61	201	9.46	39	7.44	4.92
	10	G	G	16.62	66	32.81	459	4.62	1k	4.72	316	10.66	75	**8.29**	2.58
	100	G	G	17.19	365	34.38	2k	5.37	4k	4.85	329	11.65	465	7.47	2.28
	1000	G	G	–	–	34.38	4k	5.12	9k	4.61	303	12.01	3k	7.47	2.28
	1*	G	L	13.92	46	28.35	155	3.85	234	4.53	237	–	–	7.44	4.92
	10	G	L	16.62	84	32.59	557	4.64	1k	4.87	361	–	–	7.47	2.58
	100	G	L	17.19	316	34.38	3k	4.67	4k	4.79	572	–	–	7.54	2.28
	1000	G	L	18.47	2k	34.38	5k	4.87	10k	4.70	390	–	–	7.47	2.24
	1*	L	L	14.06	47	28.91	155	4.26	234	4.41	221	9.73	40	7.44	4.92
	10	L	L	16.12	71	33.04	554	4.69	1k	4.71	437	10.93	77	7.57	2.57
	100	L	L	16.69	429	34.38	2k	5.39	4k	4.85	538	11.44	455	7.57	2.30
	1000	L	L	**21.16**	3k	34.38	5k	5.18	9k	4.71	498	12.12	3k	7.64	2.24
Outgoing edges	1*	G	G	13.64	46	28.91	143	4.00	240	4.41	207	10.39	40	7.54	**5.20**
	10	G	G	16.69	71	33.26	466	4.51	1k	4.70	290	10.64	77	7.47	2.59
	100	G	G	16.83	379	34.49	2k	5.49	4k	4.81	417	11.62	427	7.57	2.30
	1000	G	G	18.96	2k	34.49	5k	5.09	8k	4.87	435	12.07	3k	7.47	2.26
	1*	G	L	14.20	46	30.13	169	4.06	235	4.41	237	9.22	1	7.44	5.11
	10	G	L	16.48	71	32.70	565	4.48	1k	4.85	413	9.26	1	7.57	2.59
	100	G	L	16.76	286	34.49	3k	4.71	4k	4.82	**591**	–	–	7.47	2.30
	1000	G	L	17.97	2k	34.38	5k	5.01	9k	4.84	384	–	–	7.87	2.22
	1*	L	L	14.06	46	29.69	169	4.24	236	4.48	229	10.27	41	7.44	5.21
	10	L	L	16.69	71	33.04	560	4.71	1k	4.77	435	10.90	77	7.57	2.56
	100	L	L	16.83	379	34.38	3k	5.45	4k	4.83	489	11.45	444	**7.87**	2.32
	1000	L	L	18.96	2k	**34.38**	4k	**5.60**	9k	**4.87**	464	**12.17**	3k	7.57	2.26

The corresponding results are shown in Table 1.

State Schedulers Evaluation. As mentioned in Section 4.2, LibAFLstar implements two state scheduler algorithms: RR and OE. Although the experiments do not show much difference between the two approaches, this may be due to the low complexity of the state model implemented by the protocols. Nevertheless, a closer look at the code coverage suggests *OE* to perform slightly better.

Global and Local Queues Evaluation As mentioned in Section 3, local queues give more information about the quality of the messages for a certain state. The experiments show that local queues and local bitmaps, on average, give the best results in terms of coverage.

Persistent Loop Length Evaluation As mentioned in Section 4.4, the length of the persistent loop tells the fuzzer how many messages to send before restarting the SUT. As shown in Figure 6, longer persistent loops result in a higher

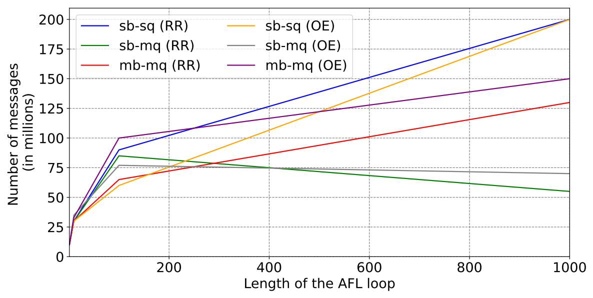

Fig. 6: Relation between the length of the AFL loop and the number of messages sent.

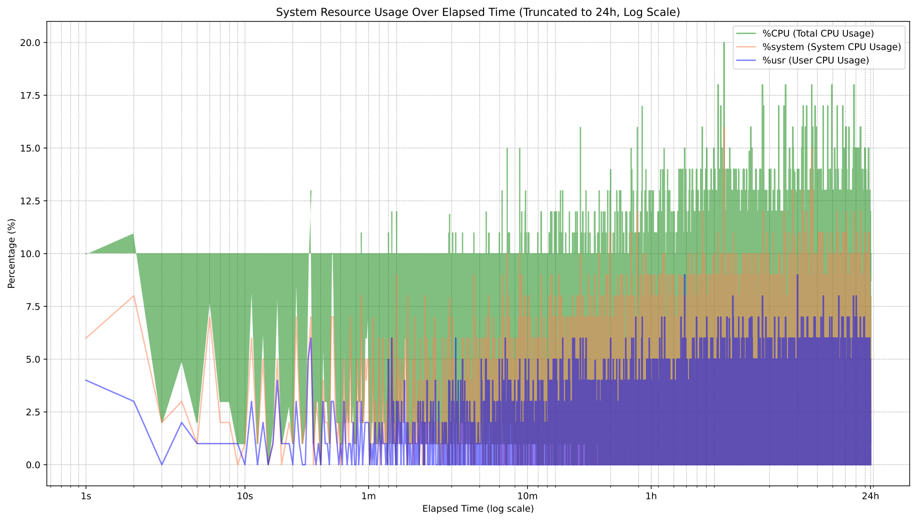

Fig. 7: CPU usage of LibAFL* testing LightFTP.

number of messages sent. The results show that the length of the persistent loop is proportional to the number of executions, as bigger loops imply less time wasted in restarting the SUT. In fact, as explained in [2], resets are one of the biggest overheads when fuzzing stateful systems. Further experiments were conducted to determine whether bigger loops would have improved the quality of the fuzzing campaign. However, we did not notice any improvement for a length bigger than 1000.

Network Overhead Evaluation Network protocols can add significant overhead due to network system calls [2]. We analyzed the network overhead by monitoring the user and system CPU usage during the fuzzing campaign and noted that the system calls overhead is about 50%, as shown in Figure 7. We de-

Table 2: Results of 24 hours of fuzzing. We highlighted in bold the best results. Na means we used the specification to infer the state model of the SUT.

	AFLNet			**ChatAFL**			**LibAFLstar**					
Subject	code coverage (%)	No. Traces (k)	No. Traces/sec.	code coverage (%)	No. Traces (k)	No. Traces/sec.	code coverage (%)	No. Messages (kk)	No. Mess./sec.(~)	Setup persistent mode	Setup LearnLib (minutes)	LearnLib learning time (minutes)
LightFTP	32.4	305	3.5	34.7	250	2.8	**38.1**	67	775	1h	30	30
BFTPD	17.5	233	2.7	20.4	239	2.7	**25.4**	56	648	1h	30	30
ProFTPD	15.3	162	3.4	**17.4**	164	1.8	17.2	44	509	1h	30	45
PureFTPd	4.8	297	3.4	4.7	138	1.6	**14.1**	85	983	1h	30	30
Lighttpd	5.4	209	2.4	5.3	241	2.8	**5.9**	22	254	30m	Na	Na
live555	5.1	839	10.14	5.2	667	8.6	**5.8**	7	103	2h	Na	Na

cided to fuzz the SUTs without replacing the network calls to be fair to AFLNet and ChatAFL.

6.2 Comparison with AFLNet and ChatAFL

For the benchmark, we compare LibAFLstar (using local bitmaps and queues, the persistent loop length of 1000 and the outgoing edge state scheduler algorithm) with two state-of-the-art stateful fuzzers, against AFLNet and ChatAFL. Due to fuzzing nondeterminism, this comparison was averaged across three runs of 24 hours. We selected AFLNet [27] since it is one of the most popular mutation-based open-source protocol fuzzers and ChatAFL [25] as it is one of the most recent stateful fuzzers published in a top conference. In the future, we plan to extend our benchmarking to SGPFuzzer [35] and Stateful Greybox Fuzzing [4].

Number of messages per second. Table 2 shows how LibAFLstar sends many more messages than AFLNet and ChatAFL. This is due to the fact that LibAFLstar drastically reduces the number of resets thanks to the persistent loop. Also, we attribute the huge improvement in terms of the number of messages to the very low impact on the performance of the synchronization between the fuzzer and the client (as explained in Section 4.4). While LibAFLstar uses the persistent mode mechanism to synchronize the SUT and the fuzzer, the other fuzzers wait for the responses to be received. It is worth highlighting that AFLNet and ChatAFL record the number of traces, not the single messages sent. A fair comparison with LibAFL* requires multiplying the *number of traces* and *number of traces per second* by the length of the trace — typically five or six messages. However, this adjustment does not take away the fact that LibAFLstar is an order of magnitude faster than its competitors.

Total Coverage. LibAFLstar achieves higher coverage on all the case studies, except ProFTPD (-1%). In all the other case studies, LibAFLstar shows a significant improvement in terms of code coverage. More in detail, LibAFLstar achieves, on average, 48.6% more coverage than AFLNet and 42.7% more coverage than ChatAFL.

7 Related Work

State awareness. A few papers have already explored the correlation between fuzzing and state learning. For example, De Ruiter et al. [11] used LearnLib to find logical bugs in TLS implementations; Bastani et al, [5] devised their active learner to synthesize a grammar to give to the fuzzer; Van-Thuan et al. [27], Yingchao Yu et al.[35] and Doupé et al.[12] developed algorithms to infer the state model of the SUT run-time by observing the responses of the (web) servers. Nevertheless, despite all the approaches dealing with the statefulness of the systems and partially solving Challenge C1 (Section 3), they do not implement schedulers to prioritize the most interesting states. Moreover, all stateful fuzzers (except those using snapshotting) rely on expensive resets to send fresh traces (challenge C2) and use timeouts to synchronize the server and client (challenge C3), which leads to poor performance.

State schedulers. Fuzzers like AFLNet prioritize states based on the number of newly discovered edges; however, to the best of our knowledge, no stateful fuzzer employs an Outgoing Edges (OE) heuristic to prioritize interesting states. Nevertheless, the approach is related to PageRank [6], the search algorithm Google uses to measure the importance of web pages. PageRank scores the web pages according to the number of external websites that point to that page (ingoing edges). The heuristic is that *good* websites are likely to be linked more often from external websites. A similar heuristic applies to our OE algorithm: "states with many outgoing edges likely implement many features and therefore contain a lot of lines of code. This makes these states honeypots for fuzzers."

Persistent mode for stateful fuzzing. LibAFLstar is the first fuzzer that leverages the persistent mode to mitigate the overhead of shutting down and restarting the SUT.

Snapshotting is another technique that tries to solve the same challenge differently. Some stateful fuzzers [31, 18, 20] copy the memory of the SUT in a certain state to be able to go back to the same state later. This allows the fuzzer to reach one specific state more quickly in the future — via the snapshot previously created — and fuzz it. In the same way, the persistent mode allows the fuzzer to reach one specific state — just by sending the correct prefix — and fuzz it. Unfortunately, snapshotting often introduces considerable overhead that often makes sending the whole trace again less expensive.

8 Limitations and Future Work

LibAFLstar requires the source code to be patched to enable the persistent mode. We noticed that the majority of the stateful protocols contain a single send-receive loop that processes the commands. In this scenario, the main loop is easy to spot and patch to enable the persistent mode. Moreover, the patch usually involves a few lines of code: 7 changes in the best scenario (ProFTPD) and 35 changes in the worst one (live555). However, in other cases, identifying such a loop can be more challenging and might require a deeper knowledge of the protocol logic. For example, OpenVPN [14], a widely used VPN implementation, is event-driven. This makes identifying the command loop much more challenging. Also, when triggering a new code coverage in a certain state (that we reached thanks to a specific prefix), we do not know whether the new coverage is triggered in that particular state. In fact, *unexpected* state transitions might have occurred before, as mentioned in Section 4.4. The persistent mode can also limit the reproducibility of detected bugs. To solve this problem, we save all the messages sent during the persistent mode loop that triggered a bug.

Future research could explore alternative scheduling algorithms to prioritize interesting states. For example, the technique devised by Liyanage et al. [22] can be used to predict how likely a certain state would lead to new edge discovery. Eventually, we plan to extend our validation to other case studies to assess the effectiveness of our approach. One solution can be to implement LibAFLstar within ProFuzzBench [26], a widely used benchmark for stateful fuzzers, to have a more systematic and wide understanding of the effectiveness of our fuzzer. As already mentioned in Section 6.1, network calls might slow down the fuzzer's performance. Libraries like PREENY [11], Green-Fuzz [1], or Sabre [3] might help here to reduce the network overhead and additionally enhance the fuzzer performance.

9 Conclusion

In this paper, we presented LibAFLstar, a fast and state-aware protocol fuzzer designed to address the challenges of fuzzing stateful systems. LibAFLstar uses the notion of prefixes to navigate the state model, implements the outgoing edge state scheduler to prioritize states that most likely cover big portions of code, implements different queues and bitmaps for different states, and uses the persistent mode (i.e., the ability to keep the SUT running between traces) to reduce the number of restarts of the SUT. The ablation study confirms that all the above-mentioned strategies enhance the fuzzer performance. In fact, LibAFLstar has

[11] `https://github.com/zardus/preeny`

the best results by using the outgoing edge scheduler algorithm, local queues, local bitmaps and the persistent mode. Also, the ablation study shows that the biggest improvement is given by the use of the persistent mode and that longer persistent loops positively affect the fuzzer speed.

We evaluated LibAFLstar on six protocol implementations (LightFTP, BFTPD, ProFTPD, Pure-FTPd, Lighttpd and live555) and compared the results with AFLNet and ChatAFL. The results show that LibAFLstar is over 30× faster than its competitors while achieving, on average, 1.4× more coverage.

Data Availability

In the spirit of open science, we release the code for this project: `https://github.com/LibAFLstar/LibAFLstar`.

Acknowledgments

We would like to thank our reviewers for their valuable inputs. We would also like to express our gratitude to the GARR Consortium for providing us with the cloud infrastructure necessary for the execution of the large-scale experiments. In addition, gratitude is extended to Leonardo Cantarella for his contribution to the setup of the Lighttpd case study. This work has been supported by the INTERSECT project (NWA.1160.18.301), funded by the Dutch Research Council (NWO), by the P6 project (Open Technology Programme No. 20475) funded by the Dutch Research Council (NWO), and by the Dutch Ministry of Economic Affairs and Climate Policy (EZK) through the AVR project "FirmPatch".

References

1. Andarzian, S. B., Daniele, C., and Poll, E. Green-fuzz: Efficient fuzzing for network protocol implementations. In *International Symposium on Foundations and Practice of Security* (2023), Springer, pp. 253–268.
2. Andarzian, S. B., Daniele, C., and Poll, E. On the (in) efficiency of fuzzing network protocols.
3. Arras, P.-A., Andronidis, A., Pina, L., Mituzas, K., Shu, Q., Grumberg, D., and Cadar, C. Sabre: load-time selective binary rewriting. *International Journal on Software Tools for Technology Transfer 24*, 2 (2022), 205–223.
4. Ba, J., Böhme, M., Mirzamomen, Z., and Roychoudhury, A. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)* (2022).
5. Bastani, O., Sharma, R., Aiken, A., and Liang, P. Synthesizing program input grammars. *ACM SIGPLAN Notices* (2017).
6. Bianchini, M., Gori, M., and Scarselli, F. Inside pagerank. *ACM Transactions on Internet Technology (TOIT)* (2005).
7. Böhme, M., Cadar, C., and Roychoudhury, A. Fuzzing: Challenges and reflections. *IEEE Software* (2020).

8. Chen, J., Diao, W., Zhao, Q., Zuo, C., Lin, Z., Wang, X., Lau, W. C., Sun, M., Yang, R., and Zhang, K. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS* (2018).
9. Chen, Y., Lan, T., and Venkataramani, G. Exploring effective fuzzing strategies to analyze communication protocols. In *ACM Workshop on Forming an Ecosystem Around Software Transformation* (2019).
10. Daniele, C., Andarzian, S. B., and Poll, E. Fuzzers for stateful systems: Survey and research directions. *ACM Computing Surveys* (2024).
11. De Ruiter, J., and Poll, E. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium* (2015).
12. Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium* (2012).
13. Eisele, M., Maugeri, M., Shriwas, R., Huth, C., and Bella, G. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity* (2022).
14. Feilner, M. *OpenVPN: Building and integrating virtual private networks.* Packt Publishing Ltd, 2006.
15. Feng, X., Sun, R., Zhu, X., Xue, M., Wen, S., Liu, D., Nepal, S., and Xiang, Y. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *ACM SIGSAC Conference on Computer and Communications Security* (2021).
16. Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. Afl++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)* (2020).
17. Fioraldi, A., Maier, D. C., Zhang, D., and Balzarotti, D. Libafl: A framework to build modular and reusable fuzzers. In *ACM SIGSAC Conference on Computer and Communications Security* (2022).
18. Geretto, E., Giuffrida, C., Bos, H., and Van Der Kouwe, E. Snappy: Efficient fuzzing with adaptive and mutable snapshots. In *Computer Security Applications Conference* (2022).
19. Kim, K., Jeong, D. R., Kim, C. H., Jang, Y., Shin, I., and Lee, B. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS* (2020).
20. Li, J., Li, S., Sun, G., Chen, T., and Yu, H. Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. *IEEE Transactions on Information Forensics and Security* (2022).
21. Li, J., Zhao, B., and Zhang, C. Fuzzing: a survey. *Cybersecurity* (2018).
22. Liyanage, D., Lee, S., Tantithamthavorn, C., and Böhme, M. Extrapolating coverage rate in greybox fuzzing. In *IEEE/ACM International Conference on Software Engineering* (2024).
23. Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).
24. Mathis, B., Gopinath, R., Mera, M., Kampmann, A., Höschele, M., and Zeller, A. Parser-directed fuzzing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019).
25. Meng, R., Mirchev, M., Böhme, M., and Roychoudhury, A. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)* (2024).
26. Natella, R., and Pham, V.-T. Profuzzbench: A benchmark for stateful protocol fuzzing. In *ACM SIGSOFT international symposium on software testing and analysis* (2021).

27. Pham, V.-T., Böhme, M., and Roychoudhury, A. Aflnet: a greybox fuzzer for network protocols. In *International Conference on Software Testing, Validation and Verification (ICST)* (2020), IEEE.
28. Raffelt, H., Steffen, B., and Berg, T. Learnlib: A library for automata learning and experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems* (2005).
29. Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., and Brumley, D. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium* (2014).
30. Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. kafl: Hardware-assisted feedback fuzzing for os kernels. In *26th USENIX Security Symposium* (2017).
31. Schumilo, S., Aschermann, C., Jemmett, A., Abbasi, A., and Holz, T. Nyx-net: network fuzzing with incremental snapshots. In *European Conference on Computer Systems* (2022).
32. Serebryany, K. Oss-fuzz: Google's continuous fuzzing service for open source software.
33. Settles, B. Active learning literature survey.
34. Srivastava, P., and Payer, M. Gramatron: Effective grammar-aware fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021).
35. Yu, Y., Chen, Z., Gan, S., and Wang, X. Sgpfuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access* (2020).
36. Zalewski, M. American Fuzzy Lop - Whitepaper. `https://lcamtuf.coredump.cx/afl/technical_details.txt`, 2016.
37. Zhang, Z., Zhang, H., Zhao, J., and Yin, Y. A survey on the development of network protocol fuzzing techniques. *Electronics* (2023).
38. Zhu, X., Wen, S., Camtepe, S., and Xiang, Y. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* (2022).