Empirical Software Engineering (2022) 27:191
https://doi.org/10.1007/s10664-022-10233-3

StateAFL: Greybox fuzzing for stateful network servers

Roberto Natella[1]

Accepted: 28 August 2022

Abstract
Fuzzing network servers is a technical challenge, since the behavior of the target server depends on its state over a sequence of multiple messages. Existing solutions are costly and difficult to use, as they rely on manually-customized artifacts such as protocol models, protocol parsers, and learning frameworks. The aim of this work is to develop a greybox fuzzer (StateAFL) for network servers that only relies on lightweight analysis of the target program, with no manual customization, in a similar way to what the AFL fuzzer achieved for stateless programs. The proposed fuzzer instruments the target server at compile-time, to insert probes on memory allocations and network I/O operations. At run-time, it infers the current protocol state of the target server by taking snapshots of long-lived memory areas, and by applying a fuzzy hashing algorithm (Locality-Sensitive Hashing) to map memory contents to a unique state identifier. The fuzzer incrementally builds a protocol state machine for guiding fuzzing. We implemented and released StateAFL as open-source software. As a basis for reproducible experimentation, we integrated StateAFL with a large set of network servers for popular protocols, with no manual customization to accomodate for the protocol. The experimental results show that the fuzzer can be applied with no manual customization on a large set of network servers for popular protocols, and that it can achieve comparable, or even better code coverage and bug detection than customized fuzzing. Moreover, our qualitative analysis shows that states inferred from memory better reflect the server behavior than only using response codes from messages.

Keywords Security · Fuzzing · Network servers

1 Introduction

According to recent statistics (Hawkes 2019; O'Neill 2021), high-severity software vulnerabilities of network servers have been on the rise, and will likely still be in the near future. Network servers are a critical part of the attack surface of IT infrastructures, as they are openly exposed to malicious users over local networks and the Internet, and can be attacked

Communicated by: Paolo Tonella

✉ Roberto Natella
roberto.natella@unina.it

[1] Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Napoli, Italy

with malformed traffic to cause a denial-of-service (e.g., crashing the server), and to execute arbitrary code on the server machine to perpetrate further attacks. For this reason, any vulnerability not yet found by developers (*0-days*) has a significant economic value for attackers (Guo et al. 2021).

Fuzzing is a relevant security testing technique to identify such vulnerabilities, by automatically generating large volumes of malformed inputs. However, fuzzing network servers is still a technical challenge, since the input space of network servers is strictly regulated by a *stateful protocol*. Therefore, the behavior of the server, and its vulnerabilities, depend on a sequence of several messages exchanged over time, which determine the *state* of the server. Examples of well-know stateful protocols include cryptographic ones such as TLS (De Ruiter and Poll 2015; Fiterau-Brostean et al. 2020), file transfer and messaging protocols such as FTP, SMB, and SMTP (Antunes and Neves 2011; Comparetti et al. 2009), and multimedia protocols such as SIP (Banks et al. 2006; Alrahem et al. 2007) and RSTP (Pham et al. 2020). All of these protocols are selective with respect to which messages they can receive at a given time, and which actions they can perform, depending on previous messages in a session.

The existing stateful protocol fuzzing techniques and tools can only be applied with a significant effort, which has prevented their widespread adoption so far: *generation-based* fuzzers require formal specifications manually written by human experts, based on their detailed knowledge of the protocols (Beyond Security 2020; Synopsis Inc 2020; Rapid7 2020); *learning-based* fuzzers infer the protocol state machine at a significant computational cost, and still require custom implementations of wrappers to abstract protocol messages in efficient ways (De Ruiter and Poll 2015; Fiterau-Brostean et al. 2020).

Coverage-driven fuzzing techniques have recently emerged as a popular solution, as demonstrated by the widespread adoption of the AFL fuzzer and similar tools (Zalewski 2021; Metzman et al. 2021; Manès et al. 2019; Boehme et al. 2021). For example, as of June 2021, OSS-Fuzz has found over 30,000 bugs in 500 open source projects (Google 2021; Serebryany 2017), with more and more open-source projects being integrated by the community (Korczynski and Korczynski 2021). This success could only be possible thanks to its fully-automated approach, which is based on unsupervised evolution of fuzz inputs, using simple and robust heuristics. However, research on coverage-driven fuzzing for stateful protocols is still at an early stage (Pham et al. 2020; Feng et al. 2021). These recent approaches infer protocol states by analyzing the contents of messages (e.g., *status codes*), using message parsers that are specifically developed for the protocol under test. Moreover, it is difficult for these approaches to fuzz many protocols, which only embed little or no state information within messages. These problems are a limiting factor towards securing more stateful network servers through fuzzing.

In this work, we propose a new solution for *stateful coverage-driven* fuzzing (StateAFL). Similarly to coverage-driven fuzzing, we inject code in the target binary using compile-time instrumentation techniques. The injected code infers protocol state information by: tracking memory allocations and network I/O operations; at each request-reply exchange, taking snapshots of long-lived memory areas; and applying fuzzy hashing (Locality-Sensitive Hashing, LSH) to map each in-memory state to a unique protocol state identifier. This approach does not rely on state information from network messages, and does not require developers to implement custom message parsers for extracting such state information. The aim of this approach is to contribute towards a completely-automated solution for stateful protocol fuzzing, similarly to what AFL was able to achieve for stateless programs, in order to promote a wider application of fuzzing in real-world systems. We

note that fuzzing research achieved significant progress from the point of view of fuzzing algorithms, but we are still witnessing at critical vulnerabilities (e.g., the well-known case of Hearthbleed (Wheeler 2020)) that in hindsight could have been easily prevented with fuzzing. Moreover, empirical research also showed that fuzzing a new system for the first time is likely to find security bugs (Böhme and Falk 2020). For these reasons, it is now a priority to make fuzzing more broadly applicable, as it is still too difficult to setup fuzzing to target new systems. In the case of stateful network fuzzing, StateAFL overcomes the issues of writing custom parsers to extract individual requests from seed inputs, and to extract status codes from response messages from the target server. These issues make fuzzing less accessible for developers that are new to this technique, since they are not inclined to write more code to use a fuzzing tool unfamiliar to them. Moreover, StateAFL is even applicable for protocols that do not provide any explicit status code in the messages, such as in the TLS protocol in our experiments, or where the status code only represents the status of the last request executed by the server instead of the protocol state, as in FTP and HTTP.

To assess the feasibility of the approach, we implemented and publicly released StateAFL as open-source software. Moreover, to support reproducible experimentation, we integrated StateAFL with a publicly-available benchmark of 13 open-source network servers, the largest experimental setup among stateful network fuzzing studies to the best of our knowledge. Our proposed approach allowed us to integrate StateAFL with no manual customization of the fuzzer to accomodate for the protocols under test. The experimental evaluation shows that StateAFL is a robust approach that can be applied to diverse network servers without requiring any protocol customization. Moreover, StateAFL can achieve comparable, or even better code coverage and bug detection than previous solutions based on stateless coverage-driven fuzzing and on stateful, protocol-customized fuzzing. We also qualitatively analyze state information both from parsing response codes returned by the target server, and from inference based on long-lived data. We found that using response codes provides misleading representation of the protocol state, leading to redundant states in the inferred protocol state machine and wasted fuzz inputs.

In summary, this paper presents the following contributions:

- A novel coverage-driven strategy for fuzzing stateful network servers, based on compile-time instrumentation and fuzzy hashing techniques to automatically infer protocol states from process memory;
- An open-source fuzzing tool based on the proposed approach, available at https://github.com/stateafl/stateafl. Similarly to AFL, this fuzzer is designed to be applicable to a wide variety of targets without requiring customizations.
- The integration of StateAFL in a public benchmark of network servers, with scripts to automate reproducible experimentation, available at https://github.com/profuzzbench/profuzzbench.
- An experimental evaluation of StateAFL, with respect to code coverage, bugs, and performance, along with a qualitative analysis of the inferred protocol states.

The paper is structured as follows. Section 2 discusses related work on stateful fuzzing. Section 3 presents the design and implementation of StateAFL. Section 4 presents the experimental plan, and Section 5 presents the experimental results. Section 6 concludes the paper.

2 Related Work

Generation-based fuzzers address stateful protocols by generating fuzz inputs using a *model* of the protocol, to be provided by a human analyst (Beyond Security 2020; Synopsis Inc 2020; Rapid7 2020). The model specifies both the format of protocol messages (e.g., field types, message separators, etc.) and their sequencing over a session (Poll et al. 2015), typically in the form of a graph, such as finite state machines, prefix acceptor trees, and Markov chains. The completeness of the model is critical for the effectiveness of fuzzing, but it can be difficult to achieve, since protocol specifications (which are typically written in natural language) are prone to misinterpretations and costly to analyze, and do not cover proprietary protocol extensions (Antunes and Neves 2011).

Several *model learning* techniques have been proposed to compensate for these issues, by (semi-)automatically inferring the types and formats of messages, and protocol state machines. *Passive* learning techniques infer from a corpus of network traces, using sequence alignment techniques (e.g., the Needleman-Wunsch algorithm) and statistical techniques (e.g., clustering into message types, and correlation of message fields) (Duchene et al. 2018; Kleber et al. 2018). *Active* learning techniques interact with the protocol server during the learning process, in order to refine the model and to elicit new protocol behaviors (e.g., based on Angluin's $L*$ algorithm and derivates) (De Ruiter and Poll 2015; Fiterau-Brostean et al. 2020). Both passive and active learning techniques provide valuable support for the human analyst, but cannot fully automate the process. For example, active learning can suffer from convergence issues and are applicable to finite input alphabets of modest size; thus, it needs an ad-hoc *mapper* to abstract protocol messages from/to the learner, to be tailored for the system-under-test (e.g., TLS-Attacker for the TLS protocol) (Somorovsky 2016). More powerful solutions leverage static and dynamic binary analysis (e.g., taint propagation analysis) to achieve full automation (Comparetti et al. 2009; Caballero et al. 2007), but in practice these solutions are difficult to implement and to port across different systems, which limits their adoption (Harman and O'Hearn 2018).

Coverage-driven fuzzing techniques have been adopted by AFL, LIBFUZZER, and other derivative tools (Manès et al. 2019) as a more practical and automated solution. This form of fuzzing only relies on lightweight metrics collected from the target system at run-time (e.g., about code blocks and branches covered by the fuzz inputs), and iteratively mutates the fuzz inputs to maximize these metrics. Therefore, the fuzzer can start from an initial set of fuzz inputs (i.e., a *seed* corpus) to automatically evolve them, without any a-priori knowledge about the protocol.

Only recently, coverage-driven fuzzing has been investigated for stateful protocols. AFLNET (Pham et al. 2020) extended AFL for fuzzing network protocols, by: structuring fuzz inputs into messages and applying mutation operators at message-level (e.g., by corrupting, dropping or injecting individual messages in a session); by learning a protocol state machine, where states are represented by response codes from the system-under-test; and by using the protocol state machine to prioritize mutations. SNIPUZZ (Feng et al. 2021) tailored coverage-driven fuzzing to IoT protocols, where the system-under-test could not be instrumented to collect coverage information, because of lack of access to the firmware. Thus, SNIPUZZ also analyzes response codes, using them as indicators to identify sensitive bytes of the inputs (snippets) that trigger different paths in the target.

This paper proposes a new approach for stateful protocol fuzzing. Our approach infers a protocol state machine on the basis on richer feedback than traditional coverage-driven fuzzing. The approach is not limited to analyze response codes, since response codes may

provide a poor indication of the current state of the server. For example, in an HTTP-based protocol, successful GET and POST requests may both receive the same response code (200), but POST requests may have side-effects on the state of the server, which are not reflected in the response code. Moreover, the protocol may lack response codes, such as in the case of TLS, thus leaving the fuzzer without any guidance about the current protocol state. Finally, even when response codes available, the fuzzer must be tailored for the target protocol, in order to extract and parse response codes from the response messages. For these reasons, our approach does not rely on response codes, but adopts compile-time instrumentation to get more information from the system-under-test and to infer the current protocol state. Moreover, the proposed approach relieves the user from providing custom message parsers.

3 Proposed Approach

We designed StateAFL to drive fuzzing based on *protocol states* covered during executions. In general terms, a protocol state guides the behavior of a process, by defining *which actions the process is allowed to take, which events it expects to happen, and how it will respond to those events* (Holzmann and Lieberman 1991). For example, most Internet protocols standardize the protocol states and their transitions in *Request for Comments* (RFC) documents, by describing them using prose in natural language or, in few cases, using finite state machines. Covering protocol states is a prerequisite for deeper code coverage of a protocol implementation, as some of its parts are only executed when the protocol reaches specific states. Moreover, exploring the protocol state space can uncover unintended or spurious behaviors of the protocol implementation that deviate from the protocol specification (Poll et al. 2015).

The StateAFL approach is designed around the fundamental receive-process-reply loop implemented by network servers. In this scheme, two parties (e.g., a *client* and a *server*) establish a *session*, which consists of a series of *request messages* and their corresponding *reply messages* (Poll et al. 2015). As the session progresses, the current protocol state is updated accordingly. The fundamental loop can be summarized by the following simplified pseudo-code:

```
long-lived data ← allocate()
while iterate indefinitely do
    short-lived data ← allocate()
    request ← receive()
    reply ← process(request, long-lived data, short-lived data)
    send(reply)
    deallocate(short-lived data)
end while
deallocate(long-lived data)
```

The key idea of StateAFL is to infer the current protocol state by inspecting the contents of process memory at each iteration of this loop. The current protocol state is necessarily stored into data structures, such as in heap and stack memory, which are updated at each request-reply exchange. In particular, the protocol state is represented by *long-lived data*, whose lifetime goes beyond an individual request-reply exchange, and spans across an entire session. Examples of such data are the current authentication status of a client, the

current working directory, and enqueued inputs to be processed (Natella and Pham 2021). Conversely, *short-lived data* have a short lifetime, as they store data only needed by one or few request-reply exchanges (such as, a buffer that temporarily holds the reply message). StateAFL follows the evolution of long-lived data structures thorough a session, and discards short-lived data. When fuzzing succeeds at reaching a new protocol state, the new state results in new contents of the long-lived data structures. Thus, the proposed approach takes a *snapshot* of such data at the end of each request-reply exchange. Then, it uses this snapshot as a proxy for the current protocol state, by assigning a unique state identifier to each unique memory state through fuzzy hashing.

Figure 1 shows the fundamental loop, with an overview of long- and short-lived data over a session. We refer to an individual request-reply exchange as an *iteration* of the fundamental loop. For the purpose of example, in addition to the loop in the previous pseudocode, the figure also shows the typical case of a *main* thread that listens for connection requests, and spawns a *worker* thread for each session. Long-lived data can be allocated both by the main and the worker thread. At the beginning of a session, the worker may optionally perform a SEND() to transmit an initial *banner* message that welcomes the client. Then, the worker performs one or more RECEIVE()s to get a request from the client, processes the request, and performs one or more SEND()s to communicate a reply to the client. The worker can allocate short-lived data both before the RECEIVE()s (e.g., a buffer for the incoming request) and after them (e.g., data for intermediate computations). Similarly, it can free short-lived data both before and after the SEND()s. We note that the end of a request/reply iteration (and the beginning of the next one) is denoted by a RECEIVE() after one or more SEND()s.

StateAFL has been designed on the basis of the fundamental loop of network servers. Similarly to AFL and other coverage-driven fuzzers, StateAFL is a mutation-based fuzzer, which automatically produces fuzz inputs by mutating previous ones, and gets feedback from the target program about the coverage achieved by the previous fuzz inputs. This feedback is important for coverage-driven fuzzers to prioritize which previous inputs to mutate, where to mutate them, and which mutation operators to apply. Differently from other fuzzers, StateAFL gets feedback not only about code coverage (e.g., which statements and branches were executed), but also about protocol states reached during an execution.

Figure 2 provides an overview of StateAFL. In the first step, StateAFL compiles the source code of the target program. During this process, we apply *compile-time instrumentation* techniques to introduce additional code in the binary executable generated by the

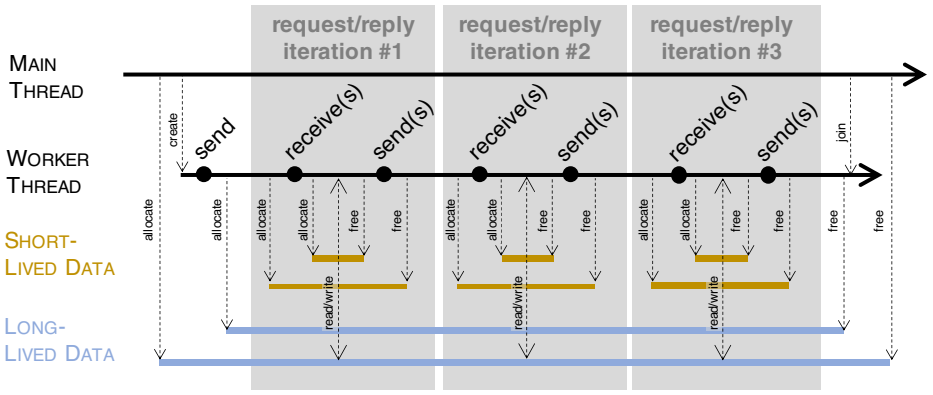


Fig. 1 The fundamental loop of network servers

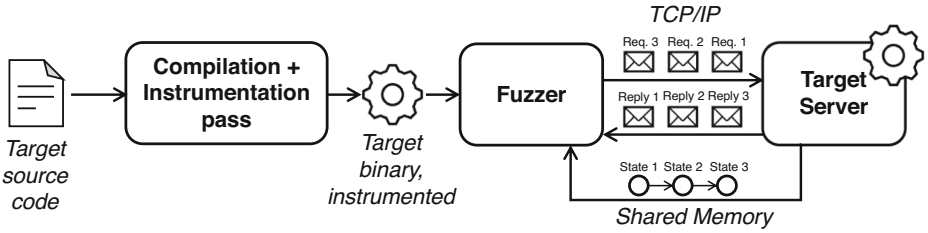


Fig. 2 Overview of StateAFL

compiler. The instrumentation adds code to collect feedback about the coverage of protocol states, in a similar manner to instrumentation code added by other fuzzers for analyzing code coverage. Note that this approach requires the availability of the source code of the target server. This scenario represents relevant use cases for stateful network fuzzing, such as developers that need to fuzz their own software (e.g., as part of an automated V&V process), and users of open-source software that need to gain additional security evidence. We leave the fuzzing of binary-only software out of the scope of this work.

After the instrumentation, the StateAFL fuzzer runs the target server by launching the binary. To exercise the target server with fuzz inputs, StateAFL exchanges TCP/IP messages with the target, in the same way of a client. A fuzz input is managed as a sequence of request messages: for each request message in the sequence, the fuzzer sends it through TCP/IP, waits for a reply message, and moves to the next request message. In addition, the StateAFL fuzzer collects information about protocol states reached by the target server, through a side channel (a shared memory area). The feedback consists of a sequence of states, one for each request/reply iteration. Note that StateAFL works on application-level messages, as demarked by the fundamental loop of SEND() and RECEIVE() primitives. The application messages may be (or may be not) divided among multiple packets by the TCP/IP stack, transparently to the fuzzer.

Our current design focuses on TCP/IP (including both the TCP and UDP transport protocols), since this protocol suite is the most commonly adopted by network servers. It is possible to easily adapt the design to other communication protocols, such as RPC servers. This design also focuses on client-server communication; fuzzing through multiple channels (e.g., multi-party protocols) represents a separate, still open research problem (Natella and Pham 2021), which we leave out of scope of this paper.

3.1 Instrumentation Probes

To collect feedback about protocol states, compile-time instrumentation weaves probes into the code of the target server. Probes are inserted at specific points of the code that allocate and free memory, and that send and receive data on the network. A probe consists of a call instruction, which invokes an external function, in order to perform actions when the server executes the instrumented points of interest. In some cases, the probe passes runtime information about the process to the external function (e.g., the address and size of a memory area).

The compile-time instrumentation links the target program to a library provided by StateAFL, which contains the external functions to be invoked by the probes. These library functions will collect and analyze data for inferring protocol states. In particular, the library provides the following functions:

- ON_ALLOCATE: This function is invoked when a heap or stack memory area has been allocated (e.g., using MALLOC). It takes in input the address and size of the memory area. It keeps track of all data structures, regardless that they are long- or short-lived (which cannot be determined at the moment of the allocation, but only afterwards).
- ON_FREE: This function is invoked when a heap or stack memory area has been deallocated (e.g., using FREE). It takes in input the address of the memory area. The function updates the status of data structures that were tracked by ON_ALLOCATE.
- ON_SEND and ON_RECEIVE: These functions are invoked when the server transmits or receives data to/from the client (e.g., a write or read on a socket), and keep track of the fundamental loop of the network server.
- ON_PROCESS_START: Executes at the start-up of the network server. It initializes the internal data structures (e.g., ALLOC_RECORDS_MAP and ALLOC_DUMPS_QUEUE), the internal state machine, and the shared memory area to communicate with the fuzzer.
- ON_PROCESS_END: Executes at the termination of the network server. It analyzes the data structures that were allocated by the network server during its execution, identifies which data are long-lived, and computes the sequence of protocol states, to be shared with the fuzzer.

StateAFL keeps track of the iterations of request/reply exchanges, by probing send()s and receive()s made by the network server. On these operations, ON_RECEIVE and ON_SEND update an internal state machine according to Fig. 3. These functions (Algorithms 2 and 3) represent the current iteration using the global integer variable $current\_iter\_no$, allocated by the StateAFL library and initially set to 0 (Algorithm 1). The state machine identifies the end of an iteration, by looking for a series of receive()s and, after some processing of the request, a send() (or, the first of a sequence of send()s). By the time that the server starts to send a reply, the long-lived data have been updated by the network server, and reflect a new protocol state. Therefore, on the first send() event, the iteration is considered as terminated, and a new one as started. We update the current iteration, by increasing $current\_iter\_no$ by one. Please note that the lifetime of short-lived data could end right before or right after the end of an iteration, depending on the network server. However, this does not pose a problem for StateAFL, since short-lived data are going to be ignored by later analysis, regardless of which iterations they span over.

During the execution, when a memory area is allocated on the heap or on the stack, the probes trigger the function ON_ALLOCATE (Algorithm 4). This function records the allocation using the ALLOC_RECORD data structure, which includes: (i) the number of the iteration at which the memory area was allocated, (ii) the number of the iteration at which it was deallocated (to be filled by ON_FREE), (iii) the address of the memory area, and (iv) the size of the memory area. The ALLOC_RECORD data structure is stored into a map (i.e.,

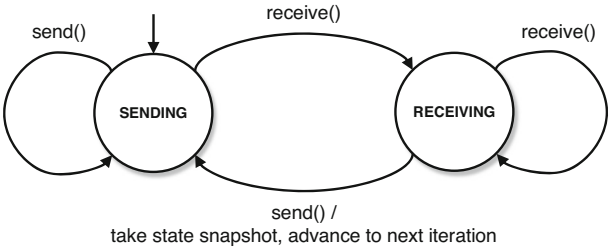


Fig. 3 State machine to keep track of protocol iterations

```
1: procedure ON_PROCESS_START
2:     alloc_records_map ← new map
3:     alloc_dumps_queue ← new queue
4:     current_iter_no ← 0
5: end procedure
```

Algorithm 1 Triggered when the server starts.

```
1: procedure ON_RECEIVE
2:     if state machine is SENDING then
3:         state machine ← RECEIVING
4:     end if
5: end procedure
```

Algorithm 2 Triggered when the server retrieves a message.

```
1: procedure ON_SEND
2:     if state machine is RECEIVING then
3:         state machine ← SENDING
4:         dump_current_state()
5:         current_iter_no ← current_iter_no + 1
6:     end if
7: end procedure
```

Algorithm 3 Triggered when the server sends a message.

an associative array), using as key the address of the memory area. The memory area is initialized to zero: StateAFL relies on the contents of the memory area as a proxy for the current protocol state, and must not contain random data. Since heap and stack memory areas in standard C are not automatically initialized, their contents are unpredictable and not correlated to the protocol state, until they are written by the program. Therefore, we initialize the heap and stack memory areas to assure that their unused parts have still a fixed and predictable value, which does not mislead the inference of protocol states. When the memory area is freed, the ON_FREE function updates its ALLOC_RECORD structure with the iteration number at which the area was freed (Algorithm 5). Still, the ALLOC_RECORD structure lasts until the termination of the network server. As an optimization, Algorithm 4 only records allocations that are made during the first iteration. The allocations made by the subsequent iterations do not span the entire lifetime of the process, and are considered as short-lived.

The ALLOC_RECORD data structures are inspected by StateAFL when the current iteration terminates, and the state machine moves to the next iteration (i.e., the transition from RECEIVING to SENDING, see Fig. 3). On this event, the ON_SEND function calls DUMP_CURRENT_STATE (Algorithm 6), which iterates over all of the currently-allocated heap and stack areas in ALLOC_RECORDS_MAP.

The DUMP_CURRENT_STATE function takes a snapshot (a *dump*) of the contents of every memory area, by saving them into an ALLOC_DUMP data structure. Moreover,

```
1: procedure ON_ALLOCATE(address, size)
2:     if current_iter_no = 0 then
3:         a ← new alloc_record
4:         a.iter_no_init ← current_iter_no
5:         a.iter_no_end ← −1
6:         a.addr ← initial address of allocated area
7:         a.size ← size of allocated area
8:         alloc_records_map.put(a.addr, a)
9:         zero-initialize the allocated area
10:    end if
11: end procedure
```

Algorithm 4 Triggered when the server allocates memory.

```
1: procedure ON_FREE(address)
2:     a ← alloc_records_map.get(address)
3:     a.iter_no_end ← current_iter_no
4:     alloc_records_map.remove(a.addr)
5: end procedure
```

Algorithm 5 Triggered when the server frees memory.

the ALLOC_DUMP will track the iteration number at which the snapshot was taken, and a reference to the ALLOC_RECORDS_MAP for the memory area. Even if the network server deallocates the memory area, its ALLOC_RECORD structure is still saved and referenced by the ALLOC_DUMP structure. All ALLOC_DUMP structures are enqueued into ALLOC_DUMPS_QUEUE.

Figure 4 provides an example of the ALLOC_RECORD and ALLOC_DUMP data structures. In this example, the network server initially allocates a long-lived data structure at address $addr_0$, and represented by $addr\_record_0$. For this long-lived area, $iter\_no\_init$ is initialized to 0, since it has been allocated before the first iteration could complete. Then, the network server iterates for three request/reply exchanges. At every iteration, the

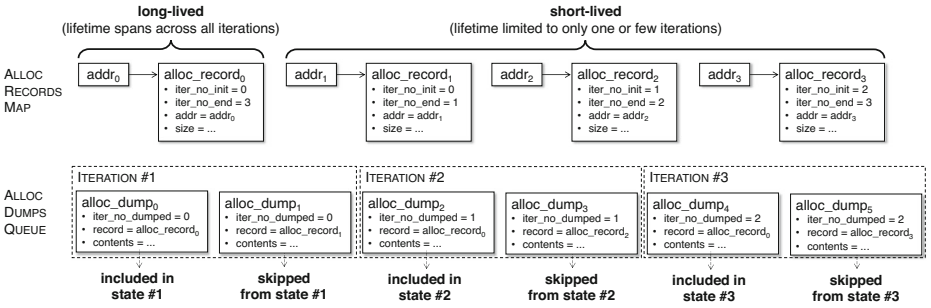


Fig. 4 Example of data structures (*alloc_record* and *alloc_dump*), after an execution with three iterations, with one long-lived area allocated at the beginning and freed at the end, and three short-lived areas allocated and freed at each iteration

```
1: procedure DUMP_CURRENT_STATE
2:     for all a ∈ alloc_records_map do
3:         d ← new alloc_dump
4:         d.iter_no_dumped ← current_iter_no
5:         d.record ← reference to a
6:         d.contents ← copy a.addr's contents
7:         alloc_dumps_queue.push(d)
8:     end for
9: end procedure
```

Algorithm 6 Takes snapshots of memory areas.

server allocates a short-lived data structure before processing the request, and deallocates it after sending the reply. Thus, the server allocates in total 3 short-lived memory areas ($addr\_record_1$, $addr\_record_2$, and $addr\_record_3$, respectively). The alloc_dump structures for the short-lived data are annotated with the iteration in which they were allocated ($iter\_no\_init = 0, 1, 2$, respectively) and deallocated ($iter\_no\_end = 1, 2, 3$, respectively). We remark that allocations performed after the first iteration $(1, 2, \ldots)$ are not actually tracked by our algorithm, but are included in this discussion as an example of short-lived data.

In the example of Fig. 4, the DUMP_CURRENT_STATE function is triggered 3 times, at the end of each iteration. At the first iteration, DUMP_CURRENT_STATE dumps the current contents of the long-lived data structure ($alloc\_dump_0$), and the contents of the first short-lived data structure ($alloc\_dump_1$). Note that the ALLOC_DUMP structures have a reference to the ALLOC_RECORD structures. Similarly, at the end of the second and third iterations, DUMP_CURRENT_STATE dumps again the current contents of the long-lived data structure ($alloc\_dump_2$ and $alloc\_dump_4$). The dumps $alloc\_dump_0$, $alloc\_dump_2$, and $alloc\_dump_4$ are from the same long-lived data structure, but they can hold different contents, as the network server updates long-lived data at each iteration. Finally, the Fig. 4 includes the dumps $alloc\_dump_3$ and $alloc\_dump_5$ for the other two short-lived area, taken respectively at the end of the second and third iteration.

3.2 Post-execution Analysis

The dumps in ALLOC_DUMPS_QUEUE are later analyzed at the end of the execution, after that all request/reply iterations for the fuzz input have been completed. After the last iteration, the network server is forcefully terminated, and the ON_PROCESS_END FUNCTION is triggered (Algorithm 7). In turn, it calls the SAVE_STATE_SEQ FUNCTION.

```
1: procedure ON_PROCESS_END
2:     total_iterations ← current_iter_no
3:     save_state_seq()
4: end procedure
```

Algorithm 7 Triggered when the server terminates.

The SAVE_STATE_SEQ function (Algorithm 8) iterates over the ALLOC_DUMPS_QUEUE. As result, SAVE_STATE_SEQ generates a sequence of states, with one state for each iteration made by the network server. A state is represented by a unique integer value (*state id*), based on the contents of long-live data at the end of the iteration. Therefore, if long-lived data are updated between an iteration and the next one, the two states will be represented by two distinct integer values. Otherwise, if the long-lived data stay unchanged between iterations, the states are represented by the same integer value. Of course, it is possible that the same integer value (i.e., the same state) appears multiple times at distant times in the sequence, as the network server can return to a previous state, depending on the server behavior. In the example of Fig. 4, SAVE_STATE_SEQ generates a sequence of three states, represented by three integer values, which can be different or identical depending on any changes made to the long-lived data structure.

Algorithm 8 iterates over ALLOC_DUMPS. The algorithm identifies dumps of long-lived data, by looking for those whose lifetime spans across all iterations. The algorithm skips a dump as short-lived data if its memory area has been allocated after the first iteration ($iter\_no\_init > 0$), or if it has been deallocated before the termination of the last iteration ($iter\_no\_end < total\_iterations$, except $iter\_no\_end = -1$ that denotes an area never deallocated). In the case of Fig. 4, the first state is obtained only from $alloc\_dump_0$ (i.e., the first dump of the long-lived area); similarly, the second and third states are only based on $alloc\_dump_2$ and $alloc\_dump_4$ (i.e., second and third dump of the long-lived area).

When iterating over the dumps, the algorithm computes a *hash function* over the union of all dumps for the same protocol iteration. The hash value is adopted to map the memory contents to a unique state identifier. The hash value is computed incrementally, by updating it with one dump at a time. When the algorithm finds a dump for a new iteration ($d.iter\_no\_dumped > prev\_iter\_no$), the state identifier for the previous iteration is finalized and pushed to the sequence, and the analysis is repeated for the next iteration, until all dumps have been analyzed.

A potential drawback of using hash functions is that the state identifier could be oversensitive to small, negligible variations of the memory contents not correlated with the protocol state, because of non-deterministic factors. For example, as the fuzzer executes the target server multiple times, the process may get from the OS different descriptors for socket and file I/O, or its data may be allocated at different addresses of the virtual memory space. In turn, these values can be copied to long-lived data structures (e.g., pointer variables). Most hash functions are designed to be sensitive to small changes, and to generate largely different hash values even if the inputs are similar. Therefore, small, non-deterministic variations would lead to different, redundant state identifiers, even if the variations do not affect the behavior of the server. Disabling OS randomization mechanisms reduces, but does not prevent such variations.

To mitigate this issue, our algorithm adopts *Locality-Sensitive Hashing* (LSH). In LSH, two similar inputs (e.g., differing only for few bits) result in two hash values that are different, but similar (Jafari et al. 2021). This form of hashing has applications in several domains, such as document retrieval, plagiarism detection, and bioinformatics. In the field of software security, LSH has most often been adopted for analyzing malware similarity (Oliver et al. 2013; Ali et al. 2020). In one previous work, LSH has been used on path constraints of symbolic execution states, in order to speed-up the search for previously-solved states (Cady 2017). In general, LSH enables the quick look-up of items that are similar to the one under analysis, by looking for items with a similar hash value according to some distance metric.

```
1:  procedure SAVE_STATE_SEQ
2:      states_sequence ← new list of integers
3:      prev_iter_no ← 0
4:      tlsh_hash ← 0

5:      for all d ∈ alloc_dumps_queue do

6:          ▷ Long-lived data span across all iterations
7:          if d.record.iter_no_init > 0 or
8:             (d.record.iter_no_end < total_iterations and
9:             d.record.iter_no_end ≠ −1) then
10:             skip d                                  ▷ Ignore short-lived data
11:         end if

12:         if d.iter_no_dumped > prev_iter_no then
13:             ▷ Save state of the current iteration before the next
14:             state_id ← get_state_id(tlsh_hash)
15:             states_sequence.push(state_id)
16:             tlsh_hash ← 0
17:         end if

18:         update_tlsh(tlsh_hash, d.contents)
19:         prev_iter_no ← d.iter_no_dumped
20:     end for

21:     state_id ← get_state_id(tlsh_hash)
22:     states_sequence.push(state_id)
23:     save_to_shared_memory(states_sequence)
24: end procedure

25: procedure GET_STATE_ID(tlsh_hash)
26:     radius ← ϵ
27:     state_id ← mvptree_lookup(tlsh_hash, radius)

28:     if state_id = ∅ then
29:         state_id ← mvptree_count() + 1
30:         mvptree_add({tlsh_hash, state_id})
31:     end if

32:     return state_id
33: end procedure
```

Algorithm 8 Generates a sequence of protocol states.

In particular, in this work we adopt the *Trend Micro Locality Sensitive Hash* (TLSH), a popular algorithm that has shown high robustness against small differences in the inputs (Oliver et al. 2013; Ali et al. 2020). TLSH computes a distribution of the bit patterns

in the data, and generates a digest from this distribution. TLSH also comes with a distance metric between hash values, which approximates the Hamming distance between two hash digest bodies. The function GET_STATE_ID (Algorithm 8) takes in input the TLSH hash of long-lived data for the current iteration, and performs a nearest neighbor search for the previous most similar hash value, within a maximum distance of ϵ. If a similar hash value is found, the algorithm returns its mapped state identifier; otherwise, a new pair $\{tlsh\_hash, state\_id\}$ is stored using a new, unique state identifier. The algorithm uses a *Multi-Vantage Point* (MVP) tree data structure to store the pairs, and to perform look-ups based on the TLSH hash and on the TLSH distance metric. We use a MVP tree for computationally-efficient nearest-neighbor search, as it avoids expensive pair-wise comparisons with previous values in the tree (Bozkaya and Ozsoyoglu 1997, 1999).

The distance threshold ϵ is dynamically calibrated for the target server under test, according to Algorithm 9. Before fuzzing, StateAFL performs a calibration stage, by executing the server multiple times using the seed inputs, and by computing a sequence of hash values at each iteration of each repetition. In our implementation, we run one "reference" execution plus 3 additional repetitions for each seed input. Since the server is executed with the same inputs, the distance threshold ϵ should be calibrated such that the sequence of state identifiers is the same across repetitions. Therefore, the calibration stage compares the hash value at each iteration of the first execution ($reference\_hashes\_seq[i]$) with the corresponding hash value of every other repetition ($new\_hashes\_seq[i]$), and collects the distances between the hash values ($distances$). Then, it takes the 90^{th} percentile of these distances as a conservative choice for ϵ. The rationale for choosing the 90^{th} percentile is that the distances across calibration runs should fall within the threshold (i.e., clustered in the same state), since these runs process the same requests starting from the same initial state. Taking a higher threshold (e.g., the maximum distance) would be too conservative, since it would be influenced by sporadic outliers caused by non-determinism. Taking a lower threshold (e.g., a lower percentile) would make the approach prone to redundant states, since two occurrences of the same state may not be recognized as such.

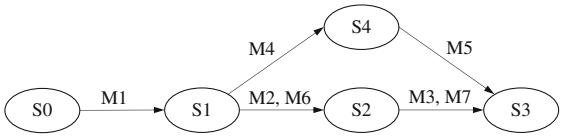
After calibrating ϵ, it is expected that the server often returns to a previously known state, and that it visits new states infrequently (e.g., the fuzzer triggers a new corner case in the protocol). However, it is possible that the threshold is set too low if the initial seed inputs are too few or too short, since only few states are visited during the calibration. This case often occurs when a security assessor undertakes a fuzzing campaign for a new or unfamiliar system. Therefore, StateAFL provides a threshold adjustment procedure as a fallback option to compensate for an under-estimated threshold. The adjustment is based on the observation that, if the threshold is too low, then a high number of new states will quickly occur during fuzzing. To handle this case, the fuzzer increases the threshold when new states are added for 5 consecutive fuzz inputs, which is very unlikely to happen for a well-calibrated threshold, since new states should be infrequent. The threshold is increased by 10, which is a relatively low value to account for small corrections, but also not too low, in order to react quickly. As in the original proposal of TLSH, we vary the threshold ϵ between 5 and 100.

Ultimately, the SAVE_STATE_SEQ function returns the sequence of states to the StateAFL fuzzer. The fuzzer incrementally grows a state machine after each fuzz input, based on the returned sequence of states. For example, when a fuzz input covers a new state, the state machine is updated by adding a new state and a new transition from the previous state in the sequence. Similarly, a new transition is added to the state machine when a pair of states appears consecutively in a sequence for the first time.

```
1: procedure THRESHOLD_CALIBRATION
2:     distances ← empty set
3:     for all s ∈ seedinputs do
4:         run_target(s)
5:         reference_hashes_seq ← get_last_run_hashes_seq()
6:         for all r ∈ 1 ... #repetitions do
7:             run_target(s)
8:             new_hashes_seq ← get_last_run_hashes_seq()
9:             for all i ∈ 1 ... len(reference_hashes_seq) do
10:                dist ← tlsh_distance(reference_hashes_seq[i], new_hashes_seq[i])
11:                distances ← distances ∪ dist
12:            end for
13:        end for
14:    end for
15:    ▷ Get 90th percentile of distances
16:    threshold ← distances_90%
17: end procedure
```

Algorithm 9 Threshold calibration.

During the fuzzing process, StateAFL uses the state machine to generate new fuzz inputs, in order to further increase code coverage and to explore the protocol. Heuristics from previous model-based fuzzing techniques can be leveraged for this purpose (Pham et al. 2020), to: (i) select a target state from the state machine; (ii) identify a previous fuzz input that reached the selected state; and (iii) apply a mutation operator on the message that is sent from the target state. Figure 5 provides an example of inferred protocol state machine, and related information that is tracked by the fuzzer for state selection purposes: *#fuzzs* is the number of previous mutated inputs that have exercised that state; *#paths* is the number of times that the code or state coverage increased when the state was previously selected; and *#selected* is the number of times that the state was previously selected. Moreover, the table tracks inputs that covered each state and that are "interesting", i.e., that increased code or state coverage. The target state is selected with a probability that is inversely proportional to *#fuzzs* and *#selected*, and proportional to *#paths*. After selecting a state, the fuzzer randomly selects one of the previous interesting inputs that covered that state (*Inputs*). Finally, the fuzzer identifies the message in the input that reaches the selected state, and it targets for mutation the subsequent message in the same input. For example, in Fig. 5, if the fuzzer targets *S4*, it will generate a fuzz input beginning with messages *M1* and *M4*, followed by a mutated version of message *M5*.



(a) Inferred protocol state machine.

States	Inputs	#fuzzs	#paths	#selected
S1	{M1,M2,M3}, {M1,M6,M7}, {M1,M4,M5}	...	...	...
S2	{M1,M2,M3}, {M1,M6,M7}	...	...	...
S3	{M1,M2,M3}, {M1,M6,M7}	...	...	...
S4	{M1,M4,M5}	...	...	...

(b) Information associates to the states.

Fig. 5 Example of protocol state machine inference and state selection

The mutations include both byte-level operators and message-level ones (Table 1). The byte-level operators are derived from the AFL fuzzer (Zalewski 2021), and modify the content of an individual message. There are two types of byte-level mutation operators: *deterministic* and *stacked*. The deterministic ones systematically mutate all of the bits, bytes, words, and double words in the original input. For example, in the "single walking bit flip" mode, the fuzzer iterates sequentially over all of the bits in the original input, and generates a distinct fuzz input by inverting each of these bits. Figure 6 shows the case of three different fuzz inputs generated by inverting three different bits of the original input. This approach is meant to discover sensitive parts of the original input (e.g., headers) that increase coverage when mutated, and that are interesting to further mutate. Afterwards, the fuzzer applies *stacked* mutations, where multiple types of mutations are applied on the same fuzz input. In this case, the mutated parts of the inputs are randomly selected. For example, in Fig. 6 three mutation operators are applied on the same input, respectively a random bit flip, a random byte set to a random value (`CC`), and a random value added to a random byte (`+3`). Four more mutation operators at the message-level are derived from the AFLNET fuzzer (Pham et al. 2020). These mutations replace, insert, and duplicate a message at the location of the message for selected fuzzing. The message-level operators are stacked with byte-level operators.

3.3 Implementation

We implemented STATEAFL on top of the codebase of AFL and AFLNET. For compile-time instrumentation, we extended the AFL-CLANG-FAST utility, which is provided by AFL to compile the target program, and which adds a compiler pass to introduce instructions for coverage profiling. In the compiler pass, we add further instrumentation to introduce probes in the program, as discussed in Section 3. We focus on the case where the source code of the target server is available for fuzzing.

Probes are injected on heap allocation sites that invoke the standard C library functions MALLOC, REALLOC, CALLOC and FREE, and the C++ operators NEW and DELETE, in order to call ON_ALLOCATE and ON_FREE. The probes take the size of the allocated memory

Table 1 Mutation operators

Mutation	Type	Fuzzer
Single, two, or four walking bit flips	Deterministic	AFL
One, two, or four walking byte flips	Deterministic	AFL
Walking 8-, 16-, or 32-bit arithmetics	Deterministic	AFL
Walking 8-, 16-, or 32-bit innteresting values	Deterministic	AFL
Walking overwrite with user-supplied dictionary values	Deterministic	AFL
Walking insertion of user-supplied dictionary values	Deterministic	AFL
Splicing multiple inputs	Deterministic	AFL
Flip single random bit	Stacked	AFL
Set random byte, word, or double word to interesting value	Stacked	AFL
Subtract value at a random byte, word, or double word	Stacked	AFL
Add value at a random byte, word, or double word	Stacked	AFL
Set random byte to random value	Stacked	AFL
Delete random bytes	Stacked	AFL
Clone random bytes	Stacked	AFL
Insert block of constant bytes in random position	Stacked	AFL
Overwrite bytes with randomly selected ones	Stacked	AFL
Overwrite bytes with fixed bytes	Stacked	AFL
Replace message with a random one from a random input	Stacked	AFLNET
Insert random message from a random input, before the target message	Stacked	AFLNET
Insert random message from a random input, after the target message	Stacked	AFLNET
Duplicate message	Stacked	AFLNET

area from the input of the allocation, and its memory address from the output. Similarly, allocation sites of stack memory are probed, by identifying PUSH and POP operations on the stack that modify the stack pointer register. The probes compute the address and size of the allocated memory area from the stack pointer register. In order to avoid excessive overhead that may be caused by probing all stack allocations, we only probe allocations of data structures larger than a threshold (64 bytes), since in practice small allocations typically represent temporary variables and do not hold long-lived data structures. Data in globals and TLS are handled as memory areas with a lifetime spanning the entire execution.

Probes are also injected on call sites to standard library functions that send and receive network data, such as SEND, SENDTO, and SENDMSG (to trigger ON_SEND), RECV, RECVFROM, and RECVMSG (to trigger ON_RECEIVE). We also probe the standard library

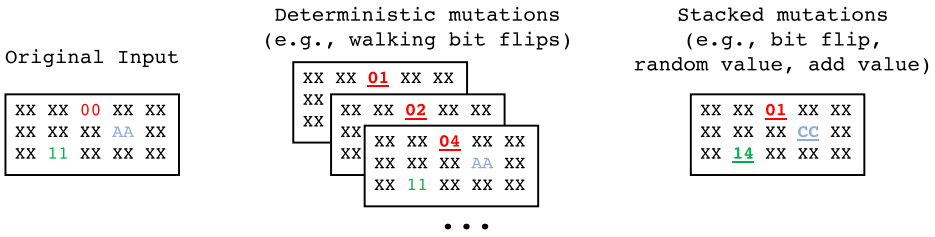


Fig. 6 Examples of mutated inputs

functions READ, WRITE, FPRINTF, FGETS, FREAD, and FWRITE, with an additional check that the file descriptor in input is a network socket. We allow the user to specify (using an environment variable) any program-specific function that should be instrumented for intercepting network communication. For example, for a server that implements a HTTP-based protocol using the *libevent* API, the user can instruct StateAFL to instrument the EVHTTP_REQUEST_* and EVHTTP_SEND_* API functions, to trigger the external functions ON_RECEIVE and ON_SEND, respectively. Finally, our probes invoke ON_PROCESS_START and ON_PROCESS_END on start-up and termination of the target program.

After the compiler pass, we link the program executable with a library that implements the event handlers, to be called by the probes. The library shares a UNIX SysV shared memory to exchange state sequences. We replaced the protocol-specific message parsers of AFLNET with a single, generic function that read the state sequence from the shared memory, without parsing response codes from the messages. We reuse the test automation from AFL and AFLNET to execute the target program (e.g., the fork server), and to mutate fuzz inputs.

As an optimization for speeding-up fuzzing, StateAFL can be configured to perform heavy-weight post-execution analysis of long-lived memory only when strictly needed. Fuzz inputs are normally processed without performing the post-execution analysis, to have a high fuzzing throughtput; when a fuzz input covers a new program path (i.e., increases the code coverage), it is processed again in order to the post-execution analysis. The analysis returns a sequence of states reached by the fuzz input, which is stored by the fuzzer. This information is used by the fuzzer to generate more fuzz inputs starting from each state in the sequence.

4 Experimental Plan

We evaluate StateAFL by fuzzing real-world network servers from popular open-source projects. The experimental plan addresses the following research questions:

RQ1: How StateAFL compares to state-of-the-art network fuzzing? We evaluate them with respect to both code coverage, which is a typical indicator of the depth of fuzz testing, and crashes of the targets, which indicates that a fuzzer can uncover potential security issues (Klees et al. 2018).

RQ2: How accurate are the inferred protocol states? This is a difficult question, since we lack a ground truth for the protocols to be inferred, and since the protocol state machine depends on the specific protocol implementation of the network server (Poll et al. 2015). We address this question through a qualitative analysis on one of the target network servers, by manually analyzing its source code, to check that the inferred states are not redundant and reflect the expected behavior of the protocol. As a further term of comparison, we also compare the inferred state machines by StateAFL with the ones inferred by custom protocol-specific fuzzing, in terms of number of states and other graph complexity metrics.

RQ3: Can StateAFL achieve a high fuzzing performance? The main principle for effective fuzzing is to generate large amounts of inputs over a long period of time. In order to assure that StateAFL can perform a high number of tests in the long run, it is important to minimize its overhead on the execution of the server under test. Thus, we evaluate the performance slow-down of the instrumented targets compared to non-instrumented execution.

To assess the feasibility of the approach and to support reproducible experimentation, we implemented StateAFL and integrated it with ProFuzzBench, a public benchmark for network fuzzers (Natella and Pham 2021). The benchmark includes 13 open-source network servers (Table 2). These targets are quite diverse with respect to several aspects: they cover 10 network protocols that have been typical targets of previous fuzzing studies; they are implemented both in C and in C++; they include both TCP and UDP, and both binary and text protocols; they adopt a variety of APIs (e.g., send/recv vs. fwrite/fread for networking, pthreads vs. fork for multiprocessing). ProFuzzBench automates the setup and the execution of the target servers using Docker containers, in a reproducible way. Moreover, ProFuzzBench configures the servers according to the best practices for coverage-driven fuzzing. In particular, the targets are patched to disable sources of randomness (e.g., pseudo-random number generators) in order to have reproducible behavior (i.e., if the program is executed again with the same input, then the same execution path is covered), which is an implicit assumption for coverage-driven fuzzing techniques. The experiments adopt the seed inputs from the ProFuzzBench project, where both practitioners and researchers contributed with both benchmark targets and with seeds for these targets. These seeds reflect typical basic usage of the servers according to their experience. The seeds include correct authentication and passwords (otherwise, the fuzzer would waste significant time before getting access to the server), and other frequent commands for the protocol (e.g., for FTP, the seeds get the list of files on the server, create directories and move across them, etc.). Table 2 provides the number of unique commands in the seeds for each target server. We remark that the need to provide initial seeds for the target server is a problem for any greybox fuzzing approach, in terms of automation and ability to work out-of-the-box for new software to test. We leave this aspect out of the scope of this work.

The experimental evaluation compares StateAFL with two baseline fuzzers. The baseline fuzzers were selected such that: (i) They are not limited to specific network protocols, but are applicable to a large set of network targets, including the ones in ProFuzzBench;

Table 2 Benchmark targets

Target	Protocol	Type	Transport	Lang.	Multiproc.	Seeds
Bftpd	FTP	Text	TCP	C	fork	54
Dcmtk	DICOM	Binary	TCP	C++	pthreads	4
Dnsmasq	DNS	Binary	UDP	C	fork	9
Exim	SMTP	Text	TCP	C	fork	9
Forked-daapd	DAAP	Text	TCP	C	pthreads	65
Kamailio	SIP	Text	UDP	C	fork	3
LightFTP	FTP	Text	TCP	C	pthreads	10
Live555	RTSP	Text	TCP	C++	N/A	33
OpenSSH	SSH	Binary	TCP	C	fork	22
OpenSSL	TLS	Binary	TCP	C	N/A	8
ProFTPD	FTP	Text	TCP	C	fork	54
Pure-FTPd	FTP	Text	TCP	C	fork	54
TinyDTLS	DTLS	Binary	UDP	C	N/A	5

this leaves out fuzzers that are highly-customized for a specific protocol (e.g., TLS (De Ruiter and Poll 2015), DTLS (Fiterau-Brostean et al. 2020)) but are not applicable to other protocols; (ii) They adopt state-of-the-art greybox, coverage-driven techniques, in order to evaluate how the proposed greybox solution relates to them. The two baseline fuzzers are:

- AFLNWE: It is a "network-enabled" version of AFL, with minor changes to send mutated inputs over a TCP/IP socket instead of using file I/O. It adopts the same mutation operators and coverage analysis from AFL.
- AFLNET: It is another fork of AFL, with extensive modifications for stateful network fuzzing. It organizes an input as a session of multiple messages, and adds mutation operators at the message level (e.g., dropping or duplicating individual messages, rather than bytes or blocks). Moreover, it relates each input message to a protocol state reached by that message, where the protocol state is represented by the "status" code from the response by the server.

These two tools represent different points in the design space of greybox network fuzzers. On the one hand, AFLNWE is a pure greybox, coverage-driven fuzzer, and it is a baseline to evaluate the relative merit of stateful fuzzing compared to plain coverage-driven fuzzing. On the other hand, AFLNET is a stateful network fuzzer that performs protocol state inference. Differently from the proposed STATEAFL fuzzer, AFLNET relies on the contents of response messages to infer protocol states. Therefore, to be applicable, AFLNET must be customized with protocol-specific parsers, in order to extract status codes from the messages (where available). Therefore, AFLNET comes with parsers for a set of common protocols, and PROFUZZBENCH extended AFLNET with more parsers to support the network servers under test (Table 2). For some protocols (e.g., TinyDTLS), response messages do not have status codes; thus, the protocol parsers generate status codes from other fields (e.g., in DTLS, by joining the *content type* field from the header, and the *message type* field from the payload), based on protocol knowledge of AFLNET's developers.

STATEAFL overcomes the need for protocol-specific parsers, by instrumenting the target process and analyzing its memory at run-time, in order to be more broadly applicable without the need for manual customizations. In our evaluation, we analyze whether the protocol state inference by STATEAFL can overcome the lack of protocol parsers. The experimental plan consists of a total of 156 experiments, with 4 repeated fuzzing experiments for each of the 13 target servers and of the 3 fuzzers, over a period of 24 h for each experiment. We execute experiments on the Google Cloud Platform, using E2 high-memory VM instances with 4 vCPUs, with a dedicated vCPU for each replication.

In our evaluation, the STATEAFL fuzzer could successfully run on all of the target servers, without any protocol customization. STATEAFL automatically instruments I/O APIs from the standard C library, to trigger ON_SEND (on SEND, SENDTO, SENDMSG, WRITE, FPRINTF, and FWRITE) and to trigger ON_RECEIVE (RECV, RECVFROM, RECVMSG, READ, FGETS, FREAD). In only one of the targets (Forked-daapd), which performs network I/O through the *libevent* API, we needed to configure STATEAFL to probe the EVHTTP_REQUEST_GET_URI and EVHTTP_SEND_REPLY API functions, in order to keep track of its request/reply loop. The information about the names of these APIs is easily available to developers, and can be learned from a quick inspection of the target server. No modification of STATEAFL was needed, as it instruments these APIs in the same way of other APIs.

5 Experimental Results

5.1 Coverage and Vulnerabilities

In Table 3 and Fig. 7, we report respectively on crashes and on coverage for each target and for each fuzzer, after 24 h of fuzzing. For coverage, we focus on edge coverage and do not show line coverage for the sake of space, as it exhibits similar results to edge coverage.

For coverage (Fig. 7), we have different outcomes depending on the target. For 6 targets (Bftpd, Dcmtk, Dnsmasq, Live555, OpenSSH, ProFTPD), we notice that all of the fuzzers achieved a similar coverage. In the other cases, the stateful fuzzers (StateAFL and AFLNet) achieved higher coverage than the non-stateful AFLNwe fuzzer. In particular, for three targets (LightFTP, Exim, TinyDTLS), the gap is larger, while for the remaining targets (OpenSSL, Pure-FTPd, Forked-daapd, Kamailio), the gap is relatively small, but there is a higher variability between experimental runs.

In the 6 targets with similar coverage, both of the stateful fuzzers could not increase coverage compared to plain greybox fuzzing. A possible reason is that the server behavior is only weakly correlated to the current state of the process, as it is influenced mostly by the current input. Therefore, stateless fuzzing could eventually catch up with the stateful fuzzers over the course of the experiment. In the other 7 cases, the stateful fuzzer benefited from inferring protocol states. When a message succeeds at discovering a new state, it uses the state (and the messages sent up to that point) as a starting point to generate more inputs. For example, the fuzzer can add further messages after that starting point, and cover new parts that are enabled by the current protocol state. Instead, a stateless fuzzer does not reason in terms of sequences of states, and focuses on mutating the "interesting bits" of the input that recently changed the coverage, slowing down the effectiveness of fuzzing. This is further discussed in the analysis of the next subsection (Section 5.2).

The availability of "status codes" in the response also seems to have an influence, as in the case of TinyDTLS and, to a minor degree, in the case of OpenSSL. These projects implement binary protocols, and lack a "status" code in the response message. For these protocols, the custom parsers in AFLNet produce surrogate values, which are computed from other fields in the header and payload. Therefore, the server gives a weaker indication about the current protocol state. In StateAFL, the (un)availability of status code in the response does not affect the fuzzing process, as the protocol state is automatically inferred from the analysis of the memory of the target server.

For bugs, we focus on identifying which are the targets where a fuzzer found any crash. We do not consider the absolute number of crashes reported by the fuzzers ("unique crashes"), which is widely acknowledged as an unreliable metric, since the crashes are

Table 3 Unique bugs found by the fuzzers after 24 h of fuzzing. The ✓ denotes unique bugs, and the numbers in parentheses denote average non-deduplicated crashes

Target	AFLNwe	AFLNet	StateAFL
Dcmtk	✓ (1)	✓ (10)	✓ (9)
Dnsmasq	✓ (54)	✓ (57)	✓ (66)
Live555	✓ (175)	✓ (211)	✓ (187)
ProFTPD	–	–	✓ (1,051)
TinyDTLS	✓ (20)	✓ (37)	✓ (56)

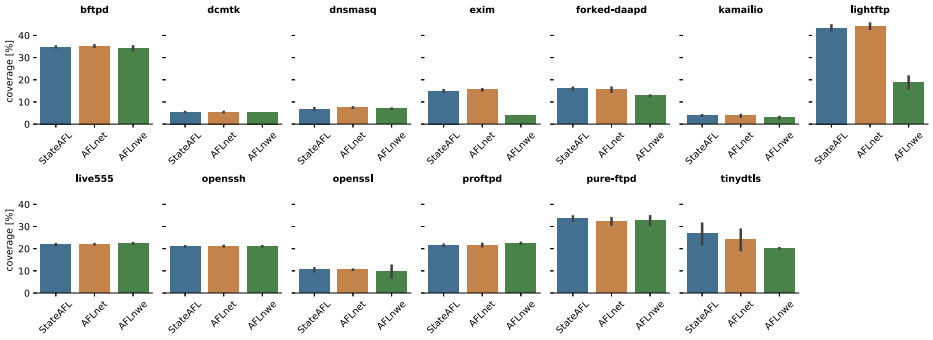


Fig. 7 Edge coverage after 24 h of fuzzing

duplicates of the same underlying vulnerability. For example, the "unique crashes" terminology has recently been dropped by the community working on AFL-based fuzzers (AFLplusplus Project 2021). Therefore, to deduplicate bugs (i.e., to identify crashes that are caused by the same root cause), we first grouped the crashes with respect to their call stack at the time of the crash. Then, we manually analyzed the call stacks, in order to establish whether two groups of crashes with different call stacks where still due to the same bug. Even if different groups of crashes had small differences in the call stack, they were still accounted to the same unique bug due to their semantic similarity. For example, for Live555, two different call stacks with *handleCmd_SETUP* are both related to the *SETUP* command in the RTSP protocol, and were considered as effect of the same bug.

Table 3 provides information on the targets that crashed in the experiments. The checkmark symbol "✓" denotes a unique bug found by fuzzing (in these experiments, at most one for each target listed in the table). StateAFL was able to crash the same targets of both AFLNwe and AFLNet (Dcmtk, Dnsmasq, Live555, TinyDTLS), but without relying on protocol customizations. Moreover, StateAFL was the only fuzzer able to find crash-inducing inputs for the ProFTPD target. The other fuzzers did not find any bug not found by StateAFL. The underlying bug is a heap buffer over-read, which could not be triggered by the other fuzzers since it is difficult to reproduce. ProFTPD introduces its own heap memory allocator on top of the GNU C library, by allocating memory blocks from pre-allocated pools. Therefore, the buffer overrun could not be reliably detected since the buffer overrun can still access to valid memory areas in the same pre-allocated pool. In the case of StateAFL, the bug was triggered since the fuzzer put more stress on the memory allocator, thus fragmenting the data and making the buffer over-run more likely to access to invalid memory areas.

All of these crashes were found within one hour of fuzzing. For the other targets, none of the fuzzers were able to find deeper vulnerabilities within 24 h. However, as discussed later (Section 5.2), StateAFL is able to restrict the set of inferred protocol states, which contributes to find deep bugs in the long run, by avoiding repeating the same tests on redundant states. For example, inferring two redundant states (i.e., the server does not actually exhibit different behaviors in these states) leads the fuzzer to repeatedly apply the same fuzz input on both the two states, causing a waste of computational efforts. We remark that the main contribution is the increase in automation (as it avoids the user to write protocol-custom code) and in broadening the scope of fuzzing (as it can support more protocol types with lower effort), while keeping a performance level comparable to existing fuzzers in terms of coverage and bugs found.

5.2 Protocol State Inference

To get a better understanding of the protocol state machine inferred by StateAFL, we first analyze one of the targets in a qualitative way. We focus on the FTP protocol, since it is a plain-text protocol that is simple to be manually interpreted, and that has been targeted by many fuzzing studies (Natella and Pham 2021). Moreover, the FTP protocol is simple enough to be modeled with a small state machine derived from the protocol specification, which can serve as a *ground truth* for our analysis. In particular, we consider as reference the model by Antunes and Neves (2011) based on the RFC 959, shown in Fig. 8. The initial states of the protocol represent the phases of user authentication ($S1$, $S2$, $S3$); most of the commands do not affect the protocol state (e.g., reading or updating the configuration of the server), as they leave the server in state $S4$; only a small set of commands (e.g., REST for resuming a transfer) introduce additional states.

Among the four FTP servers from the benchmark, we focus on LightFTP. In order to interpret the state machine inferred by StateAFL, we manually analyze both the source code of the server and the inputs that covered the states. This implementation of FTP is the simplest one and amenable for our manual analysis, as the core of the protocol implementation (not considering the parsing of the configuration file and of the command line) consists of about 1.7 kLoCs, and is limited to only one source file and one header file (`ftpserv.c` and `ftpserv.h`), thus mitigating the risk of an incorrect manual interpretation. The long-lived state of the server is all included within one data structure (`FTPCONTEXT`) allocated on the stack, which contains the socket handles, IP port numbers and addresses for the client and the server, the currently-opened file and access mode, the current working directory, and handles for a worker thread and for a mutex. Among the FTP commands implemented in this server, only few ones update the state of the server. Differing from the state machine based on the standard (Fig. 8), some commands have side effects on long-lived data, since they launch a worker thread to access the file system and to transfer data back to the client through a separate connection (e.g., the LIST and MLSD commands for listing the contents of a folder; the STOR, RETR, and APPE commands for file transfer); other commands,

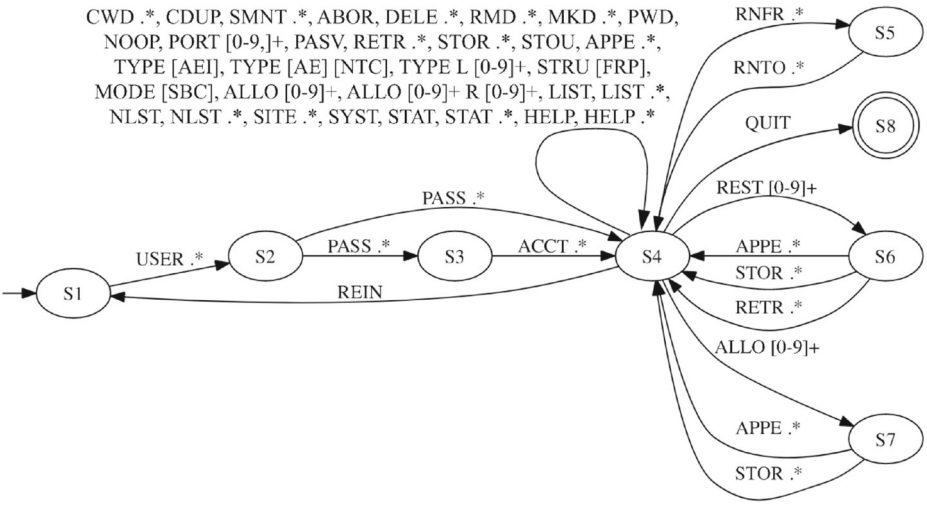


Fig. 8 Reference state machine for the FTP protocol, by Antunes and Neves (2011)

such as PORT, PASV, EPSV, and PBSZ change the configuration of the server (e.g., the client port to be used for the data connection).

Figure 9 shows a protocol state machine inferred by StateAFL for the LightFTP server. The state machine starts from a "dummy" initial state 0; the other states represent unique in-memory states of long-lived data in the target server, identified by an incremental number; an edge represents a request/reply pair between the server and the client; the edges can be self-transitions in the same state, which is the case of messages without side effects (e.g., read-only operations); or, the edges can bring the server to a different state, which reflect changes in long-lived data. As in the reference model, the states 0, 1 and 2 are followed during user authentication. In the case of unsuccessful authentication, the state machines moves to state 7. Most of the commands are stateless as in the reference model, and are represented by the self-transition in state 2. In this case, StateAFL correctly recognizes that the server is not changing state from the analysis of process memory, thus avoiding to add more states when stateless commands are issued. The server moves to the other states in the case of the PORT command (states 3, 6, and 10) and the LIST command (states 4, 5, 8, 11). Ideally, the state machine should use fewer states to represent the conditions that the data connection has been configured (PORT) and that the worker thread has been launched (LIST). However, the contents of the long-lived data vary across different executions of these commands, since the data depend on the parameters of the PORT command, and on non-determinism in the initialization of the worker thread. StateAFL performs locality-sensitive hashing to cluster these different contents of the long-lived data into few states of the inferred state machine, thus limiting the growth of redundant states.

To get more insights about the state machines inferred by AFLNet and StateAFL, we analyzed the overlap between their state machines for the LightFTP server. Figure 10a and b show respectively the state machine for StateAFL and AFLNet, by running the same two seed inputs. These inputs establish a session by logging-in, and perform basic operations

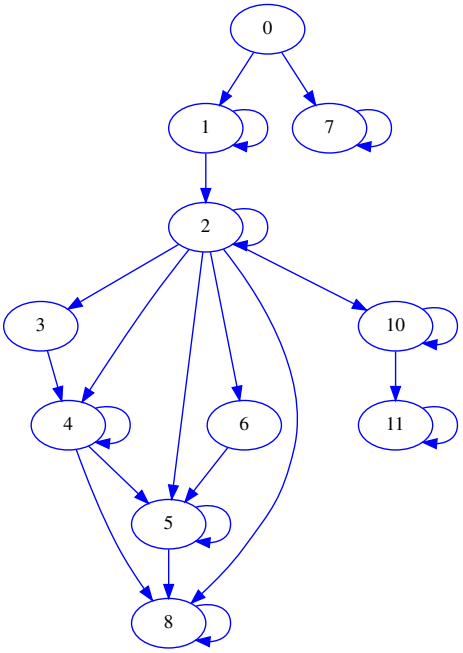
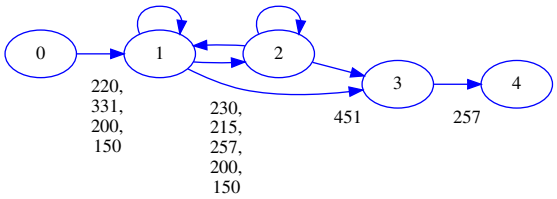
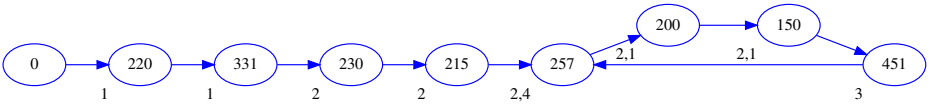


Fig. 9 State machine for LightFTP inferred by StateAFL after 24 h



(a) StateAFL.



(b) AFLNet.

Fig. 10 Inferred state machine for LightFTP using two seed inputs. States are annotated with the corresponding states for the other fuzzer

such as listing files in the root folder, querying the OS version of the server, setting the data connection, creating a folder, and quitting the session. In each figure, the states inferred by one fuzzer are annotated (nearby the vertex of the graph) with the label of the corresponding state of the other state machine. The set of status codes returned by the server (i.e., 220, 331, etc., used by AFLNET, in Fig. 10b) is larger than the set of unique in-memory states reached by the target server (i.e., 1, 2, 3, and 4 inferred by StateAFL, in Fig. 10a). This overlap highlights an important difference between status codes and the concept of "state". The status code only reflects the outcome of the most recent command (i.e., the latest request/reply iteration), regardless of which operations were previously performed on the server, which side effects have (or have not) accumulated within the target process, and how the server will behave in response to future commands. In this example, several commands return different status codes but do not have side effects on long-lived data of the server, such as the self-transitions in states 1 and 2 in Fig. 10a. The additional states inferred by AFLNET are redundant for stateful fuzzing, since applying the same fuzzed message starting from any of the redundant states (e.g., 230, 215, 257, 200, and 150 in Fig. 10b) results in the same behavior of the server, since the in-memory state of the process is always the same. In turn, these additional states result in wasted attempts by AFLNET to fuzz the server under (apparently) different conditions.

Table 4 provides statistics about the inferred protocol state machines, for both AFLNET and StateAFL, over all of the 13 target servers. The values in the table are the mean across repetitions. In the case of AFLNET, the vertexes represent the "status" code returned by the server in a request/reply iteration, while in StateAFL the vertexes represent unique memory states. The number of states for StateAFL is lower than AFLNET for almost all of the targets. Therefore, the other metrics in Table 4 also tend to be lower for StateAFL (number of edges; longest distance between the root node and other nodes; the degree of output transitions from a state; the number of circuits). StateAFL infers recurring states for most of the protocols, as for LightFTP in Fig. 10a.

AFLNET and StateAFL inferred different protocol state machines from the four FTP servers (LightFTP, Bftpd, Pure-FTPd, ProFTPD). Despite these servers implement the same protocol, it is typical for different implementations to cover a different subset of the protocol specification, or to include extensions from later standards or from the vendor (Antunes and

Table 4 Metrics about the inferred protocol state machines

target	fuzzer	vertexes	edges	longest dist. from root	out degree	circuits
Bftpd	AFLNET	24	183	4	7	>1M
	STATEAFL	3	6	1	2	3
Dcmtk	AFLNET	4	3	1	1	1
	STATEAFL	15	31	1	1	13
Dnsmasq	AFLNET	88	278	5	3	>1M
	STATEAFL	52	145	5	2	>20K
Exim	AFLNET	12	57	3	4	353
	STATEAFL	21	45	3	2	19
Forked-daapd	AFLNET	7	19	2	2	6
	STATEAFL	4	4	1	1	1
Kamailio	AFLNET	13	105	1	7	>315K
	STATEAFL	2	2	1	1	1
LightFTP	AFLNET	23	176	3	7	>1M
	STATEAFL	11	26	3	2	17
Live555	AFLNET	10	75	2	7	>37K
	STATEAFL	16	31	2	1	12
OpenSSH	AFLNET	111	246	8	2	>250K
	STATEAFL	153	467	4	3	>1M
OpenSSL	AFLNET	17	26	4	1	5
	STATEAFL	2	2	1	1	1
ProFTPD	AFLNET	26	241	4	9	>1M
	STATEAFL	4	9	1	2	5
Pure-FTPd	AFLNET	29	294	4	10	>1M
	STATEAFL	10	29	1	2	15
TinyDTLS	AFLNET	7	19	2	2	15
	STATEAFL	9	18	1	1	6

Neves 2011; Poll et al. 2015). In all cases, the state machines inferred by STATEAFL have a lower number of states and closer to the reference model of Fig. 8.

In two cases (Kamailio and OpenSSL), the protocol state machine inferred by STATEAFL consists of only two states, including the dummy state 0 and only one, fixed state over the course of the session. In the case of Kamailio, the default configuration only performs stateless routing of SIP requests, which are only based on the contents of the request; stateful processing needs to be enabled by configuring an optional module, as it is more resource-demanding and aimed for advanced use cases (Nick vs Networking 2019). By analyzing the dumps collected by STATEAFL, we found that long-lived data indeed do not change across iterations for these servers. In the case of OpenSSL, the fuzzer could not identify new states, since it is a highly-structured binary protocol, for which it is difficult to generate new sequences of valid messages. This is a general limitation of mutation-based fuzzers, that could be addressed by means of structure-aware fuzzing techniques (Serebryany et al. 2017). In both cases, STATEAFL can detect that the inputs hit the same state, thus avoiding to perform redundant tests.

5.3 Performance

Finally, we evaluated the performance overhead of the instrumentation code injected by StateAFL in the target process. Figure 11 reports the execution time of the target servers when running seed inputs. The execution time under StateAFL is normalized with respect to the execution time without instrumentation (e.g., a 1.1x slowdown means that the execution takes 10% more time to complete). The instrumentation code mainly consists of: (i) the probes injected where the target server allocates memory and performs network I/O, to make callbacks for data collection (Algorithms 2 to 6); (ii) post-execution analysis (Algorithms 7 and 8). Therefore, we separately evaluate the impact of these two types of instrumentation code.

Figure 11 shows the slowdown respectively when only the probes are injected without any post-execution analysis (labeled with *Probes*), and when the instrumentation code also includes the post-execution analysis (labeled with *Full*). For some targets, the relative slowdown is negligible (i.e., close to 1×). The relative slowdown is noticeable for those target servers that take less time to execute the inputs, and that allocate a larger amount of long-lived data to be analyzed. In these cases, the slowdown was around 1.5× the execution time of the non-instrumented server, and around 3× in the worst case of Dcmtk. In these cases, the non-instrumented execution time takes less than 100 ms to process the inputs. Most of the slowdown comes from the post-execution analysis, which computes hashes from memory snapshots. This analysis takes fractions of ms in the best cases, and around 100 ms in the worst cases. Instead, for those targets that take longer to process the inputs (e.g., Forked-daapd), the relative weight of the post-execution analysis becomes negligible. Moreover, the slowdown caused by StateAFL is balanced by a reduction of redundant states in the inferred state machines, leading to less states to be explored by fuzzing and less wasted inputs, thus achieving similar or better code coverage.

Finally, Fig. 12 reports the throughput of the fuzzers, in terms of executions of the target server per second, averaged over 24 h of fuzzing and 4 repetitions. The AFLNwe achieved the highest throughput across most of the benchmark targets. Compared to the other two fuzzers, AFLNwe is *not* a message-oriented fuzzer, as it sends fuzz inputs as an uninterrupted stream of bytes. Instead, AFLNet and StateAFL are message-oriented fuzzers, which alternate between sending request messages and receiving (and analyzing) response

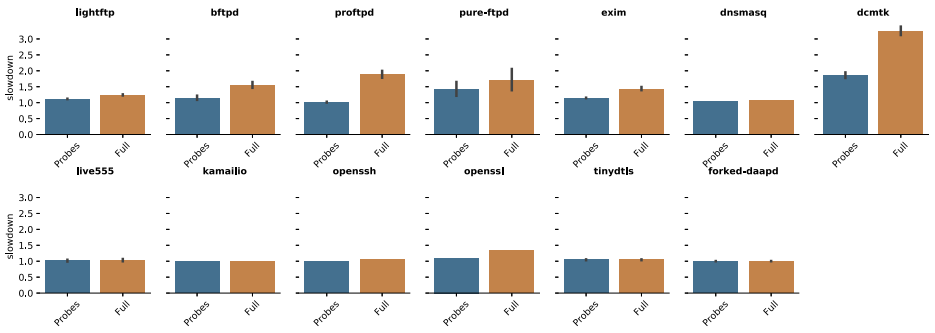


Fig. 11 Slowdown of execution time under StateAFL, respectively when the target is only instrumented with callbacks for data collection (*probes*), and when the instrumentation also performs post-execution analysis (*full*). The slowdown is normalized with respect to execution without instrumentation

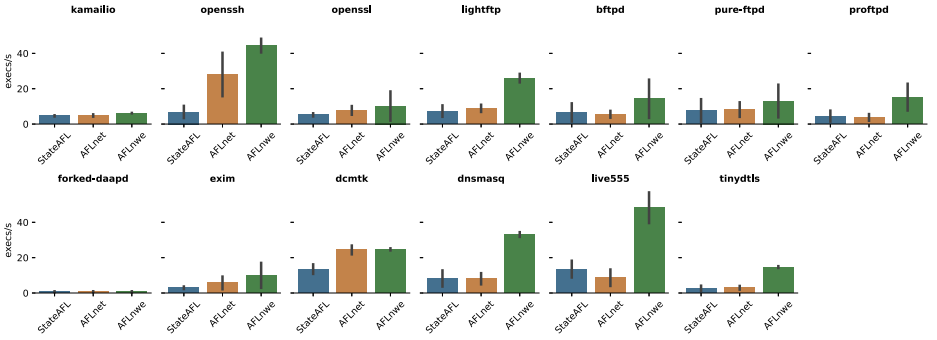


Fig. 12 Fuzzing throughput (executions of the target server per second), averaged over 24 h of fuzzing and 4 repetitions

messages, which introduces short delays. AFLNET and STATEAFL achieved a comparable fuzzing throughput for most of the benchmark targets. For few targets (OpenSSH and Dcmtk), STATEAFL exhibited a significantly lower fuzzing throughput than AFLNET. In the case of Dcmtk, we can attribute this gap to the combined effect of short absolute execution time of the target, and the higher slowdown caused by the instrumentation (Fig. 11). For OpenSSH, the lower throughput was caused by an additional delay between request messages, which was configured to make the analysis of in-memory states more deterministic. A potential extennsion to avoid the need for such delays, and in general for improving the fuzzing throughput, is represented by ongoing research on *snapshot-based fuzzing*, which saves and restores the state of the entire server process at selected times (Li et al. 2022; Andronidis and Cadar 2022). Please note that snapshot-based fuzzing is complementary area of research to STATEAFL, which infers states from a fine-grained analysis of process memory, and would guide the snapshot-based process by identifying unique application-level states.

6 Conclusion

This paper presented STATEAFL, a coverage-driven fuzzer for stateful network servers. We designed the fuzzer to not rely on manual customizations for the protocol under test, in order to make stateful fuzzing more broadly applicable. The fuzzer leverages compile-time instrumentation to insert probes, which take snapshots of long-lived data at each protocol iteration. Then, the fuzzer uses fuzzy hashing to map the snapshots to a unique state identifier, in order to infer protocol states.

We implemented and released STATEAFL as open-source software, and experimentally evaluated it on a benchmark of network servers. The experimental evaluation showed that STATEAFL can match a protocol-custom fuzzer in terms of both code coverage and vulnerabilities, and can even exceed it for some targets. Moreover, STATEAFL only introduces a limited overhead on the execution of the server under test. We also presented a qualitative analysis of the states inferred by STATEAFL. The qualitative analysis pointed out an important insight about stateful fuzzing: the response codes returned by many protocols are often not representative of the current state of the server, but only reflect the outcome of the last request. For this reason, inferring states for response codes can significantly inflate the

protocol state machine, leading to redundant fuzz tests. Having knowledge about the actual state of the server can be exploited by the fuzzer to avoid the redundant tests.

We expect that future work on stateful network protocol fuzzing will develop new solutions based on this observation. Potential directions for future research include new solutions from inferring states from memory analysis, such as by using techniques for static and dynamic program analysis, and new heuristics for generating fuzz inputs tailored for stateful protocols, such as algorithms for selecting which protocol states to fuzz and which portions of the input to mutate. Early work on these areas include Ba et al. (2022) and Li et al. (2022) for efficient definition of states based on program analysis, and Liu et al. (2021) on state selection algorithms. We also leave to future work the application of the proposed approach to binary-only programs. Since the instrumentation is limited to identifying and changing calls to library APIs, without changing the control and data flow, binary rewriting techniques represent good candidates for further research on this aspect (Dinesh et al. 2020; Duck et al. 2020). Finally, we expect future work to explore more applications of stateful fuzzing beyond network protocols, such as for the security testing of local stateful applications based on inter-process communication.

Acknowledgements I am grateful to Van-Thuan Pham (University of Melbourne) for the constructive discussions and the encouragement during this work. This work has been partially supported by the Google Cloud research credits program, and by the FRA programme (project OSTAGE) at Università degli Studi di Napoli Federico II.

Funding Open access funding provided by Università degli Studi di Napoli Federico II within the CRUI-CARE Agreement.

Data Availability The software presented in this paper is available as open-source software on https://github.com/stateafl/stateafl and https://github.com/profuzzbench/profuzzbench.

Declarations

Conflict of Interest No conflict of interest exits in the submission of this manuscript.

References

AFLplusplus Project (2021) Pull request #1200—rename. https://github.com/AFLplusplus/AFLplusplus/pull/1200/commits

Ali M, Hagen J, Oliver J (2020) Scalable malware clustering using multi-stage tree parallelization. In: 2020 IEEE International conference on intelligence and security informatics (ISI). IEEE, pp 1–6

Alrahem T, Chen A, DiGiuseppe N, Gee J, Hsiao SP, Mattox S, Park T et al (2007) Interstate: a stateful protocol fuzzer for SIP. Defcon 15:1–5

Andronidis A, Cadar C (2022) SnapFuzz: an efficient fuzzing framework for network applications. arXiv:220104048

Antunes J, Neves N (2011) Automatically complementing protocol specifications from network traces. In: Proceedings of the 13th European workshop on dependable computing, pp 87–92
Ba J, Böhme M, Mirzamomen Z, Roychoudhury A (2022) Stateful Greybox Fuzzing. arXiv:220402545
Banks G, Cova M, Felmetsger V, Almeroth K, Kemmerer R, Vigna G (2006) SNOOZE: toward a stateful NetwOrk prOtocol fuzZEr. In: International conference on information security, pp 343–358
Beyond Security (2020) beSTORM Black Box testing. https://beyondsecurity.com/solutions/bestorm.html. Online; Accessed 10 Dec 2020
Boehme M, Cadar C, Roychoudhury A (2021) Fuzzing: challenges and reflections. IEEE Softw 38(3):79–86
Böhme M, Falk B (2020) Fuzzing: on the exponential cost of vulnerability discovery. In: Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 713–724
Bozkaya T, Ozsoyoglu M (1997) Distance-based indexing for high-dimensional metric spaces. In: Proceedings of the 1997 ACM SIGMOD international conference on management of data, pp 357–368
Bozkaya T, Ozsoyoglu M (1999) Indexing large metric spaces for similarity search queries. ACM Trans Database Syst (TODS) 24(3):361–404
Caballero J, Yin H, Liang Z, Song D (2007) Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM conference on computer and communications security, pp 317–329
Cady CJ (2017) A tree locality-sensitive hash for secure software testing. Tech. rep. Air Force Institute of Technology
Comparetti PM, Wondracek G, Kruegel C, Kirda E (2009) Prospex: protocol specification extraction. In: 2009 30th IEEE symposium on security and privacy. IEEE, pp 110–125
De Ruiter J, Poll E (2015) Protocol state fuzzing of TLS implementations. In: 24th USENIX security symposium, pp 193–206
Dinesh S, Burow N, Xu D, Payer M (2020) Retrowrite: statically instrumenting COTS binaries for fuzzing and sanitization. In: 2020 IEEE symposium on security and privacy (SP). IEEE, pp 1497–1511
Duchene J, Le Guernic C, Alata E, Nicomette V, Kaâniche M (2018) State of the art of network protocol reverse engineering tools. J Comput Virol Hacking Tech 14(1):53–68
Duck GJ, Gao X, Roychoudhury A (2020) Binary rewriting without control flow recovery. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation, pp 151–163
Feng X, Sun R, Zhu X, Xue M, Wen S, Liu D, Nepal S, Xiang Y (2021) Snipuzz: black-box fuzzing of IoT firmware via message snippet inference. In: ACM conference on computer and communications security
Fiterau-Brostean P, Jonsson B, Merget R, de Ruiter J, Sagonas K, Somorovsky J (2020) Analysis of DTLS implementations using protocol state fuzzing. In: 29th USENIX security symposium
Google Inc (2021) OSS-Fuzz: continuous fuzzing for open source software. https://github.com/google/oss-fuzz. (Online; Accessed 10 Jan 2021)
Guo M, Wang G, Hata H, Babar MA (2021) Revenue maximizing markets for zero-day exploits. Auton Agent Multi-Agent Syst 35(2):1–29
Harman M, O'Hearn P (2018) From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. In: 2018 IEEE 18th international working conference on source code analysis and manipulation (SCAM). IEEE, pp 1–23
Hawkes B (2019) 0day "In the Wild". https://googleprojectzero.blogspot.com/p/0day.html, (Online; Accessed 10 Jan 2021)
Holzmann GJ, Lieberman WS (1991) Design and validation of computer protocols, vol 512. Prentice Hall, Englewood Cliffs
Jafari O, Maurya P, Nagarkar P, Islam KM, Crushev C (2021) A survey on locality sensitive hashing algorithms and their applications. arXiv:210208942
Kleber S, Kopp H, Kargl F (2018) NEMESYS: network Message syntax reverse engineering by analysis of the intrinsic structure of individual messages. In: 12th USENIX workshop on offensive technologies (WOOT)
Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. In: ACM conference on computer and communications security, pp 2123–2138
Korczynski D, Korczynski A (2021) Fuzzing 100+ open source projects with OSS-Fuzz—lessons learned. https://adalogics.com/blog/fuzzing-100-open-source-projects-with-oss-fuzz. (Online; Accessed 10 Jan 2021)
Li J, Li S, Gang S, Chen T, Yu H (2022) SNPSFUzzer: a fast greybox fuzzer for stateful network protocols using snapshots. arXiv:220203643
Liu D, Pham VT, Ernst G, Murray T, Rubinstein BI (2021) State selection algorithms and their impact on the performance of stateful network protocol fuzzing. arXiv:211215498

Manès VJM, Han H, Han C, Cha SK, Egele M, Schwartz EJ, Woo M (2019) The art, science, and engineering of fuzzing: a survey. IEEE Trans Softw Eng

Metzman J, Szekeres L, Simon L, Sprabery R, Arya A (2021) Fuzzbench: an open fuzzer benchmarking platform and service. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 1393–1403

Natella R, Pham VT (2021) Profuzzbench: a benchmark for stateful protocol fuzzing. In: Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis, pp 662–665. https://doi.org/10.1145/3460319.3469077

Nick vs Networking (2019) Kamailio bytes—stateless SIP proxy. https://nickvsnetworking.com/kamailio-bytes-stateless-sip-proxy/

Oliver J, Cheng C, Chen Y (2013) TLSH—a locality sensitive hash. In: 2013 Fourth cybercrime and trustworthy computing workshop. IEEE, pp 7–13

O'Neill PH (2021) 2021 has broken the record for zero-day hacking attacks. https://www.technologyreview.com/2021/09/23/1036140/2021-record-zero-day-hacks-reasons/. (Online; Accessed 23 Sep 2021)

Pham VT, Böhme M, Roychoudhury A (2020) AFLNET: a greybox fuzzer for network protocols. In: International conference on software testing, verification and validation (testing tools track)

Poll E, De Ruiter J, Schubert A (2015) Protocol state machines and session languages: specification, implementation, and security flaws. In: 2015 IEEE security and privacy workshops. IEEE, pp 125–133

Rapid7 (2020) Metasploit vulnerability & exploit database. https://www.rapid7.com/db/?q=fuzzer&type=metasploit, (Online; Accessed 10 Dec 2020)

Serebryany K (2017) OSS-Fuzz—Google's continuous fuzzing service for open source software (invited talk). In: USENIX Security symposium

Serebryany K, Buka V, Morehouse M (2017) Structure-aware fuzzing. https://llvm.org/devmtg/2017-10/slides/Serebryany-Structure-awarezing

Somorovsky J (2016) Systematic fuzzing and testing of tls libraries. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pp 1492–1504

Synopsis Inc (2020) Defensics fuzz testing. https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html. (Online; Accessed 10 Dec 2020)

Wheeler DA (2020) How to prevent the next heartbleed. https://dwheeler.com/essays/heartbleed.html. (Online; Accessed 07 Nov 2022)

Zalewski M (2021) American fuzzy lop. https://lcamtuf.coredump.cx/afl/. (Online; Accessed 08 Jan 2021)