StateAFL: Greybox Fuzzing for Stateful Network Servers

Roberto Natella
roberto.natella@unina.it
Università degli Studi di Napoli Federico II
Italy

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 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 by analyzing snapshots of long-lived memory areas, and incrementally builds a protocol state machine for guiding fuzzing. 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 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.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**.

KEYWORDS

Fuzzing; Benchmarking; Network Protocols

1 INTRODUCTION

According to recent statistics [21, 30], 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 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 [19].

Fuzzing is a relevant security testing technique to identify and prevent such vulnerabilities. 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 [12, 17], file transfer and messaging protocols such as FTP, SMB, and SMTP [3, 11], and multimedia protocols such as SIP [2, 4] and RSTP [31]. 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.

Many stateful protocol fuzzing techniques and tools have been developed, but they can only be applied with a significant effort, which has prevented their widespread adoption so far. Existing *generation-based* fuzzers require detailed knowledge of the protocols and formal specifications written by human experts [5, 33, 36]; *learning-based* fuzzers infer the protocol state machine at a significant computational cost, and still require custom implementations of wrappers to efficiently abstract protocol messages [12, 17].

Coverage-driven (greybox) fuzzing techniques have recently emerged as a popular solution, as demonstrated by the widespread adoption of the AFL fuzzer and similar tools [6, 26, 27, 37]. For example, as of June 2021, OSS-Fuzz has found over 30,000 bugs in 500 open source projects [18, 34], with more and more open-source projects being integrated by the community [25]. 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 [16, 31]. These recent approaches infer protocol states by analyzing the contents of messages, using message parsers that are specifically developed for the protocol. Moreover, it is difficult for these approaches to fuzz many protocols that 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 to map each in-memory state to a unique protocol state. 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 publicly released StateAFL released as open-source software at https://github.com/stateafl/stateafl.

This paper also presents an experimental evaluation of StateAFL on a public benchmark of 13 open-source network servers, the largest experimental setup among stateful network fuzzing studies to the best of our knowledge, publicly available at https://github.com/profuzzbench/profuzzbench. 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 than previous solutions based on stateless coverage-driven fuzzing and on stateful, protocol-customized fuzzing. We also qualitatively analyze state information both from parsing status codes within response messages, and from inference based on long-lived data, and find that response codes can be a misleading representation of the protocol state, leading to redundant 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 results. Section 5 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 [5, 33, 36]. The model specifies both the format of protocol messages (e.g., field types, message separators, etc.) and their sequencing over a session [32], 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 [3].

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) [14, 24]. *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) [12, 17]. 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) [35]. More powerful solutions leverage static and dynamic binary analysis (e.g., taint propagation analysis) to achieve full automation [9, 11], but in practice these solutions are difficult to implement and to port across different systems, which limits their adoption [20].

Coverage-driven fuzzing techniques have been adopted by AFL, libFuzzer, and other derivative tools [26] 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 [31] 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 [16] 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* [22]. 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 [32].

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* [32]. As the session progresses, the current protocol state is updated accordingly. The fundamental loop can be summarized by the following simplified pseudo-code:

$long\text{-}lived\ data \leftarrow$ allocate()
while iterate indefinitely **do**
  $short\text{-}lived\ data \leftarrow$ allocate()
  $request \leftarrow$ 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 [28]. 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 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.

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.

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 run-time 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 Figure 3. These functions (Algs. 2 and 3) represent the current iteration using the global integer variable

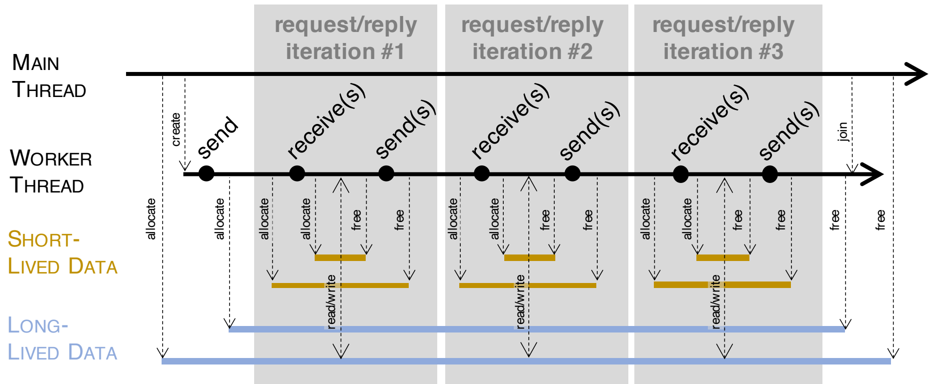


Figure 1: The fundamental loop of network servers.

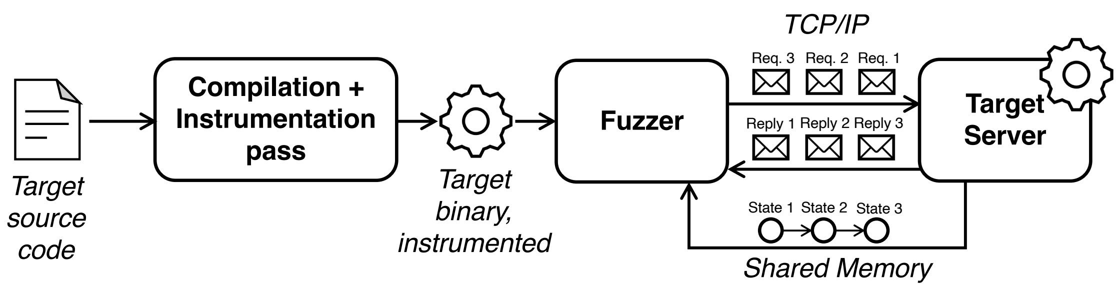


Figure 2: Overview of StateAFL.

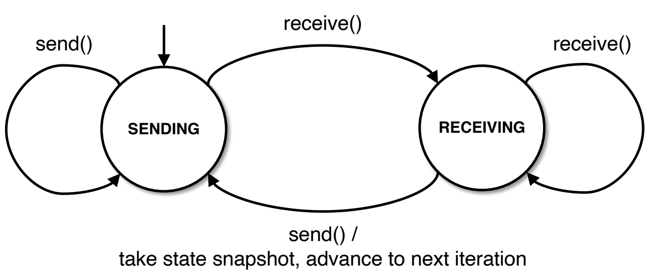


Figure 3: State machine to keep track of protocol iterations.

current_iter_no, allocated by the StateAFL library and initially set to 0 (Alg. 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 (Alg. 4). This function records the allocation using the ALLOC_RECORD data structure, which includes: (i) the number of the iteration at

---

Algorithm 1 Triggered when the server starts.

procedure ON_PROCESS_START
  alloc_records_map ← new map
  alloc_dumps_queue ← new queue
  current_iter_no ← 0
end procedure

Algorithm 2 Triggered when the server retrieves a message.

procedure ON_RECEIVE
  if state machine is *SENDING* **then**
    state machine ← *RECEIVING*
  end if
end procedure

Algorithm 3 Triggered when the server sends a message.

procedure ON_SEND
  if state machine is *RECEIVING* **then**
    state machine ← *SENDING*
    dump_current_state()
    current_iter_no ← *current_iter_no*+1
  end if
end procedure

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., 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 (Alg. 5). Still, the ALLOC_RECORD structure lasts until the termination of the network server. As an optimization, Alg. 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.

Algorithm 4 Triggered when the server allocates memory.

```
procedure ON_ALLOCATE(address, size)
  if current_iter_no = 0 then
    a ← new alloc_record

    a.iter_no_init ← current_iter_no
    a.iter_no_end ← −1
    a.addr ← initial address of allocated area
    a.size ← size of allocated area

    alloc_records_map.put(a.addr, a)
    zero-initialize the allocated area
  end if
end procedure
```

Algorithm 5 Triggered when the server frees memory.

```
procedure ON_FREE(address)
  a ← alloc_records_map.get(address)
  a.iter_no_end ← current_iter_no
  alloc_records_map.remove(a.addr)
end procedure
```

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 Figure 3). On this event, ON_SEND calls DUMP_CURRENT_STATE (Alg. 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, 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.

Algorithm 6 Takes snapshots of memory areas

```
procedure DUMP_CURRENT_STATE
  for all a ∈ alloc_records_map do
    d ← new alloc_dump
    d.iter_no_dumped ← current_iter_no
    d.record ← reference to a
    d.contents ← copy a.addr's contents

    alloc_dumps_queue.push(d)
  end for
end procedure
```

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 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,...) are not actually tracked by our algorithm, but are included in this discussion as an example of short-lived data.

In the example of Figure 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 Figure 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 (Alg. 7). In turn, it calls the SAVE_STATE_SEQ FUNCTION.

The SAVE_STATE_SEQ function (Alg. 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

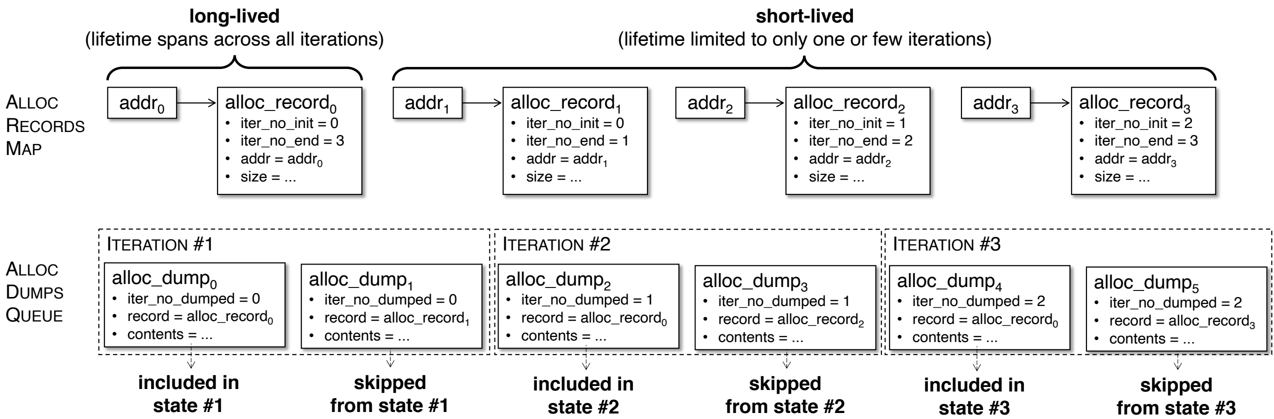


Figure 4: Example of data structures (*alloc_record* and *alloc_dump*), after an execution with three iterations, with one long-lived area allocated at the beginning and freed at the end, and three short-lived areas allocated and freed at each iteration.

Algorithm 7 Triggered when the server terminates.

procedure ON_PROCESS_END
  $total\_iterations \leftarrow current\_iter\_no$
  $save\_state\_seq()$
end procedure

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 Figure 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.

Alg. 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 Figure 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 that belong to the same 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 over-sensitive 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, the 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 [23]. 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 [1, 29]. 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 [10]. 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.

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 [1, 29]. 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 (Alg. 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

Algorithm 8 Generates a sequence of protocol states.

```
procedure SAVE_STATE_SEQ
  states_sequence ← new list of integers
  prev_iter_no ← 0
  tlsh_hash ← 0

  for all d ∈ alloc_dumps_queue do

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

    if d.iter_no_dumped > prev_iter_no then
      ▷ Save state of the current iteration before the next
      state_id ← get_state_id(tlsh_hash)
      states_sequence.push(state_id)
      tlsh_hash ← 0
    end if

    update_tlsh(tlsh_hash,d.contents)
    prev_iter_no ← d.iter_no_dumped
  end for

  state_id ← get_state_id(tlsh_hash)
  states_sequence.push(state_id)
  save_to_shared_memory(states_sequence)
end procedure

procedure GET_STATE_ID(tlsh_hash)
  radius ← ϵ
  state_id ← mvptree_lookup(tlsh_hash,radius)

  if state_id = ∅ then
    state_id ← mvptree_count()+1
    mvptree_add({tlsh_hash,state_id})
  end if

  return state_id
end procedure
```

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 [7, 8].

The distance threshold ϵ is dynamically calibrated for the target server under test. Before fuzzing, StateAFL performs a calibration stage, by executing the server multiple times using seed inputs, and by computing a sequence of hash values for each repetition. 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 values at each iteration, and collects the distances between the hash values. Then, it takes the median value of these distances as a conservative choice for ϵ that is neither too low (which would cause similar states with small differences to have different identifiers) nor too high (which would group different states into one). As a further way to prevent the growth of redundant states, the fuzzer monitors the frequency of new states added by GET_STATE_ID (Alg. 8). As the target server is fed with new mutated inputs, in most cases the server should visit again the same states, unless the inputs can discover a new corner case in the protocol. Therefore, if too many states are added within a short amount of fuzz inputs, the fuzzer further increases the ϵ threshold. As in the original proposal of TLSH, we vary 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.

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 [31], 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. The target state is selected according to how often states have been exercised (rarer ones are selected with higher probability), and how often they contributed to discover new states or to cover new paths. The mutations include both byte-level operators (e.g., bit-flipping, arithmetics, etc.) and message-level ones (e.g., replacement, insertion, duplication, and deletion of messages).

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 previously discussed. As the instrumentation is performed at compile-time, the current implementation is focused on targets where source code is available. Since the instrumentation is limited to identifying and changing calls to library APIs, without changing the control and data flow, the proposed approach could be even be applied to binary-only programs through binary rewriting techniques [13, 15].

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 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. We also include the global data area.

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 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 transmissions. 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.

4 EVALUATION

We evaluate StateAFL by fuzzing a set of real-world network servers from popular open-source projects. The evaluation includes a quantitative analysis of StateAFL, with respect to code coverage, vulnerabilities found, and performance overheads, and a qualitative analysis of protocol state inference.

4.1 Experimental setup

We performed experiments on ProFuzzBench, a public benchmark for network fuzzers [28]. The benchmark includes 13 open-source network servers (Table 1). 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.

Table 1: Benchmark targets

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

The experimental evaluation compares StateAFL with two baseline fuzzers. We adopted other fuzzers that: (i) are not limited to specific network protocols, but are applicable to a large set of network targets, including the ones in ProFuzzBench; (ii) 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 it 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 1). 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.

4.2 Experimental results

For each of the 13 target servers and of the 3 fuzzers, we ran 4 repeated fuzzing experiments, for a total of 156 experiments. In each experiment, the target was fuzzed for 24 hours. We executed the experiments on the Google Cloud Platform, using E2 high-memory VM instances with 4 vCPUs, with a dedicated vCPU for each replication. The experiments were executed using isolated Docker containers, based on the ProFuzzBench automation scripts.

Figure 5 shows the line coverage for each target and for each fuzzers after 24 hours. For the sake of space, edge coverage is not showed, as it exhibits similar results to line coverage. 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. For some these targets (LightFTP, TinyDTLS, Dcmtk, Dnsmasq, Live555, Kamailio), Both StateAFL and AFLnet were able to find the same crashing inputs, within a comparable amount of time.

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 cases, the stateful fuzzers benefited from inferring protocol states. When a message succeeds at discovering a new state, the fuzzer 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.

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.

Figure 6 shows an example of protocol state machine inferred by StateAFL for the LightFTP target 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. Some states (e.g., states 2, 3, and 9 in Figure 6) have a larger number of input transitions, as they represent frequently-recurring conditions of the target server. These states point out that the fuzzer is able to recognize protocol states from the analysis of process memory. If such frequently-visited states were missing, the fuzzer would be adding new states even when the server repeats again the same protocol behavior. For example, had not the fuzzer tolerated non-deterministic changes in long-lived data (e.g., the memory address contained within pointer variables), it would have been adding a larger number of states, without visiting again the previous states, and the topology of the inferred state machine would be "flat" without loops. Moreover, having redundant states decreases the efficiency of stateful fuzzing, as the fuzzer performs redundant tests from two states without actual differences in the server behavior.

Table 2: 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	4	10	2	2	6
Dcmtk	AFLnet	4	3	1	1	1
	StateAFL	2	1	1	1	1
Dnsmasq	AFLnet	88	278	5	3	>1M
	StateAFL	34	132	3	3	>500K
Exim	AFLnet	12	57	3	4	353
	StateAFL	10	33	3	3	152
Forked-daapd	AFLnet	7	19	2	2	6
	StateAFL	6	7	1	1	2
Kamailio	AFLnet	13	105	1	7	>315K
	StateAFL	2	2	1	1	1
LightFTP	AFLnet	23	176	3	7	>1M
	StateAFL	21	115	3	5	>316K
Live555	AFLnet	10	75	2	7	>37K
	StateAFL	6	30	2	4	128
OpenSSH	AFLnet	111	246	8	2	>250K
	StateAFL	63	205	4	3	>401K
OpenSSL	AFLnet	17	26	4	1	5
	StateAFL	5	7	1	1	2
ProFTPD	AFLnet	26	241	4	9	>1M
	StateAFL	18	115	2	6	>503K
Pure-FTPd	AFLnet	29	294	4	10	>1M
	StateAFL	11	46	2	4	816
TinyDTLS	AFLnet	7	19	2	2	15
	StateAFL	25	63	2	2	43

Table 2 provides statistics about the inferred protocol state machines, for both AFLnet and StateAFL, over 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 2 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 Figure 6.

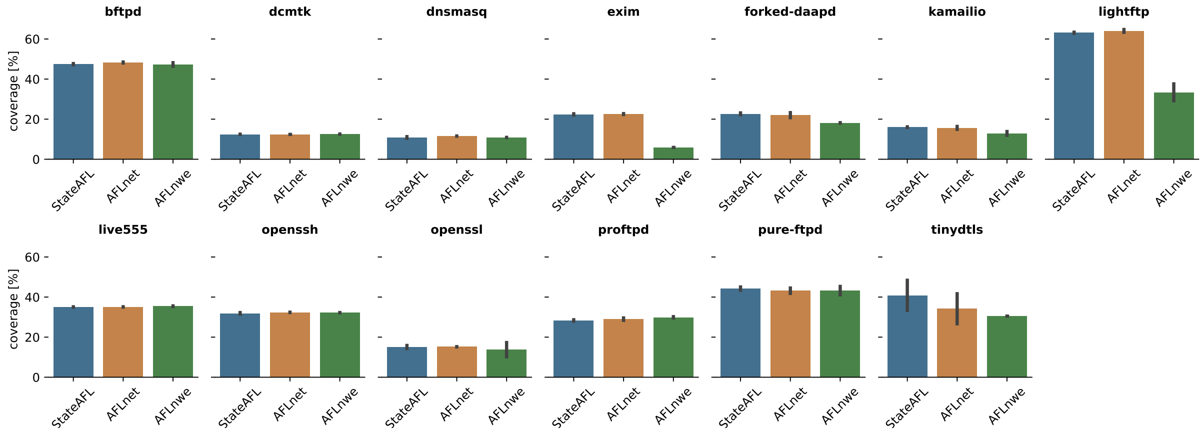


Figure 5: Line coverage after 24 hours of fuzzing.

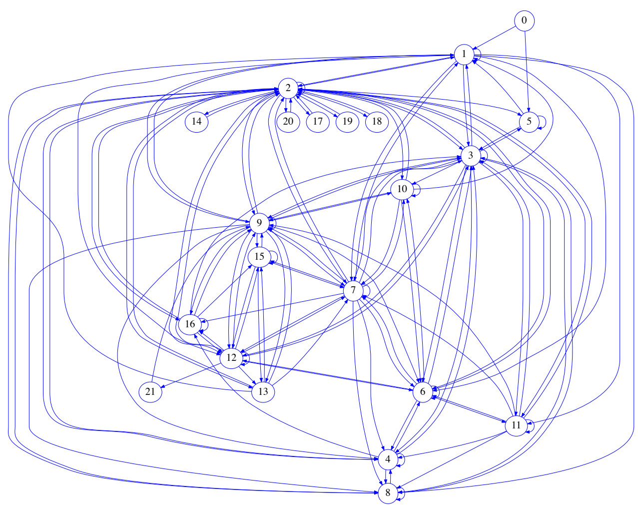
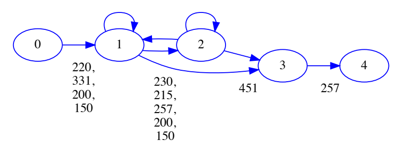


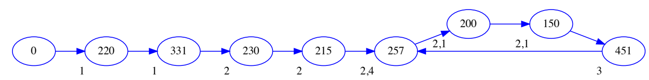
Figure 6: Protocol state machine for LightFTP inferred by StateAFL after 24 hours of fuzzing.

In two cases (Kamailio and Dcmtk), 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. Dcmtk runs a stateless process, which stores and retrieves files from the local filesystem. 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. By analyzing the dumps collected by StateAFL, we found that long-lived data indeed do not change across iterations for these servers.

Both AFLnet and StateAFL obtained 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 different subset of the protocol specification, or to include extensions from later standards or from the vendor [3, 32]. In the case of StateAFL, the differences between protocol implementations are wider than for AFLnet, since the inferred state machine follows the internals of the server process. In all FTP implementations, the servers store long-lived session information in global or dynamic memory, but some implementations store additional information, such as the path of the current working directory and the port number for the data connection.



(a) StateAFL.



(b) AFLnet.

Figure 7: Inferred state machine for LightFTP using two seed inputs. States are annotated with the corresponding states for the other fuzzer.

To get more insights about the different state machines inferred by AFLnet and StateAFL, we analyzed the overlap between their state machines for the LightFTP server. Figure 7a and Figure 7b 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 typical operations 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., as in Figure 7b) is larger than the set of unique in-memory states reached by the target server (i.e., 1, 2, 3, and 4 in Figure 7a). 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

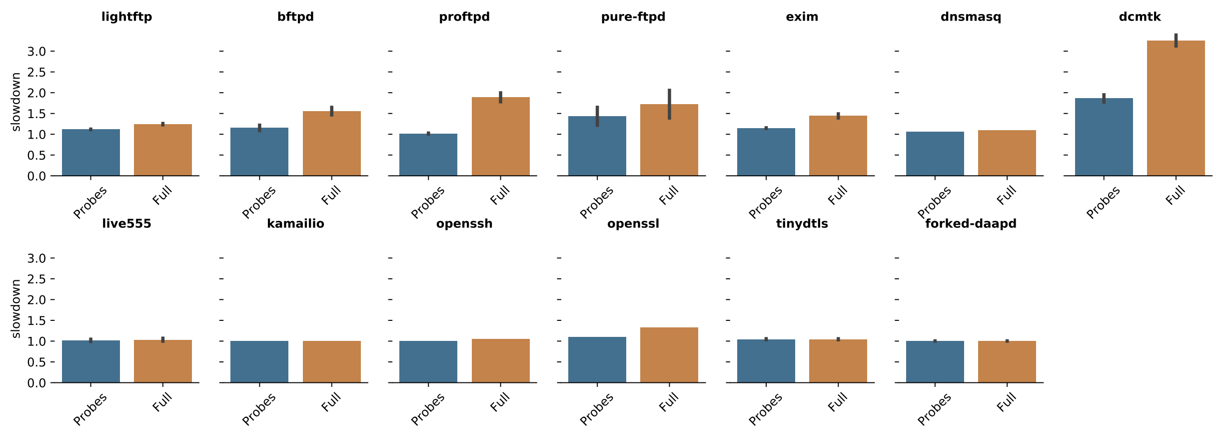


Figure 8: Slowdown of execution time under StateAFL, respectively when the target is only instrumented with callbacks for data collection (*probes*), and when the instrumentation also performs post-execution analysis (*full*). The slowdown is normalized with respect to execution without instrumentation.

iteration), regardless of which operations were previously performed on the server, and which side effects have (or have not) accumulated within the target process. 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 Figure 7a. 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 Figure 7b) results in the same behavior of the server, since the in-memory state of the process is always the same. In turn, this results in wasted fuzzing attempts from (apparently) different states.

Finally, we evaluated the performance overhead of the instrumentation code injected by StateAFL in the target process. Figure 8 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 8 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 1x). 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 1.5x the execution time of the non-instrumented server, and around 3x in the worst case of Dcmtk. In these cases, the non-instrumented execution time takes less than 100ms 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 100ms 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.

5 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 system-under-test, in order to make stateful fuzzing more broadly applicable. The fuzzer leverages compile-time instrumentation to insert probes. At run-time, the probes take memory snapshots at each protocol iteration. Finally, when the server terminates, the fuzzer analyze the snapshots for long-lived data, and uses a fuzzy hash function to map the memory contents to a unique state identifier, in order to infer protocol states. We applied StateAFL on 13 open-source, diverse network servers to show its applicability. The fuzzer has been able to achieve comparable, or even better code coverage than other greybox fuzzers that rely on custom protocol parsers. The performance overhead is reasonably low, and balanced by the reduction of redundancies in the inferred protocol state machine. In fact, our qualitative analysis pointed out that relying on status codes from messages do not reflect the current state of the server, leading to redundant tests.

ACKNOWLEDGMENTS

I am grateful to Van-Thuan Pham for the constructive discussions and the encouragement during this project. This work has been partially supported by the Google Cloud research credits program.

REFERENCES

[1] Muqeet Ali, Josiah Hagen, and Jonathan Oliver. 2020. Scalable malware clustering using multi-stage tree parallelization. In *2020 IEEE International Conference on Intelligence and Security Informatics (ISI)*. IEEE, 1–6.
[2] T. Alrahem, A. Chen, N. DiGiuseppe, J. Gee, S.-P. Hsiao, S. Mattox, T. Park, et al. 2007. Interstate: A stateful protocol fuzzer for SIP. *Defcon* 15 (2007), 1–5.
[3] João Antunes and Nuno Neves. 2011. Automatically complementing protocol specifications from network traces. In *Proceedings of the 13th European Workshop*

on Dependable Computing. 87–92.

[4] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *Intl. Conf. on Information Security*. 343–358.

[5] Beyond Security. 2020. beSTORM Black Box Testing. https://beyondsecurity.com/solutions/bestorm.html. [Online; accessed 12-10-2020].

[6] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Softw.* 38, 3 (2021), 79–86.

[7] Tolga Bozkaya and Meral Ozsoyoglu. 1997. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. 357–368.

[8] Tolga Bozkaya and Meral Ozsoyoglu. 1999. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems (TODS)* 24, 3 (1999), 361–404.

[9] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*. 317–329.

[10] Camdon J. Cady. 2017. *A Tree Locality-Sensitive Hash for Secure Software Testing*. Technical Report. Air Force Institute of Technology.

[11] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol specification extraction. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 110–125.

[12] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symp.* 193–206.

[13] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[14] Julien Duchene, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. 2018. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques* 14, 1 (2018), 53–68.

[15] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 151–163.

[16] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minghui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *ACM Conference on Computer and Communications Security*.

[17] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *29th USENIX Security Symp.*

[18] Google, Inc. 2021. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. [Online; accessed 01-10-2021].

[19] Mingyu Guo, Guanhua Wang, Hideaki Hata, and Muhammad Ali Babar. 2021. Revenue maximizing markets for zero-day exploits. *Autonomous Agents and Multi-Agent Systems* 35, 2 (2021), 1–29.

[20] Mark Harman and Peter O'Hearn. 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, 1–23.

[21] Ben Hawkes. 2019. 0day "In the Wild". https://googleprojectzero.blogspot.com/p/0day.html. [Online; accessed 01-10-2021].

[22] Gerard J Holzmann and William Slattery Lieberman. 1991. *Design and validation of computer protocols*. Vol. 512. Prentice hall Englewood Cliffs.

[23] Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. 2021. A Survey on Locality Sensitive Hashing Algorithms and their Applications. *arXiv preprint arXiv:2102.08942* (2021).

[24] Stephan Kleber, Henning Kopp, and Frank Kargl. 2018. NEMESYS: Network Message Syntax Reverse Engineering by Analysis of the Intrinsic Structure of Individual Messages. In *12th USENIX Workshop on Offensive Technologies (WOOT)*.

[25] David Korczynski and Adam Korczynski. 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 01-10-2021].

[26] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[27] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 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*. 1393–1403.

[28] Roberto Natella and Van-Thuan Pham. 2021. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 662–665. https://doi.org/10.1145/3460319.3469077

[29] Jonathan Oliver, Chun Cheng, and Yanggui Chen. 2013. TLSH–a locality sensitive hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, 7–13.

[30] Patrick Howell O'Neill. 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 23-09-2021].

[31] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *Intl. Conf. on Software Testing, Verification and Validation (Testing Tools Track)*.

[32] Erik Poll, Joeri De Ruiter, and Aleksy Schubert. 2015. Protocol state machines and session languages: specification, implementation, and security flaws. In *2015 IEEE Security and Privacy Workshops*. IEEE, 125–133.

[33] Rapid7. 2020. Metasploit Vulnerability & Exploit Database. https://www.rapid7.com/db/?q=fuzzer&type=metasploit. [Online; accessed 12-10-2020].

[34] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software (invited talk). In *USENIX Security Symposium*.

[35] Juraj Somorovsky. 2016. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1492–1504.

[36] Synopsis, Inc. 2020. Defensics Fuzz Testing. https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html. [Online; accessed 12-10-2020].

[37] Michal Zalewski. 2021. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. [Online; accessed 01-08-2021].