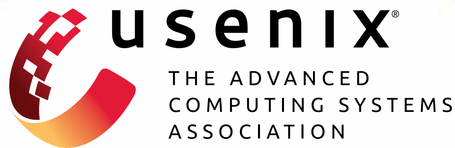

Stateful Greybox Fuzzing

Jinsheng Ba, *National University of Singapore;* Marcel Böhme, *Monash University and MPI-SP;* Zahra Mirzamomen, *Monash University;* Abhik Roychoudhury, *National University of Singapore*

https://www.usenix.org/conference/usenixsecurity22/presentation/ba

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the
31st USENIX Security Symposium is
sponsored by USENIX.

Stateful Greybox Fuzzing

Jinsheng Ba[1], Marcel Böhme[2,3], Zahra Mirzamomen[2], and Abhik Roychoudhury[1]

[1]National University of Singapore, [2]Monash University, [3]MPI-SP

Abstract

Many protocol implementations are reactive systems, where the protocol process is in continuous interaction with other processes and the environment. If a bug can be exposed only in a certain state, a fuzzer needs to provide a specific sequence of events as inputs that would take protocol into this state before the bug is manifested. We call these bugs as "stateful" bugs. Usually, when we are testing a protocol implementation, we do not have a detailed formal specification of the protocol to rely upon. Without knowledge of the protocol, it is inherently difficult for a fuzzer to discover such stateful bugs. A *key challenge* then is to cover the state space *without* an explicit specification of the protocol. Finding stateful bugs in protocol implementations would thus involve partially uncovering the state space of the protocol. Fuzzing stateful software systems would need to incorporate strategies for state identification. Such state identification may follow from manual guidance, or from automatic analysis.

In this work, we posit that manual annotations for state identification can be avoided for stateful protocol fuzzing. Specifically, we rely on a programmatic intuition that the state variables used in protocol implementations often appear in `enum` type variables whose values (the state names) come from named constants. In our analysis of the Top-50 most widely used open-source protocol implementations, we found that every implementation uses state variables that are assigned named constants (with easy to comprehend names such as INIT, READY) to represent the current state. In this work, we propose to automatically identify such state variables and track the sequence of values assigned to them during fuzzing to produce a "map" of the explored state space.

Our experiments confirm that our stateful fuzzer discovers stateful bugs twice as fast as the baseline greybox fuzzer that we extended. Starting from the initial state, our fuzzer exercises one order of magnitude more state/transition sequences and covers code two times faster than the baseline fuzzer. Several zero-day bugs in prominent protocol implementations were found by our fuzzer, and 8 CVEs have been assigned.

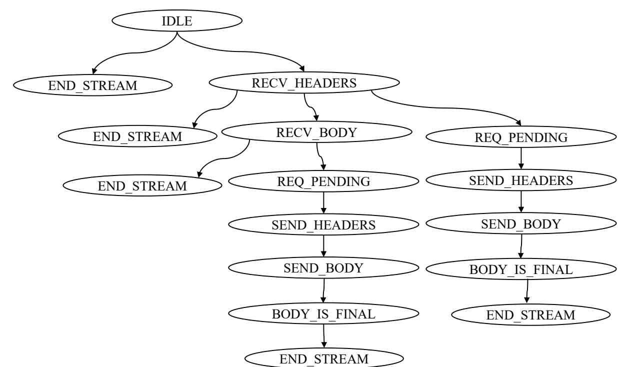

Figure 1: Dynamically constructed state transition tree (STT) for an HTTP2 protocol implementation (H2O).

1 Introduction

Protocol implementations are stateful software systems that require input messages to be sent in a certain expected order. Conventional greybox fuzzing approaches consider structured program inputs, which are subject to random mutations for exploring the space of program inputs. However, for protocols, the input needed to trigger a behavior is a *sequence* of messages, events, or actions. Thus for exposing corner cases or bugs in such systems, which we call stateful bugs, the fuzzing technique needs to explore an unknown state space by constructing and navigating suitable event sequences. We design, implement and evaluate such a fuzzer in this paper.

Existing coverage-guided greybox fuzzers (CGF) [1, 23] have been very successful at finding bugs, but they cannot effectively deal with a state space. For instance, the HTTP2 protocol requires a HEADER frame to be sent before the DATA frame. In the H2O implementation (Figure 1), a state variable is *set* to a certain value when the HEADER frame is received. The same state variable is *checked* when the DATA frame is received. Suppose, there is a bug in the response to a correctly

processed DATA frame. As there is a specific sequence of input messages needed to reveal the bug, we call this a *stateful bug*. Suppose also that the first generated input sends a DATA frame and then a HEADER frame. Clearly, the DATA frame cannot be processed; yet the code for the HEADER frame handler is already covered. A coverage-guided fuzzer has no information on the traversed state space. It is oblivious to the fact that the correct processing of the DATA frame is "unlocked" only when the requisite state is visited first (during the processing of the HEADER frame).

The *main objective* of a *stateful greybox fuzzer* should be to chart out and efficiently explore an unknown state space to discover stateful bugs. To this end, a stateful fuzzer (i) constructs a lightweight abstraction of the observed state space from the state feedback—in our case from the sequence of observed state variable values, and (ii) navigates this state space abstraction to maximize the probability of visiting unobserved states. In line with most existing efforts in greybox fuzzing, we assume that *no* external information is available, neither in the form of protocol specifications nor in the form of human annotations [2]. A stateful greybox fuzzer is pointed to the protocol implementation, fuzzes it, and reports any bugs that it finds. So, how can we enable a greybox fuzzer to effectively navigate an unknown state space?

Identifying state space. In this paper, we argue that protocols are often explicitly encoded using *state variables* that are assigned and compared to named constants.[1] State variables allow developers to implement state transitions by assigning the corresponding named constant, and to implement state-based program logic as `if`- or `switch`-statements. More specifically, using pattern matching, we identify state variables using enumerated types (`enums`). An *enumerated type* is a group of named constants that specifies all possible values for a variable of that type. Our *instrumentation* injects a call to our runtime at every program location where a state variable is assigned to a new value. Our *runtime* efficiently constructs the state transition tree (STT). The *STT* captures the sequence of values assigned to state variables across all fuzzer-generated input sequences, and as a global data structure, it is shared with the fuzzer. An example of the STT constructed for the H2O implementation of the HTTP2 protocol is shown in Figure 1. Our *in-memory fuzzer* uses the STT to steer the generated input sequences towards under-explored parts of the state space.

Stateful greybox fuzzing. We discuss several heuristics to increase the coverage of the state space via greybox fuzzing. First, we propose to add generated inputs to the seed corpus that exercise new nodes in the STT. As we will demonstrate, code coverage alone is insufficient to capture the order across different requests. Instead, we should capture the states in which the code is covered. Hence we argue, adding inputs that discover a new node in the STT facilitates a better coverage of the state space. Secondly, we propose to focus on the seeds which traverse the rarely visited nodes of the STT or whose offsprings are more likely to take a different path through the STT. We hypothesize that these heuristics will help the fuzzer to explore the state space. Finally, we propose an approach to focus particularly on the bytes in the seed, whose mutations trigger new nodes in the STT. The state space corresponding to the newly added nodes can be efficiently explored by mutating these bytes.

Results. We implemented our stateful greybox fuzzing approach into LIBFUZZER [1] and call our tool SGFUZZ (Stateful Greybox Fuzzer). We evaluated SGFUZZ against LIBFUZZER and AFLNET on eight widely used protocol implementations. The state sequence for an input is determined by the sequence of values assigned to the state variables during the execution of the protocol implementation. In our experiments, starting from the initial state, our stateful fuzzer exercises 33x more state sequences than LIBFUZZER 15x more than IJON, and 260x more than AFLNET. Observing that some code can be covered only in certain states, we found that SGFUZZ achieves the same branch coverage more than 2x faster than LIBFUZZER. By reproducing existing stateful bugs, we also found SGFUZZ exposes stateful bugs about twice as fast as LIBFUZZER and IJON, more than 155x faster than AFLNET. Most of all, SGFUZZ found 12 new bugs in widely used stateful systems, 8 of which were assigned CVEs. Our analysis further shows that stateful bugs are prevalent: every four in five bugs reported among our subjects are stateful. We have made our data set publicly available for review and will make an open-source release of our fuzzer to researchers and practitioners upon publication of the work.

In summary, we make the four key contributions:

- We propose an automatic method to identify and capture the explored state space of a protocol implementation.
- We present the design and implementation of SGFUZZ, a stateful greybox fuzzer that found 12 new bugs in widely-used and well-fuzzed programs.
- We conduct and present a comprehensive evaluation of SGFUZZ against the state-of-the-art and the baseline stateful greybox fuzzers.
- We make all data, scripts, and tools publicly available to facilitate reproducibility.

2 Motivating Example

We believe that current greybox or stateful fuzzers are ineffective in detecting stateful bugs. We use the stream state machine of HTTP2 protocol and the H2O implementation to explain the main reasons for their inefficiency to motivate our approach for stateful greybox fuzzing.

[1] We found that this observation holds for all of the top-50 most widely used protocol implementations as well as for every stateful protocol in the ProFuzzbench protocol testing benchmark [19].

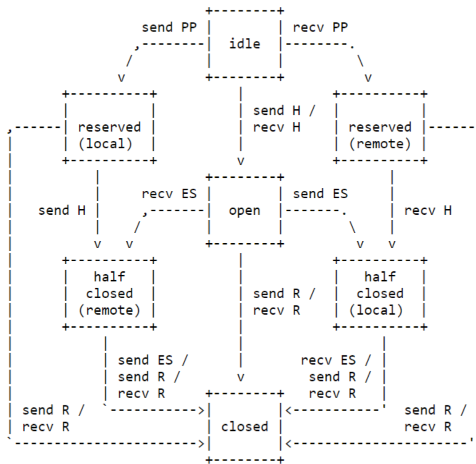

Figure 2: The HTTP2 stream state machine as per RFC 7540.

2.1 The HTTP2 Protocol

HTTP2 is a multiplexing protocol, in which each HTTP request is split into several frames corresponding to the HTTP header, HTTP body, and so on. A *frame* is the minimal data unit that is passed along the HTTP2 protocol. For instance, an HTTP2 request message may be split into a header frame and a few data frames. All frames belonging to the same HTTP request constitute a *stream*, having to be processed by the server in order. The HTTP2 protocol defines the *stream state machine*, which regulates the processing order of frames. When different frames are being processed, the server visits different states, e.g., the "receiving header" state when it receives the header frame and the "receiving body" state when receiving data frames.

Figure 2 shows the HTTP2 stream state machine as per the official specification (RFC 7540)[2]. In the initial *idle state*, the current or remote participant can send / receive a *push promise* frame (PP) or a header frame (H). A push promise frame simply marks a stream as *reserved* while a header frame marks the stream as *open*. In the open state, all frames are processed including *data* frames. An *end stream* (ES) frame will mark an open stream as *half-closed* and a half-closed stream as *closed*. In any state, if a *reset* (R) frame is sent / received, the stream is marked as closed.

2.2 The H2O Implementation

H2O is a new generation HTTP server that is very fast. H2O supports the HTTP, HTTP2, and HTTP3 protocols and is written in C programming language. We focus on HTTP2 implementation.

State Variables. H2O implementation uses a much *finer-grained HTTP2 stream state machine* than specified in Figure 2. Specifically, H2O tracks the different stages in which frames are received and responses are sent. H2O uses a dedicated variable to store the current protocol state (i.e., a state variable; here the enumeration variable `stream->state`). The eight implemented HTTP2 stream states are defined as *enumerated type* with the following named constants:

- **State (0).** `H2O_HTTP2_STREAM_STATE_IDLE`
- **State (1).** `H2O_HTTP2_STREAM_STATE_RECV_HEADERS`
- **State (2).** `H2O_HTTP2_STREAM_STATE_RECV_BODY`
- **State (3).** `H2O_HTTP2_STREAM_STATE_REQ_PENDING`
- **State (4).** `H2O_HTTP2_STREAM_STATE_SEND_HEADERS`
- **State (5).** `H2O_HTTP2_STREAM_STATE_SEND_BODY`
- **State (6).** `H2O_HTTP2_STREAM_STATE_SEND_BODY_IS_FINAL`
- **State (7).** `H2O_HTTP2_STREAM_STATE_END_STREAM`

At the start, the server is in the idle State (0) and waits for incoming requests. Depending on the frames received within the stream, the following states are visited: receiving request header (1), request body (2), sending response header (4), response body (5), and end of stream (7). State (3) is an intermediate state in which an incoming frame has not been assigned a handler, yet. State (6) is reached only when the client has indicated the end of the stream but there are still pending frames to be sent from the server.

> H2O uses *named constants* to keep track of the current HTTP2 protocol state. We use this insight to identify states.

2.3 Challenges of Fuzzing Stateful Software

If the protocol specification is unavailable, how can we automatically figure out the conditions required to exercise the state-dependent code? We make the following observations.

Uninformative Response Codes. The observable response code is unrelated to the current HTTP2 stream state. This renders ineffective the state-of-the-art fuzzer AFLNet [20] which depends on the response code to identify the current state.

Coverage is insufficient. In Figure 3, the function `handle_request_body_chunk` ④ is executed only if the current value of `stream->state` is `..RECV_BODY` when the data frame is handled ③. However, `stream->state` is set to `..RECV_BODY` ② only when the handler for the header frame is called beforehand with a valid header frame ①. Code coverage does not capture the required ordering. It is possible to cover both handler functions (① and ③) by sending a data frame first and a header frame next. However, this sequence of frames cannot be easily modified to cover the function `handle_request_body_chunk` ④. On the contrary, if the header frame is sent first and the data frame next, the same handler functions would be covered, but this latter sequence can be modified much more easily to cover the function `handle_request_body_chunk` ④ (if it was not already). For exist-

[2] https://datatracker.ietf.org/doc/html/rfc7540

```
static void handle_request_body_chunk(                                    (4)
   ...
   if (stream->req.write_req.cb(stream->req.write_req.ctx, payload,
       is_end_stream) != 0) {
       ...
}
```

```
static int handle_data_frame(...h2o_http2_stream_t *stream...){           (3)
    ...
    if (stream->state != H2O_HTTP2_STREAM_STATE_RECV_BODY && ...) {
      ...
      return 0;
    }
    handle_request_body_chunk(...);
    ...
}
```

```
static int handle_incoming_request(...h2o_http2_stream_t *stream...){    (2)
    ...
    h2o_http2_stream_set_state(conn, stream,
                               H2O_HTTP2_STREAM_STATE_RECV_BODY);
    ...
}
```

```
static int handle_headers_frame(...h2o_http2_frame_t *frame...)          (1)
{
    ...
    if ((frame->flags & H2O_HTTP2_FRAME_FLAG_END_HEADERS) != 0) {
        /* request headers are complete, handle it */
        return handle_incoming_request(conn, stream, ...);
    }
    ...
    return 0;
}
```

Figure 3: H2O handlers for header and data frames. A handler processes inputs as they arrive. Depending on the sequence in which header and data frames arrive, the value of the state variable is set and checked. Certain code is reachable only if the state variable carries a certain value.

ing coverage-guided fuzzers, there would be no distinction between these two scenarios in terms of coverage[3].

> By adding generated message sequences to the corpus that exercise a new sequence of states (i.e., add a new node to the state transition tree), we can capture both scenarios.

3 Methodology

Our main objective is to capture the protocol state space without any knowledge of the protocol. Our key observation is that the current protocol states are often explicitly represented using so-called state variables.

3.1 Offline State Variable Identification

Our first step is to automatically identify state variables in the program code. If the reader is tasked to implement a stateful protocol that contains explicitly named protocol states and a well-specified protocol logic that proceeds based on the current protocol state, how would the reader keep track of the current protocol state in his implementation? We examined the top-50 most widely used open-source implementations (cf. Section A.4)[4] of stateful protocols as indicated in a search on Shodan[5] or by Github stars. We found that all of them use named constants assigned to special state variables to keep track of the current state. Of those, 44 use `enum` type while 6 use `#define` statements to define the named constants. Unlike the state-of-the-art [20], which requires manual effort to write protocol-specific drivers that extract state information from the server responses, our state variable identification is entirely automatic.

An example of a state variable is given in Section 2. Using state variables, developers of protocol implementations can maintain a direct mapping between protocol states given in the protocol specification (cf. Figure 2) and the protocol states in the implementation (cf. Section 2.2). State transitions are implemented by assigning another named constant to the state variable. Protocol logic that is based on the current state can be implemented as switch-statements or if-conditions where the current state variable value guards the corresponding state-based protocol logic (c.f. Figure 3).

To identify state variables, we look for all variables of enumerated type (`enum` in C/C++). An *enumerated type* is a list of named constants used in computer programming to map a set of names to numeric values. Variables of enumerated type can only be assigned constants from the specified list of named constants. In our case, this list represents all states that the state variable can represent (e.g., `IDLE` to `END_STREAM` in Section 2.2). Specifically, we use regular expressions to automatically extract all definitions of enumerated types and then use the definitions to return the list of all enum variables that have been assigned at least once.

As we can see in Table 1, not all variables of enumerated type need to be state variables. The second category is enumerated types that represent all possible response or error codes. The third category is enumerated types that represent all possible configuration options for a configuration variable. However, in practice, our heuristic remains very effective. Response and error code variables are already indirectly used for state identification by the state-of-the-art AFLNET. Configuration variables are usually assigned once and only when the server starts (i.e., before we start recording the state transitions for each message sequence), or at the beginning of the session. If we record configuration variables at the beginning of the sessions, they appear at the beginning of each state transition sequence and will never be identified as "rare" states (which we focus on in Section 4.2).

3.2 State Transition Tree Data Structure

To capture the protocol state transitions exercised by generated sequences of inputs, we monitor the sequences of values

[3] As both handlers are called in a loop, there does not exist a direct edge between both handlers that could be covered.

[4] We also confirmed our observation for all stateful protocol implementations in the ProFuzzBench protocol fuzzing benchmark [19].

[5] Shodan is a search engine for various types of servers connected to the internet: https://www.shodan.io/.

Enum. Type	Example(s)	Explanation
State variables	`stream->state`, `sender_state`	Explicit representation of the protocol states. These are the enum-type variables, the values of which we aim to record in the state transition tree. In H20, 22 of 43 enum-type variables are actual state variables.
Error or response codes	`error`, `status`	Response codes or error codes are explicitly defined in the protocol and provide quick information to the client about the processing of the last message. These enum-type variables correspond to the existing response-code-based state identification in AFLNET. In H2O, 2 of 43 enum-type variables are response or error codes.
Configuration variables	`priority`, `run_mode`	These variables represent the concrete choices from a set of configuration options. Those options are read either from a configuration file or the command line. In H2O, 19 of 43 enum-type variables are configuration variables.

Table 1: Kinds of variables with enumerated types and their impact on the state transition tree.

assigned to state variables. All observed sequences of values across all state variables in all threads are stored in the same state transition tree, a data structure that represents the entire state space that has been explored by the fuzzer. An example is shown in Figure 1.

State Transition Tree. A node in the state transition tree represents the value of a state variable during program execution. Each node has only one parent node and zero or more children nodes. If there is only one state variable, the parent and the children nodes of a specific node respectively represent the state variable's value before and after the creation of that node. If there are multiple state variables, a node's parent or children nodes can represent values of different state variables. Each edge from a node to its children represents a value change of any state variable, indicating a state transition. From a global perspective, the tree has only one root node which represents the initial state, and each path from the root node to a leaf node represents a unique state transition sequence during program execution on an input.

We use the motivating example from Section 2 to explain the state transition tree (STT). Figure 1 shows the STT for the `stream->state` state variable after getting five execution traces. The root node is the idle State (0). After getting valid requests in stipulated order, the following nodes are recorded: receiving request header State (1), request body State (2), sending response header State (4), response body State (5), and end of stream State (7). If any request has malformed data, the state directly transitions to the end of stream State (7) as shown in the left three branches. Another situation is that the request body is empty. In this situation, it will go through the right branch, bypassing the receiving request body State (2).

Figure 4 shows a compact representation of the automatically constructed STT as a directed graph. Visual comparison to the official state machine in Figure 2 reveals a striking similarity with the specified protocol. Specifically, to construct the directed graph, we merge all nodes with the same values of enumeration variables, such as merging all leaf nodes which have the same State (7). Like the official state machine, our extracted state machine also starts at the idle state. After receiving a valid header frame, our extracted state machine transitions to the receiving request header State (1) while the official state machine transitions to the open state. Next, when traversing States (2)(3)(4)(5), the official state machine is still in the same open state. Finally, when the connection is ready

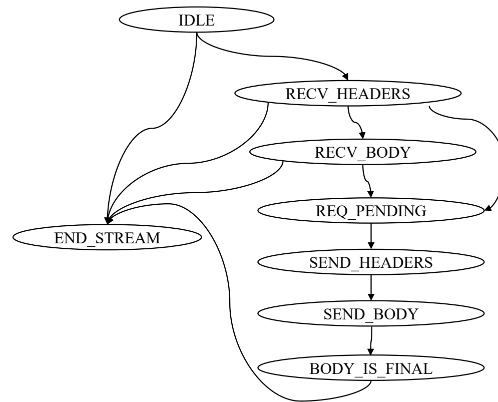

Figure 4: *The state machine extracted from the State Transition Tree of H2O (node name prefixes omitted).*

to close, our extracted state machine transitions to State (6) and State (7) which corresponds to the half-closed (local) and closed states in the official state machine, respectively. Therefore, we not only accurately recover parts of the official state machine, using a simple but highly effective approach, but we also get a more fine-grained state machine as it is actually implemented. Together with our greybox fuzzing algorithm, we aim to gradually recover more and more states on the fly.

3.3 State Transition Tree Construction

To construct the state transition tree (STT), we automatically inject a runtime into the program binary during compilation. The runtime gradually constructs the STT along with program executions on the inputs generated during fuzzing campaign.

Compile-time instrumentation. We developed a Python script that injects a call to our runtime before every assignment of a named constant to a state variable in the protocol implementation. Given the list of state variables extracted during our offline state variable identification, we use regular expressions to find all instructions in the code where the state variables are assigned new values. The injected function calls pass the state variable names and the value of the named constant to our runtime. To make the instrumentation more flexible and accurate, we also provide an option to allow users to block some variables extracted from offline state variable identification. We did not choose LLVM to instrument the pro-

tocol implementation because enum-values will be replaced by integer-type constant values in the LLVM intermediate language (LLVM-IR).

Runtime derivation of State Transition Tree. Across all executions of a protocol implementation during a fuzzing campaign, the runtime uses the information passed by the injected calls, to gradually construct the STT. The STT data structure is implemented in the runtime component with a pointer that points to the last visited node (initially root). Each node in the STT is distinguished by the variable name and its corresponding value, passed from the instrumented calls to the runtime. Once called, the runtime checks whether a matching node already exists among the child nodes of the last visited node. If not found, a new child node is created for it. Then the last-visited-node pointer is set to point to the already existing or the newly created child node, whichever applies. After execution of the program on each input, the last-visited-node pointer is reset to the root node waiting for the next execution. To support multi-threading, which is common for protocol implementations, we leverage a mutex to protect the code for updating the state transition tree from concurrency issues. To avoid that the STT grows to an unreasonable size, we provide an option to limit the maximum number of state repetitions along with a single execution. Whenever a state is repeated more often than allowed (along a path in the STT), subsequent states are ignored (and that particular path is truncated). This allows us to efficiently manage the state transitions as a tree, rather than a graph, during the fuzzing campaign. In our experiments,we chose the same threshold for all subjects, generally so that the number of nodes does not exceed 10k nodes.

3.4 Handling Implicit States

In addition to the explicit states that are captured using state variables, like for any software system, there exist implicit states: databases and file systems are changed; some memory is allocated but never freed. While explicit states are reset whenever the connection is closed after a message exchange, implicit states aggregate over time and survive various connection resets.[6] The STT can only keep track of explicit states but not of such implicit states. To explore the implicit state space, we let the implicit states aggregate without reset. This allows us to find other types of stateful bugs.

Our in-memory stateful greybox fuzzer lives in the same process as the protocol implementation. Hence, the protocol implementation under test does not need to be restarted for every message exchange. This gives a substantial performance boost but also allows us to aggregate implicit states across message exchanges. To facilitate reproducing this type of bug, we enable an option to save the list of all the inputs on which the program has been executed. After getting a crash report from the fuzzer, we gradually minimize the list until a minimal list triggers this crash. This minimal list can be used for further manual analysis.

[6] A connection includes multiple requests and responses.

4 Stateful Greybox Fuzzing Algorithm

To efficiently explore the state space, we propose to use the State Transition Tree (STT), described in section (3.2), as a guide for the stateful greybox fuzzer. SGFuzz incrementally constructs the STT at runtime and employs it as follows:

1. Generated inputs that increase coverage of the STT are added to the seed corpus.
2. Within the seed corpus, SGFuzz prioritizes the seeds that have a greater potential to increase the coverage of the STT.
3. Within the chosen seed, SGFuzz prioritizes bytes associated with newly added STT nodes which have more potential to increase the coverage of the STT.

Algorithm 1 Stateful Greybox Fuzzing

```
Input: Seed Corpus T
  Crashing inputs T_x = ∅
  for each t ∈ T do
    E_t = 1        // initial energy
  end for
  State transition tree STT = ∅
  repeat
    t = choose_next(T, E)
    t' = mutate(t, STT)
    STT = execute(t', STT)
    if t' crashes then
      return t'
    else if is_interesting(t', STT) then
      add t' to T
      t' = identify_bytes(t', STT)
      for each t_i ∈ T do
        E_{t_i} = assign_energy(t_i, STT)
      end for
    end if
  until timeout reached or abort-signal
Output: Crashing Input t_x
```

Algorithm 1 shows the general workflow of our fuzzing algorithm. We implemented this algorithm into LibFuzzer and call our fuzzer SGFuzz (Stateful Greybox Fuzzer). Starting with an initial seed corpus T and until a timeout is reached or the user aborts the campaign, the following steps are repeated. Firstly, SGFuzz samples t according to the energy distribution E which represents the likelihood of each seed in T to be chosen (*choose_next*). The chosen seed t is then mutated (*mutate*) to generate a new input t' (Section 4.3). Then we execute the protocol implementation on t' and update the

STT (Section 3.3). If t' causes a crash SGFUZZ returns the crashing input, else if t' exercises new code branch or new STT nodes (*is_interesting*), t' is added to the corpus T, and otherwise t' is discarded (Section 4.1). Whenever new STT nodes added, SGFUZZ identifies the bytes in input t' (*identify_bytes*) that contributed to the coverage of the new STT nodes (Section 4.3) and updates the energy E_{t_i} of all seeds $t_i \in T$ in the current corpus (*assign_energy*) (Section 4.2). The portions of LIBFUZZER we changed are highlighted in red in Algorithm 1.

4.1 State Coverage Feedback

Our first heuristic is to add generated inputs to the seed corpus that exercise new nodes in the STT (*is_interesting* in Alg. 1) as the augmentation of the original branch coverage feedback. The rationale is to maximize the coverage of the STT data structure to maximize the coverage of the state space. Each new node in the STT represents a new value in the current sequence of state variable values indicating a new state under the current sequence of state transitions. Like in code-coverage-guided greybox fuzzing, inputs that increase coverage are added to the seed corpus, as well.

4.2 Energy Schedule

Our second heuristic is to assign more energy to seeds that have a greater potential to lead to new states (*assign_energy* in Alg. 1). Compared to existing works[5, 31] that prioritize inputs that execute rarely executed code blocks, our algorithm additionally assigns more energy to the seeds that execute the valid state transitions which correspond to the valid protocol behaviors.

Firstly, we assign more energy to the seeds that traverse the rarely visited nodes in the STT. We set up an attribute called *hit count* for each node in the STT to record the number of inputs that traverse that node. We expect that STT nodes with low *hit count* have more unexplored states in their "neighborhood". Given the STT, we consider a node $n \in STT$ to be *rare* if it has a below-average *hit count*.

$$rare(n) = \begin{cases} 1 & \text{if } hits(n) < \frac{\sum_{n' \in STT} hits(n')}{|STT|} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $hits(n)$ returns the *hit count* of node n.

Let $path(t)$ represent the set of nodes that are traversed by input t in STT. Given an input t in corpus T, our power schedule assigns energy $e_1(t)$ of t as the proportion of rare nodes that are visited during program execution on t added to the original energy of t:

$$e_1(t) = e'(t) + e'(t)\frac{\sum_{n \in path(t)} rare(n)}{|path(t)|} \tag{2}$$

where $e'(t)$ is the existing power schedule.

Secondly, we assign more energy to the seeds whose offspring are more likely to take a different path through the STT. By Equation 2, all nodes in STT have similar opportunities to be executed. However, not all state transitions are equally important, and valid state transitions are more attractive. Our intuition is that the valid state transitions, which correspond to the expected protocol behaviors, are usually easy to be mutated to other invalid state transitions, which represent the error handling of the protocol. The change on any byte may incur error handling logic. This is also a challenge of previous stateful greybox fuzzing techniques. Thus, we identify this type of seed and assign more energy to it. Given an input t and a set of inputs t_m that are mutated from t, we calculate the inverse proportion of its offspring that traverse the same paths as $path(t)$. We extend our power schedule $e_2(t)$ as follows:

$$e_2(t) = e_1(t) \cdot \frac{|t_m|}{\sum_{c \in t_m}(path(c) == path(t))} \tag{3}$$

Finally, to avoid assigning too much energy, we limit our power schedule $e(t)$ to no more than ten times (empirically works best than other numbers) the original energy $e'(t)$. Given the seed input $t \in T$ in seed corpus T, our power schedule assigns the new energy $e(t)$ to t.

$$e(t) = \min(e_2(t), 10 \cdot e'(t)) \tag{4}$$

4.3 Fuzzing Individual Bytes

We define a simple heuristic to fuzz parts of the seed first that might have more impact on increasing state (STT) coverage. When an input seed is chosen to be mutated, its individual bytes to be mutated are selected based on the bytes' priorities. Bytes have higher priority if they are deemed to have more impact on the STT's coverage (*mutate* and *identify_bytes* in Alg. 1). Recent works on input probing [3, 11, 29] establish the information flow from particular input bytes to a given sink in a very lightweight manner. We leverage these ideas to explore state space. If program execution on a mutated input triggers new nodes in the STT and hence, that input is causing new unexplored nodes to be visited for the first time, it may have a great potential to increase the coverage of the state space. The idea is to identify the individual bytes in such an input whose mutation has contributed to the increase in the STT's coverage. Once identified, those bytes will be assigned higher priority to be chosen for mutation. It makes fuzzing more efficient by enabling earlier exploration of the "neighbor" states around the newly created nodes that have not been visited, yet.

Given an input t and a mutated input t' derived from t, if the program execution on input t' creates new nodes in STT, t' is saved by SGFUZZ. Then SGFUZZ computes the set difference of bytes $diff(t,t')$ between t and t' and records the set of newly added STT nodes N exercised by t'. We annotate the mutation range as $R_{t'} = diff(t,t')$ if $|N| > 0$, or set $R_{t'}$ to the

Subject	Protocol	Fuzz Driver	Commit	Size	Share
H2O	HTTP	h2o-fuzzer-http2	`1e7344`	337k LoC	12k IPs
MbedTLS	SSL/TLS	dtlsserver	`e483a7`	138k LoC	8m IPs
Curl	Several	curl_fuzzer	`aab3a7`	202k LoC	-
Gstreamer	Custom	gst-discoverer	`44bdad`	235k LoC	15k IPs
OpenSSL	SSL/TLS	netdriver	`1e08f3`	673k LoC	>10m IPs
Live555	RTSP	netdriver	21.Aug'08	17k LoC	12k IPs
OwnTone	DAAP	netdriver	`774d7c`	37k LoC	10k IPs
DCMTK	DICOM	netdriver	`24ebf4`	38k LoC	3k IPs

Table 2: Protocol implementations used for comparison.

entire seed t' ($R_{t'} = \langle t' \rangle$), otherwise (*identify_bytes* in Alg. 1). Later, when t' is chosen to be mutated for fuzzing (*mutate* in Alg. 1), we only mutate the bytes under the range $R_{t'}$. If it does not bring any interesting behavior (*is_interesting* in Alg. 1), we gradually enlarge $R_{t'}$ until it spans the entire seed t'. This helps us build an efficient fuzz campaign.

5 Experimental Setup

5.1 Research Questions

Our main hypothesis is that stateful greybox fuzzing allows for more effective fuzzing of stateful programs, such as protocol implementations. We investigated this hypothesis along with the following research questions.

RQ.1 (State Transition Coverage) How many more different sequences of state transitions does SGFUZZ exercise? By mapping out the state space, SGFUZZ aims to penetrate deeper into the program's state space. The number of different sequences of states exercised by a fuzzer indicates how deep the fuzzer reaches into the state space.

RQ.2 (Branch Coverage) How much more branch coverage does SGFUZZ achieve? We cannot find bugs in code that is not covered.

RQ.3 (State Identification Effectiveness) Identifying state variables, what is the false positive / false negative rate? How does it affect the mapping of the state space (STT)?

RQ.4 (Bug Finding Effectiveness) How much faster does SGFUZZ expose stateful bugs in our benchmark?

RQ.5 (New Bugs) How many previously unknown bugs can SGFUZZ find in widely used stateful programs? How many are stateful?

5.2 Subject Programs Studied

Benchmarks. We constructed our set of subjects from two mature benchmarks. PROFUZZBENCH [19] is a benchmark for stateful fuzzing of network protocols and facilitates comparison to AFLNET [20] (i.e., the current state-of-the-art stateful fuzzer). PROFUZZBENCH includes a suite of representative open-source network servers for popular protocols. OSS-FUZZ[7] is an open-source continuous fuzzing framework that builds more than 500 open source projects for fuzzing. FUZZBENCH [18] is an open-source fuzzer evaluation infrastructure that automatically conducts fuzzing experiments and produces coverage graphs. FUZZBENCH supports integrating benchmarks from the OSS-FUZZ.

Selection Criteria. We selected 8 real-world target programs as subjects for our experiments as shown in Table 2. We opted for the programs that 1) work on distinct and well-known protocols aiming to provide diverse state machines, 2) are widely used (as indicated by a search on the Shodan), 3) are used in security-critical industries, such as medical and finance, in which any bug can have serious consequences. At the time of writing, they represent the most recent version (cf. *commit* column) of eight programs implementing more than 30 different well-known protocols (curl includes multiple protocol implementations). These programs are widely used across many machines on the internet (cf. #IPs in the `Share` column). PROFUZZBENCH needs socket communication and we use *netdriver*[8] to enable it for in-memory fuzzers. Unfortunately, *netdriver* does not support the UDP protocol or the fork system call because a fork will spawn a new process that prevents fuzzers from measuring branch coverage. We selected all programs in PROFUZZBENCH which are compatible with in-memory fuzzers.

Selected Subjects. `H2O` is a popular web server; `MbedTLS` and `OpenSSL` are security-critical encryption libraries; `Curl`[9] is a utility to communicate via multiple protocols and implements many state machines; `Gstreamer` implements a real-time streaming protocol;[10] `Live555` and `OwnTone` are libraries for streaming media; and `DCMTK` is used for the medical data exchange between hospitals (where bugs can be literally fatal). Three quarter of those protocol implementations are running on more than 10 thousand different machines (IPs), according to a recent query of the Shodan search engine.

5.3 Fuzzers

We implemented our stateful greybox fuzzing approach into LIBFUZZER (the most recent version at the time of our experiment) and call it Stateful Greybox Fuzzer (SGFUZZ). In our experiments, SGFUZZ is compared against LIBFUZZER as the baseline and AFLNET and IJON as the state-of-the-art.

Stateful greybox fuzzer (SGFUZZ). To implement our algorithm into LIBFUZZER, we extended the *Fuzzer::RunOne* function with the heuristic described in Section 4.1, changed

[7] https://google.github.io/oss-fuzz/

[8] https://github.com/google/honggfuzz/tree/master/libhfnetdriver

[9] `Curl` implements 26 client-side protocols which Shodan cannot identify.

[10] However, the OSS-Fuzz fuzz driver targets the custom protocol which retrieves information about a remote media file.

the *InputCorpus::UpdateCorpusDistribution* function to implement the heuristics discussed in Section 4.2, and added a filter to the *MutationDispatcher::Mutate* function for the heuristics discussed in Section 4.3. As for the construction of the STT discussed in Section 3.3, we implemented a standalone python script for the instrumentation component which is static and does not rely on any compiler infrastructure. The runtime component is embedded in the fuzzer.

Baseline. LIBFUZZER is in-memory, coverage-guided, evolutionary fuzzer and serves as the baseline fuzzer we used to implement SGFUZZ.

State-of-the-art. AFLNET is the current state-of-the-art stateful fuzzer for protocol implementations that uses the concept of state-feedback. IJON is another state-of-the-art stateful fuzzer which needs manual annotation for states. We used both to evaluate if SGFUZZ gains an advantage over the state-of-the-art stateful fuzzer. Because IJON does not support FUZZBENCH or PROFUZZBENCH and to facilitate a fair comparison to our baseline and our implementation, we re-implemented IJON in LIBFUZZER and ported its annotation function `ijon_push_state`[11] for IJON-based state-feedback. To maximize the benefit for IJON, we assume a human annotator would annotate the same state variables as identified by SGFUZZ. AFLNET does not support *FuzzBench*.

Benchmark Integration. All fuzzers were integrated into FUZZBENCH or PROFUZZBENCH using equivalent setups and build processes. All fuzzers are provided the same setup, environment, seeds, and resources. For all fuzzers, AddressSanitizer was enabled to detect memory corruption bugs. While PROFUZZBENCH uses the `gcov` tool, FUZZBENCH uses the `llvm-cov` tool to measure the number of branches covered in a fuzzing campaign in 15-minute intervals.

5.4 Experimental Infrastructure

Computational Resources. After the integration of SGFUZZ and our benchmarks into FuzzBench, we received assistance from the FuzzBench team to run the branch and state transition coverage experiments on the Google Cloud Service [18]. For the other four subjects in the ProFuzzBench, we ran the branch and state transition coverage experiments on our local machine with an Intel Xeon Gold 6258R CPU that has 128 logical cores running at 2.70GHz. The machine runs Ubuntu 20.04.2 LTS and has access to 128GB of main memory.

Experiment configurations. As for the *initial seed corpus*, we selected one valid input for each subject as the initial seed for RQ1, RQ2, and RQ3, and selected all seeds provided FuzzBench or ProFuzzBench as the initial seed for RQ4 and RQ5. In order to *mitigate the impact of randomness*, we repeated every 23 hours fuzzing campaign 20 times. The 23 hours timeout is the default for the FuzzBench team. To make the results consistent, we use the same time budget.

[11] https://github.com/RUB-SysSec/ijon/blob/56ebfe34709dd93f5da7871624ce6eadacc3ae4c/llvm_mode/afl-llvm-rt.o.c#L75

Subject	AFLNET	LIBFUZZER	IJON	SGFUZZ	$\hat{A}_{12}$	U	Factor
H2O	- -	70.80 (3.61%)	91.85 (4.68%)	1849.30 (94.26%)	1.00	<0.01	26.1
MbedTLS	- -	22.80 (39.31%)	32.45 (55.95%)	50.80 (87.59%)	1.00	<0.01	2.2
Curl	- -	150.25 (0.98%)	375.75 (2.45%)	14630.80 (95.55%)	1.00	<0.01	97.3
Gstreamer	- -	49.40 (1.08%)	134.20 (2.94%)	4067.30 (88.96%)	1.00	<0.01	82.3
OpenSSL	13.25 (33.97%)	23.95 (61.41%)	29.60 (75.90%)	33.10 (84.87%)	0.81	<0.01	1.4
Live555	138.27 (11.42%)	184.15 (15.21%)	405.3 (33.75%)	1162.30 (95.98%)	1.00	<0.01	6.3
OwnTone	1.00 (0.10%)	46.40 (4.82%)	426.00 (44.28%)	930.15 (96.69%)	0.71	0.03	20.0
DCMTK	68.10 (0.92%)	189.25 (2.55%)	267.50 (3.93%)	6737.05 (90.87%)	1.00	<0.01	35.6
							Avg: 33.9x

Table 3: Average state transition coverage for 20 campaigns of 23 hours. The percentage shows this number as a proportion of distinct transition sequences exercised by any run of any fuzzer. Vargha-Delaney ($\hat{A}_{12}$) measures the magnitude of the difference in transition coverage at 23 hours between SGFUZZ and LIBFUZZER (effect size) while Wilcoxon rank-sum test (U) measures the degree to which these results may be explained by randomness (statistical significance). *Factor* shows the factor improvement compared to LIBFUZZER.

6 Evaluation Results

RQ.1 State Transition Coverage

For all fuzzers, subjects, and campaigns, we measured state transition coverage as the number of paths in the STT which is constructed across the execution of the subject on *all* seeds generated throughout the campaign and from the initial corpus. Each path represents a unique sequence of state transitions during the execution of the subject on an input. If a fuzzer exercises more such succinct sequences, it penetrates deeper into the protocol's state space.

Presentation. Table 3 shows the average number of state transition sequences for 20 campaigns of 23 hours experiments. To assess the "completeness" of the state space coverage, we counted the number of distinct sequences for any run of any fuzzer and consider this as 100% for all fuzzers and the average percentage of state transitions exercised by each fuzzer is shown in the brackets. *Factor* represents the factor improvement of SGFUZZ to LIBFUZZER in terms of average #state sequences. Vargha Delaney $\hat{A}_{12}$ measures the *effect size* and gives the probability that a random fuzzing campaign of SGFUZZ achieves more state transition coverage than a random fuzzing campaign of LIBFUZZER (i.e., $\hat{A}_{12} > 0.5$ means SGFUZZ is better). The Wilcoxon rank sum test U is a non-parametric *statistical hypothesis test* to assess whether the coverage differs across both fuzzers. We reject

the null hypothesis if $U < 0.05$, i.e., SGFUZZ outperforms LIBFUZZER with statistical significance. AFLNET does not support *FuzzBench*. We annotated the same states as in SGFUZZ to IJON which needs manual state annotations.

Results. For all subjects, SGFUZZ covers substantially more state transitions than AFLNET, LIBFUZZER, and IJON. Compared to baseline LIBFUZZER, SGFUZZ exercises 34x more state transitions on average (20x median) across all eight subjects, owing to the STT guided fuzzing approach. Especially for *Curl*, *Gstreamer*, and *DCMTK*, SGFUZZ increases the state transition coverage significantly. Those have much more state variables than others. SGFUZZ even executes 97x more state transitions than LIBFUZZER on *Curl* and 930x more state transitions than AFLNET on *OwnTone*. An extended discussion about about the contribution of the individual heuristics in SGFUZZ can be found in Section A.1.

AFLNET vastly underperforms. We believe the reasons are 1) AFLNET identifies state by a different method (using the response code) and 2) AFLNET restarts the tested program after each generated message sequence which incurs a big performance loss. In contrast, other fuzzers leverage the same in-memory architecture for high throughput.

IJON exercises 2.75x more state transitions than LIBFUZZER on average as it adds state feedback, but SGFUZZ still exercises 15.29x more state transitions than IJON. As a proportion of all sequences exercised in any run of any fuzzer, SGFUZZ covers 92% of state transitions, on average (99% on the sum of 20 campaigns). This shows that SGFUZZ can cover almost all state transitions covered by other fuzzers and more. 1% sequences are executed by LIBFUZZER only because of the randomness of fuzzing.

LIBFUZZER, as the only stateless fuzzer, performs well only on OpenSSL where the total number of distinct state transition sequence is relatively low (39). We found that LIBFUZZER explores states mostly near the initial states or end states (cf. Section A.3).

For subjects *MbedTLS* and *OpenSSL*, we found that SGFUZZ has smaller state transition increase than for the other subjects. For those subjects, the number of distinct sequences explored by any fuzzer is relatively low. For instance, in *OpenSSL*, *st->hand_state* variable is an enum-type variable used for tracking handshake states of TLS protocol. From state *TLS_ST_SR_CLNT_HELLO*[12], it needs at least four more conditions to reach state *DTLS_ST_SW_HELLO_VERIFY_REQUEST*, and at least two more conditions to reach state *TLS_ST_OK*. Some of the conditions need a specific external state. For instance, *SSL_IS_DTLS(s)* is true when the traffic comes via UDP protocol rather than TCP.

[12] https://github.com/openssl/openssl/blob/c74188e/ssl/statem/statem_srvr.c#L585

On average, SGFUZZ covers 33x more sequences of state transitions than LIBFUZZER, 15x more than IJON, and 260x more than AFLNET. By tracking the state space via the STT, SGFUZZ is able to cover more of the state space.

RQ.2 Branch Coverage

Presentation. Table 4 shows the branch coverage result after 23 hours. *Improvement* represents the percentage improvement of SGFUZZ to LIBFUZZER based on the average branch coverage. *time-to-coverage* measures the time it takes AFLNET, IJON, and SGFUZZ, respectively to achieve the same coverage that LIBFUZZER achieves in 23 hours. *Factor* represents the factor improvement of SGFUZZ to LIBFUZZER based on time-to-coverage. $\hat{A}_{12}$ represents the Vargha Delaney effect size while U represents Wilcoxon signed rank statistical significance.

Results. Although AFLNET also aims to explore the state space of the program, there are substantial inefficiencies that prevent AFLNET from achieving good coverage on most subjects. IJON performs slightly worse than LIBFUZZER because it introduces performance cost for state tracing. However, with our algorithm, in general, SGFUZZ overcomes the performance cost caused by the STT and outperforms both LIBFUZZER and IJON to reach more state-dependent code. The seeds are valid but only execute few state transition sequences. LIBFUZZER seems to be stuck in "shallow" states, and IJON explores more state space with state tracing, while SGFUZZ explores much more states that reach much "deeper" by both state tracing and heuristic algorithms. More discussions about our algorithm's contribution to the branch coverage can be found in Section A.1 and the growth trend of branch coverage can be found in Section A.2.

One exception is the *Gstreamer*. We explain the reason with the huge state space in *Gstreamer* which corresponds to a small code space. The enum-type variable `best_probability`, which ranges from 1 to 100, represents the certainty about the media type of that stream. When *Gstreamer* receives a stream, it gradually updates the value of `best_probability` as more information becomes available. This state variable has 100 states and is updated frequently to seemingly random values, so SGFUZZ generates much more inputs to explore the state space despite not achieving much more branch coverage. To confirm our hypothesis, we removed `best_probability`, repeated this experiment, and saw a good improvement in the coverage results for SGFUZZ. LIBFUZZER would take 23 hours to achieve about the same coverage as SGFUZZ achieves in 12 hours.

Table 4 (left) shows that, on average, SGFUZZ achieves 2.20% more branch coverage than LIBFUZZER, 4.41% more than IJON, and 38.73% more than AFLNET. Table 4 (right) shows that SGFUZZ achieves the same branch coverage 2.3x faster than LIBFUZZER in the 23 hours, on average. For state-

Subject	Branch Coverage							Time-to-Coverage						
	AFLNET	LIBFUZZER	IJON	SGFUZZ	$\hat{A}_{12}$	U	Improvement	AFLNET	LIBFUZZER	IJON	SGFUZZ	$\hat{A}_{12}$	U	Factor
H2O	-	2879.20	2820.25	3050.68	0.84	<0.01	5.96%	-	23.00h	14.04h	6.69h	0.98	<0.01	3.4
MbedTLS	-	11929.60	11885.65	12057.68	0.58	0.43	1.07%	-	23.00h	>23.00h	17.86h	0.75	<0.01	1.3
Curl	-	19262.16	16892.55	19942.39	0.71	0.03	3.53%	-	23.00h	21.89h	12.23h	0.95	<0.01	1.9
Gstreamer	-	63280.06	60993.58	61698.56	0.11	<0.01	-2.50%	-	23.00h	>23.00h	>23.00h	-	-	-
OpenSSL	11342.00	12463.14	12456.00	12610.50	0.72	0.02	1.18%	>23.00h	23.00h	>23.00h	10.08h	1	<0.01	2.3
Live555	2179.87	2244.63	2214.38	2278.10	0.73	0.02	1.49%	>23.00h	23.00h	>23.00h	15.86h	0.95	<0.01	1.5
OwnTone	991.00	2184.64	2299.20	2312.60	0.65	0.15	5.86%	>23.00h	23.00h	6.14h	5.04h	1	<0.01	4.6
DCMTK	5763.00	6042.15	5997.05	6100.50	0.77	<0.01	0.97%	>23.00h	23.00h	>23.00h	14.40h	0.93	<0.01	1.6
							Avg: 2.20%							**Avg:** 2.3x

Table 4: Average branch coverage (left) and average time to coverage (right) for 20 campaigns of 23 hours. The time to coverage measures the time it takes AFLNET, IJON, and SGFUZZ to achieve the same coverage that LIBFUZZER achieves in 23 hours (0.25h accuracy). Vargha-Delaney ($\hat{A}_{12}$) measures the magnitude of the difference (effect size) while Wilcoxon rank-sum test (U) measures the degree to which these results may be explained by randomness (statistical significance). *Factor* shows the factor improvement compared to LIBFUZZER.

dependent code, this is either because SGFUZZ takes much shorter time than LIBFUZZER to reach the same coverage, (Time-to-coverage), or SGFUZZ reaches parts of the code that LIBFUZZER has been unable to reach (Branch coverage and RQ2 demonstrated in Figure 3). Looking at RQ1 and RQ2 together, there is a greater increase in transition coverage than in branch coverage. This further supports our hypothesis that *conventional branch coverage feedback is insufficient to measure the state space coverage of programs.*

We recall our example at Section 2 to explain the branch coverage gap between SGFUZZ and LIBFUZZER. The function sequences ①②③ and ③①② would result in the same branch coverage, but they are different regarding the value of the state variable (`stream->state`). This difference is captured by the STT in SGFUZZ and hence, both two corresponding seeds are saved for further exploration, while LIBFUZZER is unable to distinguish between these function call sequences and hence only the first input seed that result in one of these function calls will be saved. Consequently, as the function ④ is easier to be explored from mutating the seed resulting to ①②③ than from the other seed, LIBFUZZER has a high chance of missing the corresponding precious seed and becoming unable to cover the function ④. In our experiments, the function ④ is always explored by SGFUZZ faster than LIBFUZZER, or is unexplored for LIBFUZZER in 23 hours. There are more state-dependent code that produces the branch coverage gap.

> On average, SGFUZZ achieves 2.20% more branch coverage than LIBFUZZER, 4.41% more than IJON, and 38.73% more than AFLNET. SGFUZZ achieves the same branch coverage about 2x faster than LIBFUZZER.

RQ.3 State Identification Effectiveness

To evaluate how well our heuristic is able to identify true state variables, we manually examined the valid identified variables of enumerated type for all subjects as well as the correspond-

Subject	Enum			STT			RFC
	All	State	Percentage	All	State	Percentage	
H2O	43	22	51.16%	6418	6417	99.98%	✓
MbedTLS	36	33	91.67%	167	167	100.00%	✓
Curl	81	44	54.32%	35690	35629	99.83%	✓
Gstreamer	218	151	69.27%	11240	11224	99.86%	NA
OpenSSL	64	40	62.50%	817	789	96.57%	✓
Live555	11	10	90.91%	17446	17446	100.00%	✗
OwnTone	51	37	72.55%	3671	3671	100.00%	NA
DCMTK	145	80	55.17%	27178	27109	99.75%	NA
		Avg:	68.44%		**Avg:**	99.50%	

Table 5: [Left] The number of identified (All) and actual state variables (State) among enum-type variables. [Right] The number of nodes (All) and state-related nodes (State) in the STT constructed in 23 hours.

ing Request for Comments (RFC) documents, which contain technical specifications and organizational notes for the Internet. To evaluate *false positives*, we classified the identified variables into 1) `configuration variables` if their values are got from static sources (configuration files, command-line parameters), 2) `state variables` if their values are got from inputs and affect the program execution by the `if`- or `switch`-statements, 3) `error or response codes` if their values are assigned as the return value of functions. To evaluate the *impact on state space exploration*, we measured the percentage of nodes in the STT that are actually state variables. To evaluate *false negatives*, we checked if the STT covers the protocol states as discussed in the corresponding RFCs. Specifically, we checked a) if an RFC exists, b) if the specified state machine is represented as enumeration type in the protocol implementation, and c) if STT covers these variables.

Presentation. Table 5 shows the results of the state identification analysis. *All* represents the number of enum-type variables or of nodes in STT. *State* represents the number of state variables in enum-type variables or state nodes in STT. *RFC* represents that if the protocol state machines stipulated in the Request for Comments(RFC) documents are covered

by the STT. *NA* represents *Not Applicable*.

Results. On average, 68.44% of variables that were identified using our enum-based heuristic are actually state variables. However, the 32% false positive rate has a low impact on the State Transition Tree (STT) construction. On average, 99.5% of nodes in the STT are state variable values. The value of state variables change several times during the execution which is reflected in the growing STT. The remaining 0.5% of nodes are related to non-state variables. For instance, configuration variables are usually assigned once and often appear at the beginning of an execution (e.g., when the protocol exchange is configured). Error or response codes are usually written to log files or returned back to users *once* per execution and without any change in value. This means non-state-variables rarely appear in the STT and thus do not affect our algorithm in Section 4 or the achieved branch coverage.

> While an average of 32% of identified variables are not actually state variables, an average 99.5% of nodes in the STT constructed in 23 hours are referring to values of actual state variables. This is explained by the STT tracking *changes* in variable values.

We also investigated whether protocol states specified in the Request for Comments (RFC), if any, are reflected as enumeration-type state variables, and whether they appear in the STT. In other words, we investigated false negatives. *There is one false negative among our eight subjects.* As we can see in the last column of Table 5, three of the eight protocol implementations are proprietary protocols and no RFC exists, and the implementation (using state variables) already provides the ground-truth.[13] Four of the remaining five protocols implement the specified states as enumeration-type state variables.[14]. For the Live555 implementation of the RTSP protocol, the corresponding RFC-2326 does specify the RTSP protocol state machine but it is implicitly represented by state variables related to data transport or parsing, which imply the protocol state machine. For instance, the enum-type variable *fCurrentParseState* maintains the states of the media file parser, which must be the initial state under the *READY* state of RTSP protocol and must not be the initial states under the *PLAY* state of the RTSP protocol. However, as there is no explicit state variable, we count this as a false positive.

To evaluate state coverage regarding true state variables, we recorded the number of distinct values of true state variables in our experiments, as shown in Table 6. On average, SGFuzz covers 88.98% more states than LibFuzzer in 23 hours. For *Gstreamer*, which has the best improvement, there is a huge state space in which some states are easier to be explored, such as `best_probability` variable (explained in RQ2). For *H2O*, which has least improvement, we found that the RFC-related variable `stream->state` only has eight possible values and all eight values are covered by all the fuzzers in 23 hours, and hence, there has been less room for state coverage improvement. In general, exercising new states is more challenging because specific inputs or external environment are usually needed, but SGFuzz still outperforms other fuzzers in state coverage.

Subject	AFLNet	LibFuzzer	IJON	SGFuzz	Improvement
H2O	-	12	12	12	+0.00%
MbedTLS	-	14	17	21	+50.00%
Curl	-	55	67	75	+36.36%
Gstreamer	-	23	31	113	+391.30%
OpenSSL	40	40	44	47	+17.50%
Live555	15	15	17	19	+26.67%
OwnTone	10	18	23	27	+50.00%
DCMTK	6	5	7	12	+140.00%
				Avg:	**+88.98%**

Table 6: The average number of distinct state variable values observed in campaigns of length 23 hours (for enumeration variables that we know for sure represent states).

We should also note that our automatic state identification is much more complete than a manual annotation, as in IJON [2]. For the `protocol-implementation-related example`[15] in the paper, the authors annotated the message type in TPM to represent state (Listing.D&Section.V.D). IJON can generate more "distinct sequences" of message types. SGFuzz would identify the state variables [16] of each TPM subsystem and track the state [17] of these subsystems. We examined the experiment data of IJON and found that the message sequences with the same message types may correspond to different subsystem states. For example, for the same message type sequence 32 and 34 (integers used in TPM to mark message type), the subsystem's state may be `SU_RESTART` or `SU_RESET`. It is because the subsystem's states are affected by previous execution results, not the message type. This shows how SGFuzz can automatically annotate state in a more fine-grained manner than human.

RQ.4 Bug Finding Effectiveness

Among known bugs from OSS-Fuzz and ProFuzzBench, we selected seven stateful bugs to evaluate if SGFuzz exposes stateful bugs faster, and three stateless bugs to evaluate if SGFuzz hinders the finding of stateless bugs. In ProFuzzBench, we selected the stateful bug CVE-2019-7314 in *Live555* which was found by AFLNet. In OSS-Fuzz, we used the bug tracker that contains fuzzer-generated bug reports to chose

[13] It is interesting to point out that these protocols cannot be tested by traditional specification-guided protocol testing tools.

[14] *H2O*(rfc7540), *MbedTLS* (rfc8446), *Curl* (rfc959, rfc3501, etc.), and *OpenSSL* (rfc8446)

[15] https://github.com/RUB-SysSec/ijon-data/blob/master/tpm_fuzzing/src/CommandDispatcher.c#L396

[16] https://github.com/RUB-SysSec/ijon-data/blob/master/tpm_fuzzing/src/CommandAudit.c#L100

[17] https://github.com/RUB-SysSec/ijon-data/blob/master/tpm_fuzzing/src/Global.h#L369-L374

Subject	Bug	AFLNET Avg	AFLNET Num	LIBFUZZER Avg	LIBFUZZER Num	IJON Avg	IJON Num	SGFUZZ Avg	SGFUZZ Num	Factor
Stateful:										
H2O	12096	-	-	4.17h	12	4.89h	18	2.20h	19	1.9
OpenSSL	4528	-	-	>23h	0	14.28h	6	7.09h	10	∞
Curl	16907	-	-	6.29h	19	4.92h	11	1.98h	19	3.2
Gstreamer	20912	-	-	0.07h	20	0.07h	20	0.03h	20	2.3
Live555	CVE	7.78h	20	0.05h	20	0.05h	20	0.05h	20	1.0
H2O	3303	-	-	>23h	0	>23h	0	>23h	0	-
Curl	22874	-	-	>23h	0	>23h	0	>23h	0	-
									Avg:	2.1x
Stateless:										
H2O	554	-	-	4.69h	20	4.48h	16	3.86h	20	1.2
Curl	17954	-	-	0.18h	20	0.21h	20	0.15h	20	1.2
OpenSSL	3122	-	-	>23h	0	>23h	0	>23h	0	-
									Avg:	1.2x

Table 7: Time to expose existing bugs: 20 campaigns, 23 hr.

the other seven bugs. Among *stateful bugs*, we take the bugs that require at least two state transitions to be exercised before they can be exposed. *Stateless bugs* are the bugs that can be executed without any state transition.

Presentation. Table 7 shows the average time to expose all bugs in 20 campaigns of 23 hours. *Num* is the number of campaigns in which the bugs were exposed. *Factor* represents the factor improvement of SGFUZZ to LIBFUZZER based on the average time-to-bug. *Bug* shows the bug identifier in the OSS-Fuzz issue tracker (for Live555, CVE-2019-7314).

Results. For stateful bugs, SGFUZZ is never slower than the competition in exposing the bugs. For OpenSSL, which has a more complicated state machine than others, LIBFUZZER cannot trigger the bug in 23 hours, while IJON and SGFUZZ trigger it in 6 and 10 out of 20 fuzzing campaigns of 23 hours, respectively. For *Live555*, LIBFUZZER, IJON, and SGFUZZ expose the bug much faster than AFLNET because of the high throughput. In general, IJON outperforms LIBFUZZER, but it is worse than SGFUZZ. These known stateful bugs were originally found by LIBFUZZER because few state transitions are needed (cf. Section A.3).

For stateless bugs, SGFUZZ shows a slight advantage over LIBFUZZER and IJON. We found that these stateless bugs are usually in the functions which are executed together with, but do not depend on state transitions, such as message parsing functions must be executed before state transitions (bug #554). Our algorithms explore the state space more efficiently, facilitating the stateless bug finding as well.

On average, SGFUZZ exposes stateful bugs about twice (2x) as fast as LIBFUZZER and IJON, and about 155x faster than AFLNET.

RQ.5 New Bugs Found

We found 12 unique bugs in three of four subjects from ProFuzzBench. All new bugs are listed in Table 8. All of these bugs can be exploited remotely and have serious safety risks. So far, we have received 8 CVE IDs (we are in the process of applying for CVEs for the remaining 4 bugs). Out of the 12 new bugs we found, 10 are stateful. Of those, 7 exist in the states represented by explicit state variables of enumerated type, while the other 3 stateful bugs result from aggregated implicit states (cf Section 3.4). Two stateful bugs (the first and tenth line in Table 7) can also be triggered by LIBFUZZER because they depend on implicit states. One stateless bug (the eighth line in Table 7) can also be triggered by LIBFUZZER. With the same state variables identified by SGFUZZ, IJON additionally expose two bugs (the second and third line in Table 7). All 12 new-found bugs cannot be triggered by AFLNET in 23 hours.

Case study: CVE-2021-39283[18] *in Live555* (stateful bug in explicit state). This bug is exposed with the following minimal prefix of RTSP states $\langle INIT \implies READY \implies PLAY \implies READY \implies PLAY \rangle$ and requires the following state transition in the MPEG parser (i.e., enum-type state variable *MPEGParseState*) $\langle PARSING_PACK_HEADER \implies PARSING_PACK_HEADER \rangle$, all of which are tracked in the STT. After the first SETUP command, the first PLAY command starts the parsing process and sets the parser state for the media file header to PARSING_PACK_HEADER. The second set of SETUP and PLAY commands stops and restarts the parsing process, setting the parser state again to PARSING_PACK_HEADER. However, the number of bytes read in the first round is not reset properly, such that in the second round the media file header is parsed incorrectly which incurs an assertion error.

SGFUZZ found 12 new bugs in three widely-used and well-fuzzed stateful programs within 23 hours fuzzing. 10 of 12 bugs are stateful bugs.

7 Related Work

State Space Inference. The most closely related works employ learning algorithms to infer state machines. There are two different ways to infer the state machine from a protocol implementation. *Passive learning* based approaches [8, 12, 13, 14, 22, 30] infer a state machine by analyzing a corpus of protocol messages observed from network traffic. A fundamental restriction to this kind of approach is the inability to learn protocols that communicate over an encrypted channel. *Active learning* based approaches [7, 9] actively query a protocol implementation with generated message sequences and inference a state machine using Angluin's L^* algorithm. HVlearn [21] infers DFA-models of SSL/TLS hostname verification implementations via learning algorithms. In contrast, we do not infer the state machine from a protocol implementation but derive the STT structure from execution traces to identify the state space (Sec 3.2). The construction of the STT requires no analysis of protocol messages or learning

[18] fifth line in Table 8.

Subject	version	Type	Stateful	State Variable	CVE
Live555	1.08	Stack-based overflow in liveMedia/MP3FileSource.cpp	True	Implicit	CVE-2021-38380
Live555	1.08	Heap use after free in liveMedia/MatroskaFile.cpp	True	Explicit	CVE-2021-38381
Live555	1.08	Heap use after free in liveMedia/MPEG1or2Demux.cpp	True	Explicit	CVE-2021-38382
Live555	1.08	Memory leak in liveMedia/AC3AudioStreamFramer.cpp	True	Explicit	CVE-2021-39282
Live555	1.08	Assertion in UsageEnvironment/UsageEnvironment.cpp	True	Explicit	CVE-2021-39283
Live555	1.08	Heap-based overflow in BasicUsageEnvironment/BasicTaskScheduler.cpp	True	Explicit	CVE-2021-41396
Live555	1.08	Memory leak in liveMedia/MPEG1or2Demux.cpp	True	Implicit	CVE-2021-41397
OwnTone	28.2	Heap use after free in src/misc.c	False	-	CVE-2021-38383
DCMTK	3.6.6	Memory leak in dcmnet/libsrc/dulparse.cc	False	-	CVE Requested
DCMTK	3.6.6	Memory leak in dcmnet/libsrc/dulparse.cc	True	Implicit	CVE Requested
DCMTK	3.6.6	Heap use after free in dcmqrdb/libsrc/dcmqrsrv.cc	True	Explicit	CVE Requested
DCMTK	3.6.6	Heap-based overflow in dcmnet/libsrc/diutil.cc	True	Explicit	CVE Requested

Table 8: The 0–day security vulnerabilities found by SGFuzz. Eight CVEs were assigned. Four CVEs are under application.

algorithms. We provide a new perspective of stateful system testing, in which protocol state transitions can be identified by changes on state variables.

Blackbox and Whitebox Protocol Fuzzing. *Blackbox fuzzers* [4, 15, 16, 17, 24, 25, 26, 27, 28] take the protocol server under test as a black box and keep generating message sequences to the server for discovering bugs or security flaws. Despite being simple and easy to be deployed, blackbox fuzzers are completely unaware of the internal structure of the program, perform random testing, and are thus ineffective. *Whitebox fuzzers* use program analysis and constraint solving to synthesize inputs that exercise different program paths. MACE [7] combines concolic execution and active state machine learning for protocol fuzzing. MACE uses concrete execution and symbolic execution to iteratively infer and refine an abstract model of a protocol so as to explore the program states more effectively. However, these techniques suffer from scalability issues arising from the heavy machinery of symbolic execution.

Greybox fuzzers for protocols. These fuzzers observe program states in the execution and use coverage- and state-feedback to guide input generation for discovering program states. IJON [2] uses human code annotations to annotate states. INVSCOV [10] uses the likely invariants to partition program states. AFLNet [20] uses response codes as states and evolves a corpus of protocol message sequences based on coverage- and state-feedback from executed sequences. For each run, AFLNet selects one of the states and fuzzes the entire sequence of messages which reach the state. To improve performance, [6] uses fork mechanism to snapshot the states which are under fuzzing to avoid executing the whole protocol messages from the start. However, the fork points need to be specified manually in the code which demands the knowledge of the protocol state machine. Our approach comes under *greybox* fuzzing. SGFuzz does not require knowledge of the protocol state machine. Instead, we use state variables to identify the explored state space. In contrast to AFLNet, SGFuzz does not identify states by response codes.

8 Discussion

The contribution of our work is in fuzzing reactive and stateful software systems. We propose to capture the exercised state space using a highly efficient dynamic construction of a state transition tree (STT). We developed a greybox fuzzer SGFuzz for stateful systems. Our experimental results show that SGFuzz discovered stateful bugs about twice as fast as LibFuzzer, the baseline greybox fuzzer that SGFuzz was implemented into. SGFuzz covers 33x more sequences of state transitions than LibFuzzer, 15x more than IJON, and 260x more than AFLNet. At the same time, SGFuzz achieves the same branch coverage about 2x faster than LibFuzzer. Finally, SGFuzz found 12 new bugs. Our tool SGFuzz has been released open source.

https://github.com/bajinsheng/SGFuzz

We followed a responsible disclosure policy for the bugs we found, and all reported bugs in this paper have now been fixed by developers. We feel that tools like SGFuzz have significant practical usage since the security of the internet-facing protocol implementations is of paramount importance.

Acknowledgements. This work was partially supported by the National Research Foundation Singapore (National Satellite of Excellence in Trustworthy Software Systems)

References

[1] libfuzzer-a ibrary for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2020.

[2] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy*, 2020.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 748–758, 2019.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[6] Yurong Chen, Tian lan, and Guru Venkataramani. Exploring effective fuzzing strategies to analyze communication protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, page 17–23, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Chia Yuan Cho, Domagoj Babic, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *20th USENIX Security Symposium*, 2011.

[8] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *30th IEEE Symposium on Security and Privacy*, pages 110–125, 2009.

[9] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *24th USENIX Security Symposium*, pages 193–206, 2015.

[10] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, August 2021.

[11] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data flow sensitive fuzzing. In *29th USENIX Security Symposium*, pages 2577–2594, 2020.

[12] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *Security and Privacy in Communication Networks*, pages 330–347, 2015.

[13] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. 2010.

[14] Y. Hsu, G. Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. In *2008 IEEE International Conference on Network Protocols*, 2008.

[15] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano. T-fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In *IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 323–332, 2014.

[16] Rauli Kaksonen, Marko Laakso, and Ari Takanen. Software security assessment through specification mutations and fault injection. In *Communications and Multimedia Security Issues of the New Century*, pages 173–183, 2001.

[17] T. Kitagawa, M. Hanaoka, and K. Kono. Aspfuzz: A state-aware protocol fuzzer based on application-layer protocols. In *The IEEE symposium on Computers and Communications*, 2010.

[18] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 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*, pages 1393–1403, 2021.

[19] Roberto Natella and Van-Thuan Pham. Profuzzbench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[20] Van-Thuan Pham, Marcel Boehme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *IEEE International Conf. on Software Testing Verification and Validation*, 2020.

[21] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana. Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *IEEE Symposium on Security and Privacy*, 2017.

[22] Yipeng Wang, Zhibin Zhang, and L. Guo. Inferring protocol state machine from real-world trace. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, 2010.

[23] Website. American fuzzy lop (afl) fuzzer. http://lcamtuf.coredump.cx/afl/technical_details.txt, 2017. Accessed: 2017-05-13.

[24] Website. beSTORM Black Box Testing. https://www.beyondsecurity.com/bestorm.html, 2017.

[25] Website. Boofuzz: A fork and successor of the sulley fuzzing framework. https://github.com/jtpereyda/boofuzz, 2017. Accessed: 2019-08-12.

[26] Website. Codenomicon Intelligent Fuzz Testing. http://www.codenomicon.com/index.html, 2017. Accessed: 2017-05-13.

[27] Website. Peach Fuzzer Platform. http://www.peachfuzzer.com/products/peach-platform/, 2017. Accessed: 2017-05-13.

[28] Website. Sulley: A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley, 2017. Accessed: 2019-08-12.

[29] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786. IEEE, 2019.

[30] Yingchao Yu, Zuoning Chen, Shuitao Gan, and Xiaofeng Wang. Sgpfuzzer: A state-driven smart graybox protocol fuzzer for network protocol implementations. *IEEE Access*, 8:198668–198678, 2020.

[31] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324. USENIX Association, August 2020.

Subject	HONGGFUZZ	LIBFUZZER	SGFUZZ$_x$	SGFUZZ$_y$	SGFUZZ
H2O	82.65	70.80	114.50	1693.45	1849.30
MbedTLS	18.95	22.80	32.55	49.40	50.80
Curl	144.60	150.25	456.70	8899.00	14630.80
Gstreamer	48.70	49.40	186.50	2399.10	4067.30
OpenSSL	25.34	23.95	30.00	31.03	33.10
Live555	179.65	184.15	594.33	825.25	1162.30
OwnTone	49.71	46.40	675.45	748.32	930.15
DCMTK	183.22	189.25	1233.45	3755.05	6737.05

Table 9: Average state transition coverage for 20 campaigns of 23 hours.

Subject	HONGGFUZZ	LIBFUZZER	SGFUZZ$_x$	SGFUZZ$_y$	SGFUZZ
BRANCH COVERAGE:					
H2O	3207.90	2879.20	2885.50	3042.70	3050.68
MbedTLS	11735.15	11929.60	11890.35	11902.05	12057.68
Curl	19987.30	19262.16	16665.80	19619.77	19942.39
Gstreamer	63235.45	63280.06	61807.94	62551.20	61698.56
OpenSSL	13003.46	12463.14	12598.24	12596.10	12610.50
Live555	2197.85	2244.63	2223.15	2270.00	2278.10
OwnTone	2104.13	2184.64	2124.75	2292.00	2312.60
DCMTK	6087.33	6042.15	5992.25	6033.70	6100.50
TIME-TO-COVERAGE:					
H2O	3.60h	23.00h	12.41h	7.16h	6.69h
MbedTLS	>23.00h	23.00h	>23.00h	>23.00h	17.86h
Curl	14.25h	23.00h	22.58h	13.88h	12.23h
Gstreamer	>23.00h	23.00h	>23.00h	>23.00h	>23.00h
OpenSSL	5.54h	23.00h	15.44h	16.32h	10.08h
Live555	>23.00h	23.00h	>23,00h	19.33h	15.86h
OwnTone	>23.00h	23.00h	>23.00h	8.18h	5.04h
DCMTK	20.11h	23.00h	>23,00h	>23.00h	14.40h

Table 10: Average branch coverage for 20 campaigns of 23 hours (top), and the time it takes each fuzzer to achieve the same coverage that LIBFUZZER achieves in 23 hours (0.25h accuracy; bottom)

A Appendix

Due to the space limit, we present some of the evaluation results here.

A.1 Sensitivity Analysis

We conducted sensitivity analysis to evaluate the effectiveness of each heuristic in Section 4. To do this, we configured three new fuzzers at the FUZZBENCH framework: a) SGFUZZ$_x$, a specific configuration of SGFUZZ that does not have the algorithms of energy schedule in Section 4.2 and prioritizing input bytes in Section 4.3. b) SGFUZZ$_y$, another configuration of the SGFUZZ that does not have prioritizing input bytes in Section 4.3. c) HONGGFUZZ, a stateless fuzzers similar to LIBFUZZER. The experiments were carried out with the identical configurations as our main experiments: 23 hours and 20 runs.

Table 9 shows the average state transition coverage. Similar to LIBFUZZER, HONGGFUZZ is a stateless fuzzer, so its state transition coverage is little and close to LIBFUZZER. This demonstrates that other stateless fuzzers are inefficient in state-space exploration as well. From SGFUZZ$_x$, to SGFUZZ$_y$, until SGFUZZ, the state transition increase factor to LIBFUZZER are 4.43x, 21.95x, and 33.92x on average which represent the contributions of each algorithm in Section 4.1, Section 4.2, and Section 4.3, respectively. It demonstrates that our heuristic algorithms (Section 4.2 and Section 4.3) promote around 33x state transition coverage. Looking together with Table 3, SGFUZZ$_x$, which only has STT feedback, still slightly outperforms IJON. We explain that IJON only records the last four states in a state sequence for high throughput, while we capture all states of a sequence in the STT.

Table 10 shows the average branch coverage in 23 hours and the time it takes HONGGFUZZ, SGFUZZ$_x$, SGFUZZ$_y$, and SGFUZZ, respectively to achieve the same coverage that LIBFUZZER achieves in 23 hours. HONGGFUZZ excels at subjects *H2O*, *Curl*, and *OpenSSL* and fails miserably at *MbedTLS* and *OwnTone*. The reason for this is that SGFUZZ$_x$, SGFUZZ$_y$, and SGFUZZ are all based on LIBFUZZER, while HONGGFUZZ is an entirely different fuzzer. In general, SGFUZZ$_x$ slightly performs worse than LIBFUZZER as it introduces performance cost for state tracing, and our heuristic algorithms overcome the performance cost as in SGFUZZ$_y$ and SGFUZZ. For *Gstreamer* which has a big state space in small code space, SGFUZZ$_y$ performs better than SGFUZZ as the latter puts too much effort on the state space exploration. Time-to-coverage improvement is substantially greater than branch coverage improvement, which is consistent with our major results in Table 4.

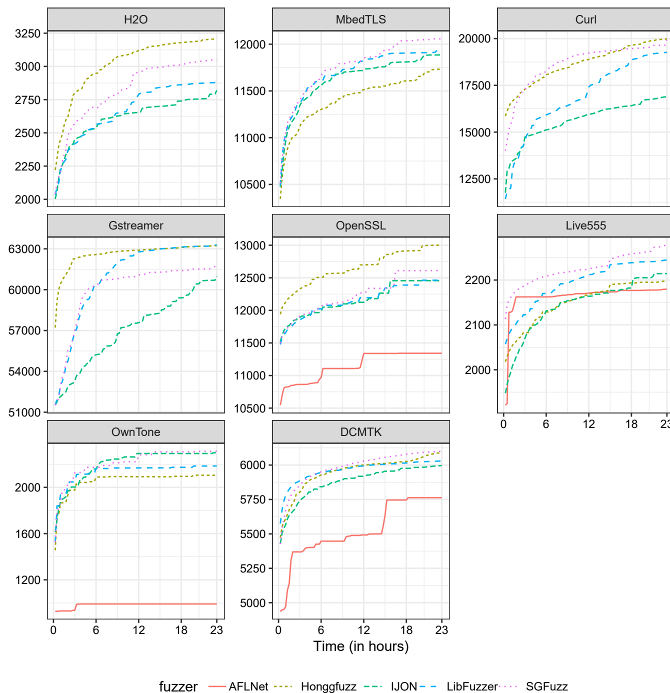

Figure 5: Average branch coverage over time for SGFUZZ and LIBFUZZER across all eight subjects in 23 hours.

A.2 Growth Trend of Branch Coverage

Figure 5 shows the average branch coverage with time for all fuzzers in Table 4 and HONGGFUZZ in Table 10. All fuzzers' branch coverage is nearly saturated after the 12 hours, although their state transition coverage is vastly varied. The slowing growth rate of branch coverage further explains why the time-to-coverage improvement is more significant than branch coverage improvement and branch coverage alone isn't enough to examine program state space. AFLNET is substantially slower than other fuzzers, especially on OwnTone. On OwnTone, AFLNET 's throughput is less than 10 inputs/second, while SGFUZZ 's throughput is above 1000 inputs/second. Therefore AFLNET achieves much fewer branch coverage than others.

A.3 Prevalence of Stateful Bugs

To investigate the prevalence of stateful bugs, we studied the bugs that were reported in OSS-Fuzz against our subjects. We use the keyword `subject/fuzz_driver status:Verified -Type=Build-Failure` to search available bugs on the OSS-Fuzz platform[19]. These bugs were reported against previous versions, and these reports are publicly available. A bug report links the bug fix and a crash-reproducing test case. We used this test case to debug these

[19] https://bugs.chromium.org/p/oss-fuzz/issues/list

Subject	Total	Stateful	Subject	Total	Stateful
H2O	9	6	Gstreamer	34	28
OpenSSL	4	3	Curl	26	20
MBedTLS	0	0			
			Total:	73	57 (78.08%)

Table 11: Prevalence of stateful bugs in OSS-Fuzz as of Sep'21.

crashes and to determine whether these bugs are stateful. Table 11 illustrates the prevalence of stateful bugs. We classify a bug as *stateful* if the crash location is reached after at least one state transition. We find that 57 (78.08%) of all 73 bugs are stateful.

> Every four in five bugs that are reported in OSS-Fuzz for protocol implementations among our subjects are stateful.

To understand the nature of bugs in protocol implementations, let us discuss all the bugs in H2O, our motivating example in Section 2.

Stateful bugs in H2O. All 6 stateful bugs in H2O happen after specific state transitions on the state variable *stream->state* explained in Section 2. Each stream in HTTP2 protocol has a weight value to represent the priority of the stream to be handled and H2O defines a scheduling component to maintain it. Bugs #12093, #12096, #12127, and #12100 have the same root reason that they happen in the *END_STREAM* state of *stream->state* within the scheduling component, so we only explain bug #12096 here. To trigger it, the H2O *firstly* needs to set a higher priority for the current stream, (e.g., via the header frame or a dedicated priority frame), and *then* close the stream (e.g., via a "reset stream" or an "end stream" frame to trigger a scheduler relocation). At the same time, the value of *stream->state* will be changed from *IDLE* until *END_STREAM* representing a sequence of state transitions. The scheduling component only works for active streams, not closed streams. Lastly, when we assign a new priority to the closed stream, the bug 12096 is triggered because it does not check if the stream is closed. Bug #2623 and #3303 have the same root reason that they happen under the *END_STREAM* state of *stream->state*, but not within the scheduling component. We explain the bug #2623 here. To trigger it, H2O *firstly* receives a header frame with the end-of-stream attribute which indicates that no more frames for the stream. *Then* the stream will be closed as the corresponding *stream->state* is changed to *END_STREAM*. Lastly, when receiving another frame called trailing header for the closed stream, H2O may trigger the bug #2623 because of incomplete checking.

Stateless bugs in H2O. The 3 stateless bugs in H2O are observed before processing requests where no state transitions execute. Bug #2695 #2923 happens in the response generation component. H2O defines an object called *generator* to maintain related operations of response generation, but the *generator* is used without checking if it is NULL incurring

unexpected behaviors. Bug #37023 happens in the socket connection functions which is also executed before processing inputs without state transitions.

Totally, 6 out of 9 bugs in H2O are stateful. Although existing fuzzer can find stateful bugs, the stateful bugs they found only happen at the *END_STREAM* state of the *stream->state* which represents the endpoint of the stream state. No bug is found during the state transitions except for the start and end states. It provides further evidence to support that the existing fuzzer cannot efficiently explore the state space of the protocol programs. On the other hand, even if the state space of the protocol program has not been explored enough, the number of stateful bugs is still greater than stateless bugs. Therefore, we claim that the stateful bugs are prevalent in protocol implementations.

A.4 Top-50 most widely used open-source protocol implementations.

To investigate how states are tracked in the programs, We reviewed the code of top-50 widely used protocol implementations that are open-sourced. We used the same criterion as in Section 6 to examine the state variables: the values of state variables are got from the input and affect the program execution by the switch or if statements. We only show the first state variables if several state variables exist in a subject.

Table 12 shows the results. It includes more than 16 protocols and 50 corresponding subjects. We marked the corresponding protocol of the state variable in the *Protocols* column or noted *Customized* if the state variables are used for customized states, such as connection and data processing states, not protocol states. Although some subjects (Sendmail, Postfix, e.g.,) are protocol implementations, the state variables we found in them are about customized states, not protocol states. This result explains that developers usually design many customized states in programs, and these states can not be found by protocol documents. The column of *Variable Types/Values Examples* represent the state variable types: enumerated type or macro definition (#define), and an example of state variable values. All 50 subjects implement their states by named constants, of which 44 subjects use enum-type variables.

Case study. We used *OpenSSL* to explain how the program states are manipulated by the corresponding state variables. *Openssl* has a **handshake state machine** to maintain the state during the negotiation process between the two parties to establish a connection. OpenSSL defines fifty state values[20], partially ordered, to implement it. We focus on the states on the server-side. At first, it is in the idle State (TLS_ST_BEFORE). When receiving HELLO message from the client, it enters State (TLS_ST_SR_CLNT_HELLO), sends other optional messages one by one in States (DTLS_ST_SW_HELLO_VERIFY_REQUEST to TLS_ST_SW_CERT_REQ), and indicates it is done by State (TLS_ST_SW_SRVR_DONE). Then when the client is ready and sends back optional messages, the server checks them in these States (TLS_ST_SR_CERT to TLS_ST_SR_FINISHED, TLS_ST_SW_CERT_VRFY). At last, it sends final messages in States (TLS_ST_SW_SESSION_TICKET to TLS_ST_SW_ENCRYPTED_EXTENSIONS) to confirm the server is ready too. States (TLS_ST_SW_KEY_UPDATE, TLS_ST_SR_KEY_UPDATE) are used for updating keys after establishing the connection.

Protocols	Subjects	Variables Types/ Values Examples
FTP	Bftpd	enum / STATE_CONNECTED
	LightFTP	macro / FTP_ACCESS_NOT_LOGGED_IN
	vsftpd	macro / PRIV_SOCK_LOGIN
SFTP	ProFTPd	macro / SFTP_SESS_STATE_HAVE_AUTH
TLS/SSL	BoringSSL	enum / state13_send_hello_retry_request
	libssh2	enum / libssh2_NB_state_idle
	MbedTLS	enum / MBEDTLS_SSL_HELLO_REQUEST
	OpenSSL	enum / TLS_ST_BEFORE
	WolfSSL	enum / SERVER_HELLOVERIFYREQUE..
	Botan	enum / HELLO_REQUEST
SMTP	Exim	enum / FF_DELIVERED
	OpenSMTPD	enum / STATE_HELO
HTTP/2	aolserver	enum / SOCK_ACCEPT
	H2O	enum / H2O_HTTP2_STREAM_STATE_IDLE
	httpd	enum / H2_SS_IDLE
	LiteSpeed httpd	enum / STREAM_FIN_RECVD
	NgHttp2	enum / NGHTTP2_STREAM_INITIAL
	Nginx	enum / NGX_HTTP_INITING_REQUEST...
	Tengine	enum / NGX_HTTP_TFS_STATE_WRITE...
	thttpd	macro / CHST_FIRSTWORD
	WebSocket++	enum / READ_HTTP_REQUEST
	Lighttpd	enum / LI_CON_STATE_DEAD
RDP	FreeRDP	enum / DRDYNVC_STATE_INITIAL
	xrdp	enum / WMLS_RESET
NTP	NTP	macro / EVNT_UNSPEC
IMAP	Dovecot	enum / AUTH_REQUEST_STATE_NEW
IRC	UnreadIRCd	enum / CLIENT_STATUS_TLS_STAR...
SMB	Squid	enum / SMB_State_NoState
DAAP	OwnTone	enum / PLAY_STOPPED
SIP	Kamailio	enum / H_SKIP_EMPTY
DICOM	DCMTK	enum / DIMSE_StoreBegin
VNC	libvncserver	enum / RFB_INITIALISATION_SHARED
	TigerVNC	enum / RFBSTATE_UNINITIALISED
RTSP	ffmpeg	enum / RTSP_STATE_IDLE
	VideoLAN	enum / VLC_PLAYER_STATE_STOPPED
MQTT	Mosquitto	enum / mosq_ms_invalid
Customized	aria2	enum / STATUS_ALL
	inetutils	marco / TS_DATA
	Gstreamer	enum / GST_STATE_READY
	OpenSSH	enum / MA_START
	Pure-FTPd	enum / FTPWHO_STATE_IDLE
	libgcrypt	enum / STATE_POWERON
	Postfix	enum / STRIP_CR_DUNNO
	Sendmail	enum / SM_EH_PUSHED
	Boa HTTPd	enum / READ_HEADER
	WebServer	enum / STATE_PARSE_URI
	tevent	enum / TEVENT_REQ_INIT
	Live555	enum / PARSING_VIDEO_SEQUENCE...
	OpenMQTTGateway	enum / OFF
	Curl	enum / MSTATE_INIT

Table 12: The top-50 most widely used open-source protocol implementations and their state variables.

[20]https://github.com/openssl/openssl/blob/5811387bac39cdb6d009dc0139b56e6896259cbd/include/openssl/ssl.h.in#L1026