yFuzz: a yield-driven progressive fuzzer for stateful communication protocols

Abstract—**While coverage-based greybox fuzzing has gained great success in the field of vulnerability detection due to its simplicity and efficiency, it could become less powerful when applied directly to protocol fuzzing due to the unique challenges of protocol fuzzing. In particular, 1) The implementation of protocols usually involves multiple program binaries, i.e., multiple fuzzing entries; 2) The communication among multiple ends contains more than one packet, which are not necessarily dependent upon each other, i.e., fuzzing single (usually the first) packet can only achieve extremely limited code coverage. In this paper, we study such challenges and demonstrate the limitation of current non-stateful greybox fuzzer. In order to achieve higher code coverage, we design and implement a stateful, yield-drive protocol fuzzer, *yFuzz*, to explore the code related to different protocol states. yFuzz incorporates a stateful fuzzer (which contains a state switching engine) together with a multi-state forkserver (which enables multi-state program forking) to consistently and flexibly fuzz different states of an compiler-instrumented protocol program. Our experimental results on OpenSSL show that yFuzz achieves an improvement of 73% more code coverage and 2X more unique crashes when comparing against fuzzing the first packet during a protocol handshake.**

I. Introduction

Vulnerabilities in network protocols (such as Heartbleed in OpenSSL and Remote Code Execution in SNMP) are among the most devastating security problems since their exploitation typically exposes hundreds of thousands of networked devices to catastrophic risk. Efforts have been made toward developing automatic and scalable techniques to detect vulnerabilities in large protocol codebase. In particular, fuzzing has gained increasing popularity due to its simplicity and efficiency in practice, as compared to other testing techniques such as symbolic/concolic executions.

A network protocol defines a set of rules that allow multiple entities to interact with each other, e.g., through packet exchanges. Existing protocol fuzzers can be broadly categorized into two classes. 1) The fuzzer is part of the communication chain by either directly mimicking a client/server in the protocol or acting as a Man-In-The-Middle (MITM) proxy. It generates/intercepts packets among multiple network entities, mutates and relays them. This could be implemented as a whitebox fuzzer (where the protocol specifications are known), or a blackbox fuzzer (where the protocol is reverse-engineered through packet analysis). 2) The fuzzer works together with a proper testing program (TP) provided for the network protocol. This is often known as a greybox fuzzer. It feeds the mutated inputs to the TP, which is responsible for executing the protocol, while the fuzzer stays out of the communication between clients and servers. For example, OpenSSL has several "official" testing programs [22] for LibFuzzer [26] and AFL [34].

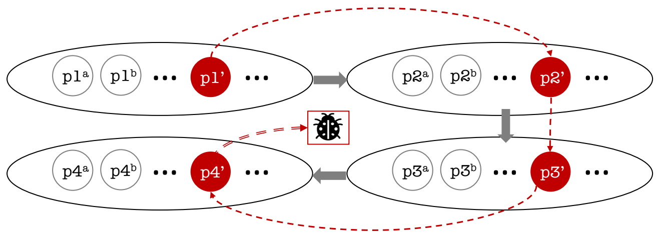

Fig. 1. Fuzzing a four-packet-flight communication protocol: Different packets/fields are independent, e.g., each variation of packet $p2$ may lead to a different subsequent packet $p3$, $p4$. The bug can only be triggered when following the exact sequence $p1' \to p2' \to p3' \to p4'$, driving the protocol to progressively traverse the corresponding states and eventually reach the bug.

In particular, a whitebox protocol fuzzer such as $sulley$ [1] and $boofuzz$ [24] assumes that the user knows the packet formats, thus she is able to construct packets based on this knowledge and to mutate packet fields separately. It monitors the program execution and network behaviors of the server/-client, to detect possible failures triggered by the mutated packets/inputs. On the other hand, a blackbox protocol fuzzer [15], [3], [12] typically consists of a MITM proxy, a packet reverse engineering module, and a mutation engine. Since the packet formats and protocol states are unknown, the fuzzer starts with packet monitoring, reverse engineering and packet clustering. Then, mutations are applied to each packet cluster based on state transitions and information learned. While both whitebox and blackbox protocol fuzzers perform "blind" fuzzing and fail to leverage some useful program execution information, the blackbox fuzzers particularly suffer from the inaccuracy of protocol reverse engineering. Finally, a greybox fuzzers [34], [29], [26], [6], [32], [16], [5], [19], [30] instruments the TP to track runtime information such as code coverage and dynamic data flow, to guide future testcase generation and the exploration of additional code. Extensive studies have focused on better testcase generations and TP transformation [20], [23] for efficient greybox fuzzing.

Limitations and Challenges: Despite recent progress on fuzzing tools, a number of fundamental limitations and challenges still exist. 1) Communication protocols are typically implemented through state machines on servers/clients with state transitions driven by critical protocol events such as packet/message exchange. Although proxies can be employed to mutate and fuzz packets on the fly [4], the fuzzing is often not stateful and lacks the ability to drive protocol to a specific state of interest, trap it in the state, and keep replaying and fuzzing it (e.g., with different packet inputs). Stateful fuzzing is necessary for communication protocols. 2) A general program takes inputs when being launched, and the execution status depends solely on the inputs (excluding irrelevant factors such as system status and user interruptions). However, in protocols, there are multiple rounds of message flights that contain both independent and dependent packets/fields. Simply fuzzing one single packet/field limits achievable code coverage, whereas mutating packets/fields together may lead to inconsistent (and invalid) message flights. Protocol fuzzers need to identify the dependence and adapt its fuzzing strategies accordingly. 3) Finally, protocol implementations need to be properly instrumented and transformed in order to be directly fuzzed by popular tools such as AFL [34] and VUzzer [25]. For instance, socket communications need to be logged and converted to direct data operations for higher efficiency, and servers and clients need to be integrated into a single program for effective execution in a coordinated fashion. For example, the programs in DARPA CGC (Cyber Grand Challenge) that involve communications between multiple entities cannot be directly fuzzed by tools such as Driller [28] (which contains an AFL component).

We illustrate the inefficiency of stateless and individual-packet fuzzing in Fig. 1. The example involves four packets in a complete round of handshake, and the packets are partially interdependent, e.g., the response to $p1^a$ could be $p2^a$ or $p2^b$ and so on. The buggy code is only executed when the packet sequence exactly follows $p1' \rightarrow p2' \rightarrow p3' \rightarrow p4'$. Simply fuzzing the first packet could help us discover $p1'$. However, a single execution with $p1'$ may not always trigger response $p2'$ and subsequent $p3', p4'$, while continuing to mutate the first packet can only generate different variations of $p1'$, getting us no closer to $p2'$. On the other hand, if the fuzzer starts fuzzing from $p2$ given a random $p1$ (say, $p1^a$), the bug will not be triggered either. Therefore, a protocol fuzzer needs have the ability to trap the protocol execution inside a state right after $p1'$ and to keep replaying and fuzzing the state repeatedly. It enables the protocol fuzzer to move forward in a progressive manner - by moving into (and focusing on) a new state after finding $p2'$, and then $p3', p4'$ - and eventually finding the bug more efficiently.

Our proposal: A protocol fuzzer needs to be stateful. To achieve maximum code coverage and fuzzing depth, it should be able to identify, replicate, and switch between different protocol states while maintaining execution consistency. In this paper, we propose yFuzz, a yield-driven progressive fuzzer for stateful communication protocols. It (i) makes novel use of a multi-state forkserver to fork and replicate protocol execution states, (ii) intelligently replay selected protocol states, based on both state information and fuzzing yield, e.g., code coverage and unique crashes, fuzz the corresponding packets, and switch states when necessary. In particular, yFuzz consists a *state-aware fuzzer*, a *multi-state forkserver* and an *instrumented testing program (TP)*. The state-aware fuzzer builds multiple fuzzing states across the TP execution and identifies the corresponding fuzzing targets (i.e., packets and fields) for different fuzzing states. It then commands the forkserver when to replicate protocol states, progress (move forward to the next fuzzing state), and regress (roll back to the previous fuzzing state), based on the fuzzing yield achieved on the fly. The instrumented TP, once cloned by the forkserver, will take the mutated inputs from the state-aware fuzzer, execute them from the current protocol state, and send back the status information to the fuzzer.This feedback is analyzed by the state-aware fuzzer to decided next fuzzing state. The state-aware fuzzer, multi-state forkserver and instrumented TP work in concert

Algorithm 1: Behavior of default $\_\_AFL\_INIT()$. The logic is simplified.

```
1  while TRUE do
2      receive forking signal from fuzzer;
3      pid = fork();
4      if pid < 0 /*fork failed*/;
5       then
6          exit();
7      if pid == 0 /*in child process*/;
8       then
9          return; /*continue execute TP*/
10     /*in parent process*/;
11     send pid to fuzzer;
12     wait for child to exit;
13     send child exit status to fuzzer;
```

to identify *progression seeds* (testcases that change the protocol states), change fuzzing states, and explore the program execution space efficiently. For the example in Fig. 1, yFuzz will first fuzz $p1$ to find $p1'$, then save the execution state (by forking) to continue fuzzing $p2$ given $p1'$, and continue the same process for $p3'$ and $p4'$. The fuzzing state may move forward and backward several times during progression and regression to identify the sweet spot for highest fuzzing yield, until the fuzzer can no longer find interesting testcases.

In summary, this work makes the following contributions.

- We propose a novel framework, yFuzz, for stateful protocol fuzzing. It consists of three key components, a state-aware fuzzer, a multi-state forkserver and an instrumented TP, which work in concert to identify, replicate, and switch between different protocol states while maintaining execution consistency during fuzzing.
- Leveraging the multi-state forkserver, yFuzz has the ability to intelligently replay selected protocol states, fuzz the corresponding packets, and switch states when deemed necessary using both state information and fuzzing yield, e.g., code coverage and unique crashes.
- We enable flexible power schedules[1] to fully capitalize the potential of yFuzz. In this paper, we implement and evaluate a new yield-driven power schedule that continuously focuses on fuzzing the (current) most rewarding protocol states.
- Our experimental results of fuzzing the OpenSSL library show that yFuzz is able to achieve 1.73X code coverage and discover 3X unique crashes (on average) compared with the default AFL.

II. Background

A. Overview of AFL

AFL [34] is a popular coverage-guided greybox fuzzer. It maintains a queue for the file paths of the testcases. Starting from the seed testcase provided by the user, AFL will select one testcase at a time, map the file from disk to a memory buffer for mutation. Each testcase will go through multiple

[1]The power schedule is the policy of assigning time to each testcase. In yFuzz , power schedule also denotes the time spent on each protocol state.

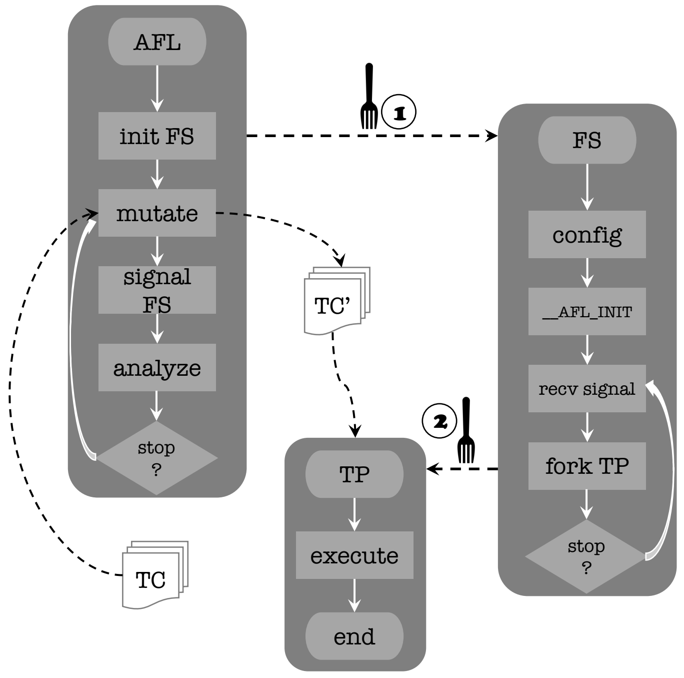

Fig. 2. Simplified AFL forking workflow. FS: forkserver, TC/TC': testcase, TP: testing program

rounds of mutation with various mutation operations (such as bit flips, additions, replacement and so on). After each mutation, the modified buffer will be written to a file, which will be the input to the TP. Then the fuzzer will signal TP to execute and wait for the execution to finish to collect information such as code coverage and exit status.

Instead of blindly generating testcases to the TP, AFL utilizes compile-time instrumentation to track the basic block transitions in the TP. Each basic block of TP will have a unique ID and the pair of two IDs can represent the control flow transitions (called **edges**, which we will use throughout the following sections). AFL stores the occurrence of edges in a 64KB memory (shared between the fuzzer and TP). For each execution, TP will update the shared memory about the edges information and the fuzzer will get such information. If new edges occur or the numbers of edge occurences change (counts are categorized by value range buckets), the fuzzer will consider the current testcase as an interesting one. Such testcases will be appended to the queue for further mutation. Intuitively, the testcase that can result in more code coverage will get more attention and serve as the base for later mutations. (For binary-only fuzzing, instead of compile time instrumentation, AFL employs QEMU [2] to perform coverage tracking.)

Forkserver: In order to accelerate the fuzzing process, AFL develops a forkserver to avoid repeated program initializations. Without the forkserver, the fuzzer would call $execve()$ to run the TP every time a new testcase is generated. And the TP will have to start from beginning, e.g., the library initialization and dynamic linking. Such process could occupy a large ratio of the total execution time. In fact, it is the part of code after reading the input that can affect the code coverage in most of the programs, and such repeated program initializations are redundant. Hence, AFL designs a forkserver to call $execve()$ only once. The forkserver logic works as follows. The fuzzer will launch after some configurations, it will call $fork()$ to generate the forkserver. The forkserver will perform some configuration first and then call $execve()$ to execute the TP. The TP will execute until a function called $\_\_AFL\_INIT()$ (which is placed at TP by users at desired positions beforehand)[2]. In $\_\_AFL\_INIT()$, the TP will enter an infinity $while$ loop as shown in Algorithm 1. Every time the fuzzer finishes one mutation of the input and generates a new testcase, it will send a signal to the forkserver, saying that the new input file is ready. The forkserver then receives such signal from the fuzzer and calls $fork()$ to generate a cloned TP that reads the input and execute till exit. The forkserver also collects the exit status (by $waitpid()$) of TP and sends it to the fuzzer. The work flow of forkserver is shown in Fig. 2. The fork ① denotes the forking in fuzzer to generate forkserver and the fork ② denotes the forking in forkserver to generate the process that reads inputs and executes as in normal TP. In this way, the $execve()$ is called only once in forkserver. After that, the forkserver can simply clone itself from the point where program initialization is already done. Note that if the functoin $\_\_AFL\_INIT()$ is not specified by the user, AFL will automatically put it right before $main()$ function. However it would be better for the user to put it manually in a later stage of the program to skip more program initialization/computation. This is done via "deferred forkserver mode". The forking mechanisms of these two approaches are the same.

B. Motivating example

Though the forkserver mechanism "speeds up the fuzzing of many common image libraries by a factor of two or more" [35], one obvious limitation is that, it can only deal with one program state at a time. The forking point of the TP is fixed before runtime, and there is only one forkserver that maintain the TP process. while it works on general single-state programs (such as image processing softwares, PDF viewers, $binutils$ and so on), it will not suffice when it comes to multi-state program (such as protocol) fuzzing. Our experiments show that single-state fuzzing performs poorly on a simple OpenSSL handshake testing program. During the OpenSSL handshake process, while fuzzing the first and second packet yield similar code coverages at around 10%(using the same amount of time), fuzzing the third and fourth packet suffer from extremely low code coverages. When fuzzing late-stage packets, the program execution is close to its end, and the amount of remaining code available for exploration is significantly limited. However, the same experimental results of code coverage composition also show that there exists unique code related to each packet, which means that simply fuzzing any single packet will not be able to explore them all (as shown in Fig. 8). The details will be explained in Section V.

The forkserver framework in AFL cannot handle multi-state protocols. As shown in Fig. 3, there will be typically four packet flights between a ssl client and server (as in

[2]The function $\_\_AFL\_INIT()$ is visible to users, it is actually a macro that represents the function $\_\_afl\_manual\_init()$, which will call the function $\_\_afl\_start\_forkserver()$, where the infinite $while$ loop truly resides. However, to simply the description, we will use $\_\_AFL\_INIT()$ throughout this paper.

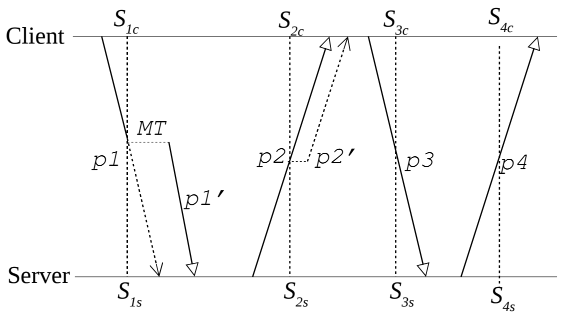

Fig. 3. Forking example of a four-stage SSL handshake

OpenSSL version 1.0.1) when performing handshake: $p1$: *client_hello*, $p2$: *server_hello* (with optional *certificate*, *server_key_exchange*) +*server_hello_done*, $p3$: *change_cipher_spec* (with optional *client_key_exchange*) and $p4$: *change_cipher_spec*. Suppose we have a TP that is already programmed to combine client and server in a single program binary and socket communications are already converted to file operations. The packet will be first generated by the sender, and be put into memory buffer. The receiver will read the packet from the buffer then process it and respond. If the default AFL-style fuzzing is applied to fuzz $p1$, the *__AFL_INIT*() will be placed right after $p1$ is generated by the client, before it is read by the server, such that the testcase generated by AFL $p1'$ can be read from file and used to replace $p1$ in the memory buffer. The server will then read $p1'$ instead of p1 to continue program execution.

However, later-stage packets may not be completely dependent on $p1$. There could be extension fields in $p2$ that are only determined by the server but not the client side, which means that no matter how we mutate $p1$, there are still some code related to $p2$ that cannot be covered. This is different from general programs where the TP take inputs once at the beginning and no independent inputs are needed.

Consider coverage-guided fuzzing, where if $p1'$ is found to be valid and interesting, what AFL will do is to add $p1'$ to the testcase queue and mutate it later to generate more related testcases, which will never affect the independent fields in $p2$. It's worth keeping $p1'$ **as well as the current program state of the client and server**, to continue fuzzing $p2$ to explore code related to the independent fields in $p2$. In order to fuzz $p2$ using interesting $p1'$, we need to accomplish the following.

- We will need to transfer the "forking point" from $p1$ to $p2$ such that the forkserver can continue to work. If the forking point (the position of function *__AFL_INIT*()) is right after $p2$ is generated, then the "new" forkserver will fork every time the fuzzer generates $p2'$ and $p2$ can get replaced.
- The packet $p1'$ need to be reused and the program execution before sending $p1'$ should remain unchanged. To reuse $p1'$, an intuitive idea is to store the interesting $p1'$ and perform packet replay to further fuzz $p2$. However, it will not work in protocols that involve randomness (which is true in SSL). When interesting $p1'$ occurs, it is bound with the current state of client S_{1c} as shown in Fig. 3. If we restart client to replay $p1'$, the state of client is no longer S_{1c}. Hence, we need to keep the exact program state S_{1c} when $p1'$ occurs.

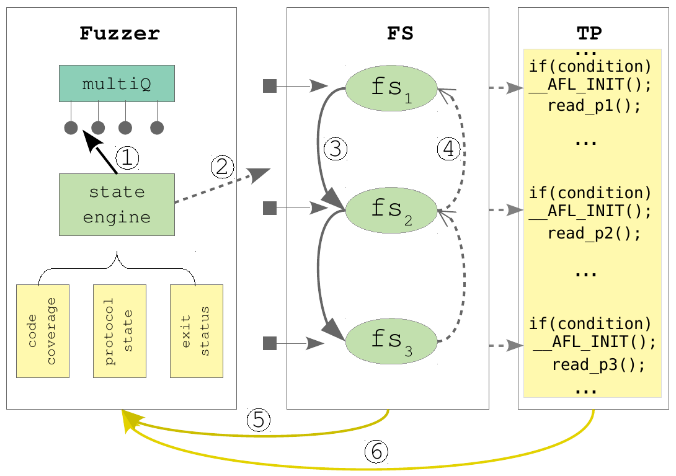

Fig. 4. System Overview of yFuzz . FS: forkserver, multiQ: queues for storing different types of testcases, TP: testing program

In yFuzz , we implement multi-state forkserver to achieve the state trapping and transition. In the fuzzer side, yFuzz has a state decision engine that commands the forkserver and TP to work on the same state and fuzz the target packet. The code coverage mechanism in yFuzz is similar to that in AFL.

C. Focus of this work

Apart from better testcase generation as proposed in existing greybox fuzzers, efficient protocol fuzzing requires that the fuzzer is able to recognize the state changes in the TP and to fuzz different packets flexibly during the program execution. In this work, we aim to design and implement a state-aware progressive fuzzing framework for protocols to achieve an fast and deep code exploration. The framework of yFuzz can also be easily adapted to fuzz other stateful programs.

III. System Design

A. Overview

yFuzz achieves stateful protocol fuzzing by combining a state-aware fuzzer(Section III-C), a multi-state forkserver (Section III-D) and an instrumented TP(Section III-B) as shown in Fig. 4. The fuzzer contains an array of queues. Each queue is used to only store the testcases that belong to the same fuzzing stage. For the example in Fig. 3, there will be four queues for $p1$, $p2$, $p3$ and $p4$. The fuzzer will collect the execution status and code coverage information after one execution of TP, then decides whether to move forward (*progression*) or backward (*regression*). The corresponding queue will be chosen based on such decision to store the target testcases. Meanwhile, the fuzzer also sends such decision to the forkserver. The forserver will then fork at the right point by moving one step forward or backward (detailed in Section III-D). When the forkserver is ready, it will keep listening to the fuzzer, waiting for signals to fork and generate a new process of TP. An overview yFuzz workflow is described in Algorithm 2. We will explain each module separately in the following subsections. At the end of this section, we will detail the communication and cooperation among different modules.

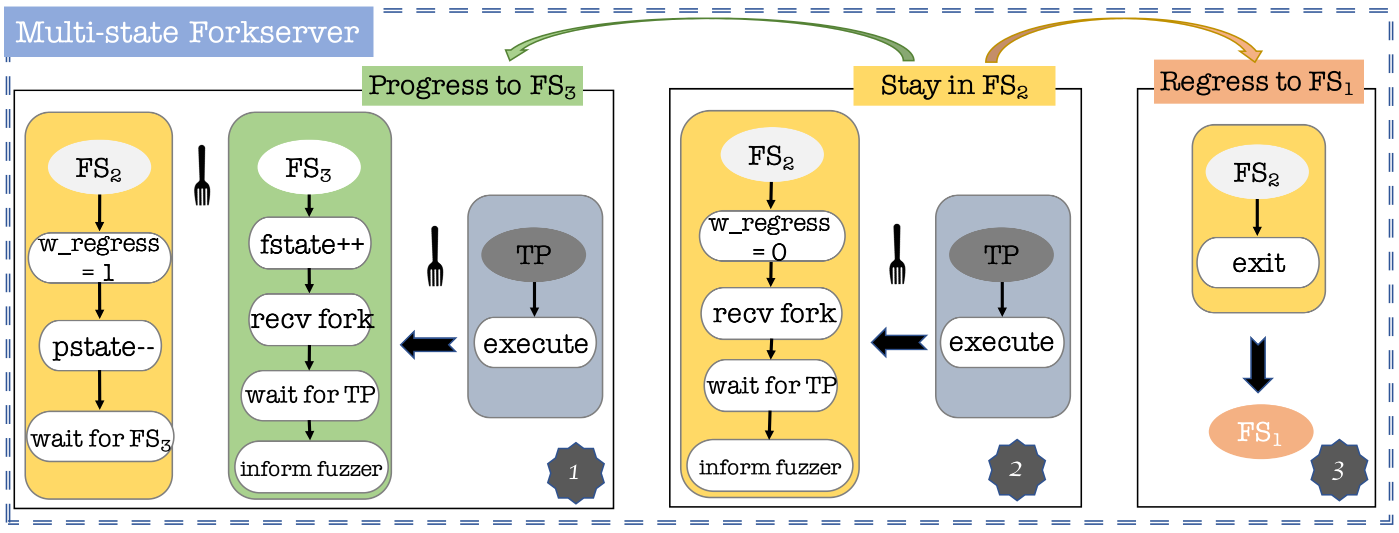

Fig. 5. Multi-state forkserver in yFuzz . Suppose current state is FS_2 and the next fuzzing state could be one of the three: staying in FS_2, progressing to FS_3 and regressing to FS_1.

B. Testing program

For better understanding, we first show the changes of the TP for protocol fuzzing in this subsection, then explain how the fuzzer and forkserver work with the TP in following subsections. The difference between a protocol implementation and a general single-state program is that, there are multiple "inputs"(packets) across the protocol while there is one input for single-state programs. As explained in Section II, the function $\_\_AFL\_INIT()$ is used to mark the forking point in the TP so that the forkserver will always clone itself at that position. In practice, state machines of protocols are implemented in a while loop, such as in OpenSSL. A simplified SSL client/server model can also be developed for fuzzing [4], where socket communication is transformed to file operations[3]. We unroll the while loop into three states to better demonstrate the instrumentation of $\_\_AFL\_INIT()$ as shown in the shadowed area in TP, Fig. 4, but the ideal and actual implementation of yFuzz are not limited by the number of states a protocol may have. In TP, we also add a variable $TPstate$ to indicate the current protocol state of the TP. The function $\_\_AFL\_INIT()$ in Fig. 4 will be invoked conditionally, when the $TPstate$ matches the target fuzzing state of the fuzzer (which is $fstate$, as will be explained in Section III-D). While the forkserver is only initialized once in AFL, yFuzz conditionally initializes the forkserver multiple times for different fork points in TP.

In addition to the forking points, the communications between the TP and the fuzzer are also injected to the TP to share code coverage information (by default in AFL) and the protocol state information through shared memory.

C. State-aware fuzzer

The fuzzer passes forking and fuzzing state information to forkserver, based on the execution status of TP. It collects protocol state and code coverage information from TP after each execution, and in turn, analyze such information to decide the forking state of forkserver and TP in the next execution. These decisions are done by the **state engine** in the fuzzer. At the core of the state engine is the data structure $multiQ$ and the methods operate on it: $constructQ$, $storeQ$, $destroyQ$, $switchQ$. Each $multiQ$ struct will store the queue entries with the same type (basically a linked list) as well as the global variables associated with them for logging and analysis.

The reason for designing the multiple fuzzing queues is that, packets in various stages typically have different formats. It is obviously inefficient to uniformly mutate these types of packets to generate new testcases for whatever state the TP has. And simply putting all packets into one queue will definitely disrupt the analyses that are only meant for one queue. For general programs, one queue will suffice as what is done in AFL, because it only needs to consider a single state of the TP. All the inputs denoted by the queue entries (no matter what content they contain), will be read by the TP at exactly the same location during the execution, which is not the case for protocols as mentioned in Section II.

After each execution of the TP, the fuzzer analyzes the protocol state, TP exit status as well as code coverage, as denoted by arrow ⑤ and ⑥ in Fig. 4. In AFL, there is a 64kB shared memory between fuzzer and TP to track the code coverage information. yFuzz also shares the protocol states between them (will be explained in III-E). The protocol state is updated per execution of TP and once the fuzzer detects new states (or decides to move to the next/previous state), it will invoke Q-related methods to store/destroy current fuzzing Q, and switch to the new Q, as denoted by arrow ①. Meanwhile, it sends the state information to forkserver (as denoted by arrow ②).

The state engine is able to utilize flexible policies for progression (moving to the next state) and regression (rolling back to the previous state) based on the specifications of the target protocol or the user's requirement. The heuristics are explained later in Section IV.

[3]In general, tools such as *preeny* [36] can be used to convert socket communications into file operations through preloading customized libraries, if the source code of TP is not available

Algorithm 2: Workflow of yFuzz : **FS:** forkserver, **TP:** testing program

```
1  Data: multiQ[];
2  Initialization: Qid = 0 (or specified by user);
3  while fuzzing not stopped do
4      Fuzzer: current_entry =
         multiQ[Qid]→current_entry;
5      while fuzzing current_entry do
6          Fuzzer: testcase = mutate(current_entry);
7          Fuzzer: send_to_FS(Qid, forking signal);
8          FS: receive(Qid, forking signal);
9          FS: fork/progress/regress according to Qid;
10         if FS forked then
11             TP: execute and send_to_FS(exit status);
12         else if FS progressed then
13             FS: executes to the next forking point;
14             FS: fork and wait;
15             TP: fork at next point;
16         else /*FS regressed*/
17             TP: exit;
18             FS: go back to last forking point;
19         Fuzzer: collect execution information;
20         Fuzzer: if testcase is interesting then
21             add_to_Q(Qid);
22         Fuzzer: if new protocol state occurs then
23             Qid+=1;
24             proceed_fuzzing();
25             break;
26         Fuzzer: if give up current fuzzing state then
27             Qid-=1 ;
28             regress_fuzzing();
29             break;
30     Fuzzer: current_entry = current_entry→next;
31     if current_entry is NULL then
32         multiQ[Qid]→current_entry =
             multiQ[Qid]→head;
33         fuzzing_cycle[Qid] += 1;
```

D. Multi-state forkserver

The multi-state forkserver is not only the parent of all cloned TPs (that actually execute the mutated testcases), but also a switch that shifts to the right state when receiving commands from the fuzzer (in order to fork TPs with different protocol states). The forkserver in AFL can only be initialized once, i.e., the function $\_\_AFL\_INIT()$ is called only once, when the first $\_\_AFL\_INIT()$ statement is entered. It works well if we only want to fuzz one packet memorylessly regardless of the previous and following packets. To fuzz later packets efficiently (by utilizing early stage packet states), we need to "remember" the execution state of previous packets generation/processing, and need to switch to new fuzzing states without restarting the TP (which will call $execve()$ repeatedly). The forkserver needs to know the current fuzzing state, the next fuzzing state (when progressing) and possibly previous fuzzing state (when regressing).

The forkserver accomplishes this by comparing the protocol state information (received from fuzzer) against the status of $\_\_AFL\_INIT()$ in the current forkserver process. In particular, we use $pstate$ to indicate the current fuzzing packet and $fstate$ to indicate the status of current forkserver. Initially, if the user starts yFuzz to mutate the first packet, then the value of $pstate$ and $fstate$ are both 1. The forkserver with $fstate = x + 1$ are forked by the forkserver with $fstate = x$, where x is in range $[fstate\_min, fstate\_max - 1]$. In yFuzz , we set $fstate\_min$ to 1, and $fstate\_max$ will be updated by the maximum number of forkservers seen so far.

In Fig. 5, to better illustrate both progression and regression, we start from forkserver FS_2 (because FS_1 cannot regress), which means we are currently mutating $p2$ ($pstate$ is 2) when $fstate$ is also 2. Once the fuzzer has decided the next fuzzing state, it will pass the new value of $pstate$ to FS_2 (will be explained in Section III-E), denoted by $pstate'$. Then FS_2 compares $pstate'$ with $fstate$ and take one of the following actions.

- **Staying**: If they are equal, which means the fuzzer wants to continue mutating $p2$, then FS_2 will get the forking signal from the fuzzer to clone a new process (TP), to run with the newly generated testcase, as shown in box 2. The FS_2 will get the return status of TP and send that information to the fuzzer.
- **Progressing**: If $pstate > fstate$, which means that the fuzzer wants to move forward and mutate the next packet $p3$, the FS_2 will fork immediately to generate FS_3. Then FS_3 will resume execution, until the point that $\_\_AFL\_INIT()$ is called again in the TP as shown in Fig. 4 (the line before $read\_p3()$). The condition for entering this $\_\_AFL\_INIT()$ is that the current protocol state $TPstate$ is equal to $fstate$. After FS_3 is generated (by cloning FS_2), the $fstate$ in FS_3 is incremented by 1 (set to 3), such that when FS_3 enters the while loop in $\_\_AFL\_INIT()$ and performs the same check as FS_2 just did, it will find that $fstate == pstate == 3$. Hence, FS_3 will take over the forking task from FS_2. By listening to the fuzzer, FS_3 will keep generating the TP and perform the actual executions. Meanwhile, the FS_2 will be put on hold, waiting for FS_3 to finish and regress (decided by the fuzzer). The FS_2 also reduces the value of $pstate$, setting it to 2, such that when FS_3 regress to FS_2, FS_2 can start normal forking immediately (when $pstate == 2 == fstate$). Note that the $pstate$ changed by FS_2 is in FS_2's address space after forking, this change will not affect the value of $pstate$ in FS_3's address space (which is 3).
- **Regressing**: If $pstate < fstate$, which means that the fuzzer has decided to roll back to the previous state (mutating $p1$). Then FS_2 will exit and give control to the state that generated it, which is FS_1. Recall the status of forkserver that is being held as the FS_2 in the previous condition, FS_1 had been held when generating FS_2. When FS_1 knows that FS_2 exits, it will take back the control to fork when the fuzzer commands so.

The above decision procedure repeats itself for each $pstate$ received from the fuzzer to achieve a continuous multi-state forking and fuzzing.

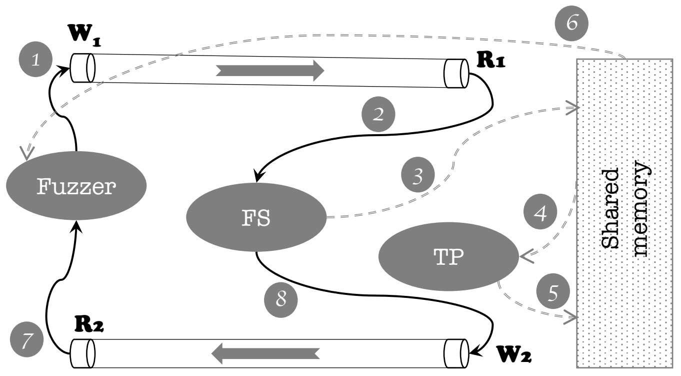

Fig. 6. Communication among fuzzer, forkserver and TP in yFuzz . The solid arrows denotes messages passed by pipes and dashed arrows denotes values shared through shared memory

E. Communication and Coordination

AFL already has communications between the fuzzer and forkserver through pipes and shared memory. In yFuzz , we utilize the existing infrastructure to construct our state-aware communication. As shown in Fig. 6, there are two pipes used to build the communication channels between the fuzzer and forkserver. The fuzzer gets the write end of the top pipe and the read end of the bottom pipe, while the forkserver has read end of top pipe and write end of bottom pipe. In yFuzz , the pipe ends will be duplicated when the forkserver forks, i.e., all fork servers will share the same pipe ends and are able to read and write through them. However, since there is only one active forkserver at a time, there will not be race conditions for different forkservers.

When yFuzz starts to work, the fuzzer will send protocol state information ($pstate$) to the forkserver through W_1 as indicated by arrow ①. The forkserver reads $pstate$ from R_1 and compares it against the value of $fstate$ that it currently has, denoted by arrow ②. Meanwhile, it will also write $pstate$ to the shared memory to inform the TP (as denoted by arrow ③ and ④). After proper state transition as explained in III-D, the forkserver will read the forking signal from the fuzzer through R_1, then it forks the TP at the right position and TP will execute. The forkserver also sends the process ID of TP to the fuzzer through W_2. After execution, TP writes the value of $TPstate$ (as well as code coverage information) to the shared memory to inform the fuzzer (as denoted by arrow ⑤ and ⑥). The forkserver will get the exit status of TP and write it to pipe through W_2 and the fuzzer will read it from R_2. At last, the fuzzer will analyze the execution status of TP based on code coverage, TP exit status and TP protocol state $TPstate$, to decide the next fuzzing state (which is one of stay, progression or regression). Then the fuzzer will starts over by repeating the procedure described above.

The reason of using pipes to send and receive $pstate$ is that a race condition does exist if using shared memory to do so. Suppose the fuzzer decides to move from $pstate = 1$ to $pstate = 2$, so it writes the $pstate = 2$ to the shared memory, then it signals the forkserver to fork, which means that the forkserver only need the signal to fork but does not need any signal to read $pstate$ from shared memory. Since the forkserver keeps looping in $\_\_AFL\_INIT()$, it may already read the old value of $pstate$ from the shared memory even before the

Algorithm 3: Conditions of reading the $pstate$ from the pipe of multi-state forkserver in yFuzz . Other operations such as forking and state transitions are intentionally neglected.

```
1  $looped$ = 0;
2  while $TRUE$ do
3      if ($looped$ || !$fs\_init$) && !$w\_regress$ then
4          read $pstate$;
5          ...;
6          $fs\_init$ = 1;
7      $looped$ = 1;
8      if $fstate == pstate$/*stay*/;
9       then
10         $w\_regress = 0$;
11         ...;
12     else if $fstate < pstate$/*progression*/;
13      then
14         ...;
15         $w\_regress = 1$;
16     else
17         /*regression*/ ...;
```

fuzzer updates it. A $readbeforewrite$ race condition exists, which causes the forkserver to stay in the current state even though the fuzzer wants it to proceed.

During the communication using pipes, yFuzz performs another checking to avoid double reads. Recall that the fuzzer writes two messages in arrow ①: the $pstate$ and the forking signal. The current forkserver will always read $pstate$ first to decide the next state. If the forkserver jumps from FS_2 to FS_3, then FS_3 should not read $pstate$ again since the $pstate$ message is consumed by FS_2 and *does not* exist in the pipe any more.In fact, the next message in pipe should be the forking signal. Interestingly, when FS_3 performs further forking, it has to read $pstate$ from the pipe to perform (potential) state transition. Hence, there exist two different reading operations for FS_3. On the other side, when FS_3 at some point perform state regression and goes back to FS_2, the $pstate$ would have been already consumed by FS_3. The resumed FS_2 is not allowed to try to read $pstate$ again, while FS_2 has to read $pstate$ in normal fuzzing. In general, double reads exist when newly generated/resumed forkserver tries to read $pstate$, which is consumed by the parent/child forkserver. yFuzz uses three additional variables to ensure the correct read behavior: fs_init, $looped$ and $w\_regress$. And the meaning of these variables are as follows. a) fs_init is a global variable that is initialized to 0 and only be set once, when the function $\_\_AFL\_INIT()$ is called at the first time. This is used to identify the first time the forkserver is initialized because the first occurrence of $\_\_AFL\_INIT()$ should read the $pstate$. b) $looped$ is a local variable in function $\_\_AFL\_INIT()$. It is initialized to 0 outside the infinite $while$ loop (where repeated fork() takes place). Inside the loop, $looped$ will be set to 1. This is used to distinguish the first round of newly generated forkserver. As mentioned previously, the first round of newly generated forkserver should not read $pstate$ while in the rest of loop rounds it should. c) $w\_regress$ is used to mark the forkserver that is held and waiting for regression of its child forkserver, such that when it resumes forking, it will not read $pstate$. $w\_regress$ is assigned to 0 in the $staying$ branch of forkserver and assigned to 1 in the $regressing$ branch as mentioned in Section III-D. Thus, the condition for reading the $pstate$ value is as shown in Algorithm 3.

In the example of OpenSSL, suppose we are mutating the first packet $p1$ in Fig. 3, with $pstate = 1$ and $fstate = 1$ (FS_1). And fs_init is set to 1 because $\_\_AFL\_INIT()$ is called. At first the mutated $p1$ has wrong packet format and will not pass the format checking and the server sends back $p2$ to stop the handshake. At some point, the mutated $p1'$ has the correct format and triggers new state in the TP ($TPstate$ changes[4]). In this case, when TP finishes current execution and the fuzzer gets the updated $TPstate$, it decides to mutate $p2$ using this interesting $p1'$ for the next execution ($p1'$ can be retrieved from the previous fuzzing by the fuzzer). So the fuzzer sets $pstate$ to 2 and passes it to FS_1. FS_1 reads and clears the message $pstate$, finds that new $pstate(2)$ is larger than $fstate(1)$, so it forks FS_2. FS_2 will call $\_\_AFL\_INIT()$, and $fstate$ becomes 2. Now the variable $looped$ is 0 and fs_init is 1, so the condition for reading $pstate$ does not hold. FS_2 will **not** try to read $pstate$. The variable $looped$ is change to 1 immediately after this. So when the first round of $while$ loop finishes, the condition of reading $pstate$ holds for FS_2 and FS_2 can read the $pstate$ to perform new state transitions and $fork()$s. The FS_1 is blocked by waiting FS_2, and FS_1 sets its $w\_regress$ value to 1. Later, the fuzzer decides to go back to FS_1 ($pstate = 1$), so FS_2 reads $pstate$, compares it with $fstate$ in FS_2 (which is 2). FS_2 decides to exit which activates FS_1. Since in the process of FS_1, the value of $w\_regress$ has been set to 0, when it resumes and start a new round of $while$ loop, the condition for reading $pstate$ does not hold, preventing FS_1 from reading the non-existing $pstate$. When FS_1 enters the $stay$ branch, the $w\_regress$ is assigned to 0 such that in next round of $while$ loop, FS_1 can read the $pstate$ again.

In summary, yFuzz performs program fuzzing flexibly at multiple execution points with "memory" of precedent program states, by incorporating a state-aware fuzzer, a multi-state forkserver and a stateful TP into a closed loop. The fuzzer analyzes the information provided by the forkserver and TP to decided fuzzing state. The forkserver carries out the state transition for TP. The policies of state transitions (i.e., when to $stay$, $progress$ or $regress$) will be discussed in Section IV and evaluated in Section V.

IV. Implementation

We implement yFuzz based on AFL to utilize its coverage-guided testcase generation and the infrastructure of communication. Our code consists 3234 different lines of C code compared with default AFL, together with 601 lines of Python code for automated testing and analyzing. The code of the state-aware fuzzer, multi-state forkserver and the instructions to instrument the TP are ready to be released. We select some core components/functions from yFuzz and show them in Fig. 7 with the components unique to yFuzz being shaded.

[4]Currently we use the number of packet flights seen in each execution as $TPstate$. However, there could be more options for specific protocls and fuzzing purposes, such as execution time, packet size ranges or specific actions that the TP triggers.

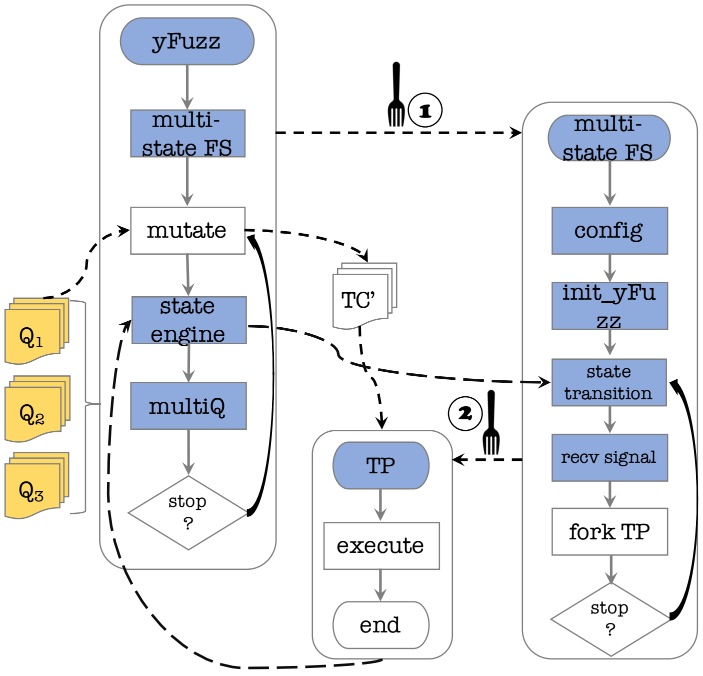

Fig. 7. Implementation of yFuzz based on AFL

The $multiQ$ is composed of the data structure $multiQ$ and methods such as $constructQ$, $storeQ$, $destroyQ$, in-memory $resumeQ$ and $switchQ$, as well as the handling of file descriptors, path constructions and cleanups. The core function of $stateengine$ is $has\_new\_state()$, where the information from forkserver and TP is analyzed to decided next fuzzing state. Once the decision is made, then $progression/regression$ is called to change fuzzing state, or none of them is called and the fuzzer will continue working on current fuzzing state.

On the forkserver side, yFuzz instruments the LLVM pass used by $afl\text{-}clang\text{-}fast$ to implement multi-state forkserver. The function $init\_yFuzz$ is used to initialize some global variables, such as those used to determine the $read$ condition (as explained in Section III-E). The state transition implements the three branches ($staying$, $progression$ and $regression$) mentioned in Section III-D. And the $recvsignal$ components stands for the logic of reading from/writing to the pipes and shared memory.

The TP is also instrumented as follows. 1) Multiple program locations are selected and set as the forking points for the multi-state forkserver initialization. 2) In addition to the code coverage and exit status, the TP will share more information (such as $TPstate$ and $pstate$) with the state engine in the fuzzer. 3) The fuzzer and TP also communicate to record the packets when progressions are performed, such that yFuzz can keep track of the chain of packets that lead to vulnerabilities.

Search Policy: Based on the structure of the multi-state forkserver, yFuzz implements a DFS-like searching policy to transit among different fuzzing states. Taking the example in Fig. 3 as an example, when interesting $p1'$ occurs, yFuzz will use the program state of $p1'$ and starts to fuzz $p2$. If interesting $p2'$ is generated, then yFuzz will follow the program state of $p1'$ and $p2'$ to fuzz $p3$, and so on. During any state in-between

$p1$ and $p4$, if no interesting case is generated, then yFuzz will regress to previous fuzzing state ($p4 \rightarrow p3$, $p3 \rightarrow p2$ or $p2 \rightarrow p1$). When yFuzz comes back at $p1$, then it continues fuzzing $p1$ and wait for the next progression.

The conditions of progression and regression define the power schedule of yFuzz . yFuzz will perform progression to move the fuzzing state forward when $TPstate$ satisfies certain conditions (such as the increase of the number of packets occurred during the current execution). In particular, the increase of number of packets indicates that the packet currently being fuzzed has triggered a new protocol state, as well as new code coverage. However, such condition will potentially prevent progression from happening when $TPstate$ already reaches its maximum value and cannot increase any more. In Fig. 3, suppose yFuzz is fuzzing $p1$, the initial seed of $p1$ might not be valid and the number of packet flights is 2 ($p1$ and $p2$, where $p2$ terminates the session). After certain amount of mutation, a valid $p1$ is generated ($p1'$) and $TPstate$ reaches 4. yFuzz will start to fuzz $p2$ based on the program state of $p1'$. At this point the $TPstate$ will not exceed 4 any more, which means progression will not be triggered to fuzz $p3$ and $p4$. To solve this problem, in addition to the $TPstate$ monitoring, yFuzz also adopts a profile-based progression policy. In particular, the fuzzing process is separated into two stages: *profiling* and *testing*. During the profiling stage, each packet is fuzzed for a fixed amount of time (say, one hour) to provide an overview of code coverage and fuzzing queue related to each packet. After profiling, the probability of progression at each state is decided. Intuitively, the fuzzing state that has higher code coverage and more pending queue entries will be assigned more fuzzing time, and the probability of progressing to this fuzzing state is assigned a larger value. In the testing stage, yFuzz perform random progression based on the probabilities determined during profiling. Periodically, yFuzz updates the probabilities by jointly consider the code coverage (and queue entries) in the profiling and testing stages. yFuzz also assign a higher score to the packets that trigger new protocol states, giving more mutation time to these packets.

A similar mechanism is applied to regression, i.e., the fuzzing state with higher code coverage and more pending queue entries will have lower probability of regressing to previous fuzzing state. Also, yFuzz sets other thresholds for regression such as max_Q_cycles and $max\_entries$. When the current fuzzing state finishes max_Q_cycles (as depicted in Algorithm 2 line 33) or the index of current queue entry exceeds $max\_entries$ (Algorithm 2 line 30), yFuzz will enforce regression to prevent wasting too much resources upon current fuzzing state.

Note that we set the search policy in yFuzz heuristically. In fact, the progression and regression conditions can be easily changed to adapt to different protocols.

V. Evaluation

In this section, we evaluate yFuzz to answer the following questions: (i) What is the performance of yFuzz with respect to metrics such as code coverage and number of unique crashes? (ii) How does it compare with non-stateful fuzzing like default AFL? (iii) What is the runtime overhead of yFuzz due to state forking and replay? (iv) What are the benefits of yield-driven fuzzing strategy?

TABLE I. Statistics of fuzzing single packet (OpenSSL v101) at four different stages using default AFL for 6 and 24 hours.

	Code Coverage(%)	Unique Crashes	Cycles Done	Total # of Executions(M)	Time (hours)
p1	9.51	1	4	7.87	6
p2	10.18	9	0	12.68	6
p3	5.56	9	15	12.21	6
p4	2.61	6	157	12.43	6

	Code Coverage(%)	Unique Crashes	Cycles Done	Total # of Executions(M)	Time (hours)
p1	9.64	11	30	42.05	24
p2	11.16	9	6	49.58	24
p3	5.6	14	410	66.20	24
p4	2.61	9	1308	54.80	24

A. Environment Setup

All experiments are done on a ubuntu server (16.04.5 LTS) with 48 cores (Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz) and 92 GB RAM. Each fuzzer runs on a single core in the same environment. We choose OpenSSL (with version 101, 110) as our benchmark. As mentioned in [4], we first compile OpenSSL using $afl\text{-}clang\text{-}fast$ to generate the static library $libssl.a$ and $libcrypto.a$. Then in our instrumented TP as shown in Fig. 10, we invoke functions from $libssl.a$ and $libcrypto.a$ to perform the ssl handshake. We add $init\_yfuzz()$ after each packet is generated, under the condition "$pstate == TPstate$". In TP, we also get the shared memory pointer through the environment variable "$\_\_AFL\_SHM\_ID$", which is created by the fuzzer and shared among its children processes. We use the number of packets occurred in the execution as the value of $TPstate$. Hence, each time $TPstate$ changes, we will consider a state change in the protocol. We utilize AFL's built-in support for ASAN [14] to consider more crash conditions.

We conduct each batch of experiment that with the same parameters (w.r.t OpenSSL version, fuzzer settings, fuzzing time) four times and show the average numbers where it applies.

B. Effect of single-packet fuzzing

TABLE II. Code coverage breakdown: the code explored by fuzzing four individual packet. Time is in hours. The total size of bitmap is 64kB (65536 Bytes). Ui stands for the number of edges that are only explored when fuzzing packet i but not explored when fuzzing other packets (i.e., edges that is unique to packet i).

Version	Time	Covered	Uncovered	U1	U2	U3	U4
101	6	7677	57859	563	955	30	386
	10	8879	56657	373	962	29	360
	24	8896	56640	312	966	32	359
110	6	10721	54815	1472	755	26	296
	10	10093	55443	2123	81	15	293
	24	11272	54264	2054	738	22	295

We evaluate the performance (in terms of code coverage and unique crashes) of fuzzing single packet during OpenSSL handshake to demonstrate the limitations of default non-stateful fuzzing. The TP is constructed by following the idea in [4] as listed in Fig. 10. By assigning different values to $packetID$ in line 11, we can utilize the default forkserver in AFL to conveniently fuzz different packets.

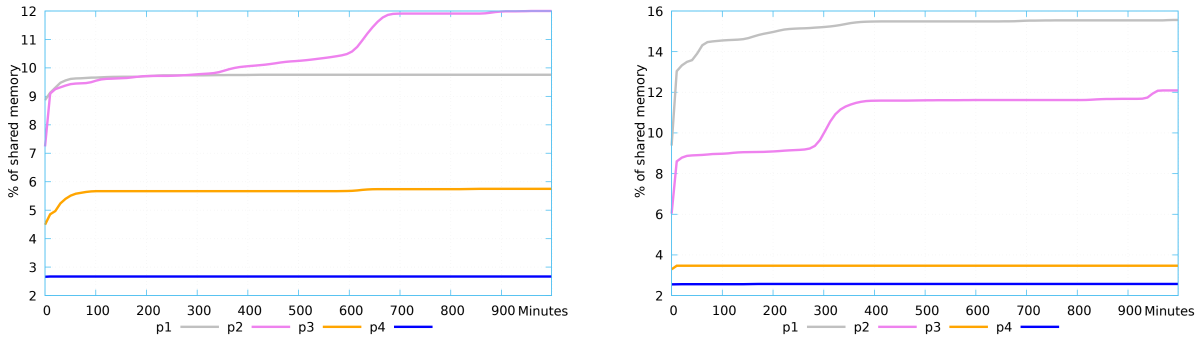

Fig. 8. Code coverage trend of fuzzing single packet at four different stages with OpenSSL 101 and 110.

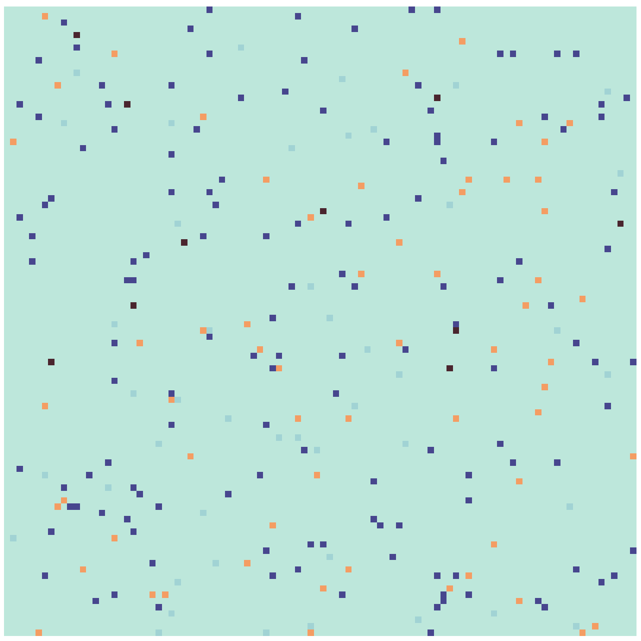
Fig. 9. Code coverage composition of fuzzing individual packet on OpenSSL v110. The 64kB shared memory is converted to a 256*256 map (only 100*100 is shown here because of the size limit), each cell representing an unique edge. Difference color represents execution edges that are discovered by fuzzing different packets.

```
/*libssl.a and libcrypto.a are statically compiled by afl-clang-fast, with ASAN enabled*/
#include <openssl/ssl.h>
#include <openssl/crypto.h>
...
int main(void)
{
/*set up client and server, including SSL BIO, session, server certificate and private key*/
  ...
/*initialize the counter for packet flights, and which packet to fuzz*/
  packetCount = ... ;
  packetID = ... ;
/*perform handshake*/
  do{
    /*client performs handshake once*/
    SSL_do_handshake(client);
    packetCount++;
    if(packetCount == packetID){
      __AFL_INIT();
      /*replace client packet with AFL-mutated input*/
    }
    /*pass client packet to server*/
    ...
    /*server performs handshake once*/
    SSL_do_handshake(server);
    packetCount++;
    if(packetCount == packetID){
      __AFL_INIT();
      /*replace client packet with AFL-mutated input*/
    }
  }
  while(/*handshake not finished*/)
  ...
}
```

Fig. 10. TP for single-packet fuzzing

In the case of OpenSSL version 101, fuzzing different packet results in different code coverage as shown in Fig. 8. Fuzzing the first and second packet typically can yield more code coverage than fuzzing the third and fourth packet. The average numbers are shown in Table I. In particular, fuzzing the first and second packet during OpenSSL handshake for 6 hours can achieve 9.51% and 10.18% code coverage, respectively. However, the third and fourth packet fuzzing can only reach 5.56% and 2.61% code since the code space for them to explore is greatly reduced when starting from the late stage of handshake. Correspondingly, the completed fuzzing queue cycle of later stage fuzzing ($p3$ and $p4$) are much larger than early stage fuzzing ($p1$ and $p2$), which means that AFL cannot find interesting testcases anymore, so the length of queue is much less and it will finish one round of fuzzing quickly then start the next cycle. When the experiments are conducted for 24 hours, the code coverage results are similar. This indicates that the growth of code coverage when fuzzing single packet is extremely slow after 6 hours or less.

Among the different code coverage explored by fuzzing different packets, some are common code (edges) and others may be unique to each packet. We want to find out the composition of the code coverage by fuzzing individual packets. However, the default AFL assign ID to basic blocks randomly during runtime. If we restart the program to fuzz $p2$ after fuzzing $p1$, then the assignment of block IDs will be different, which means that the same edge could appear in different position of the bitmap. Hence, we fuzz the four different packets in one run, each for 6 (10,24) hours. When the current packet fuzzing lasts for 6 (10,24) hours, we force the progression to fuzz the next packet, by clearing the code

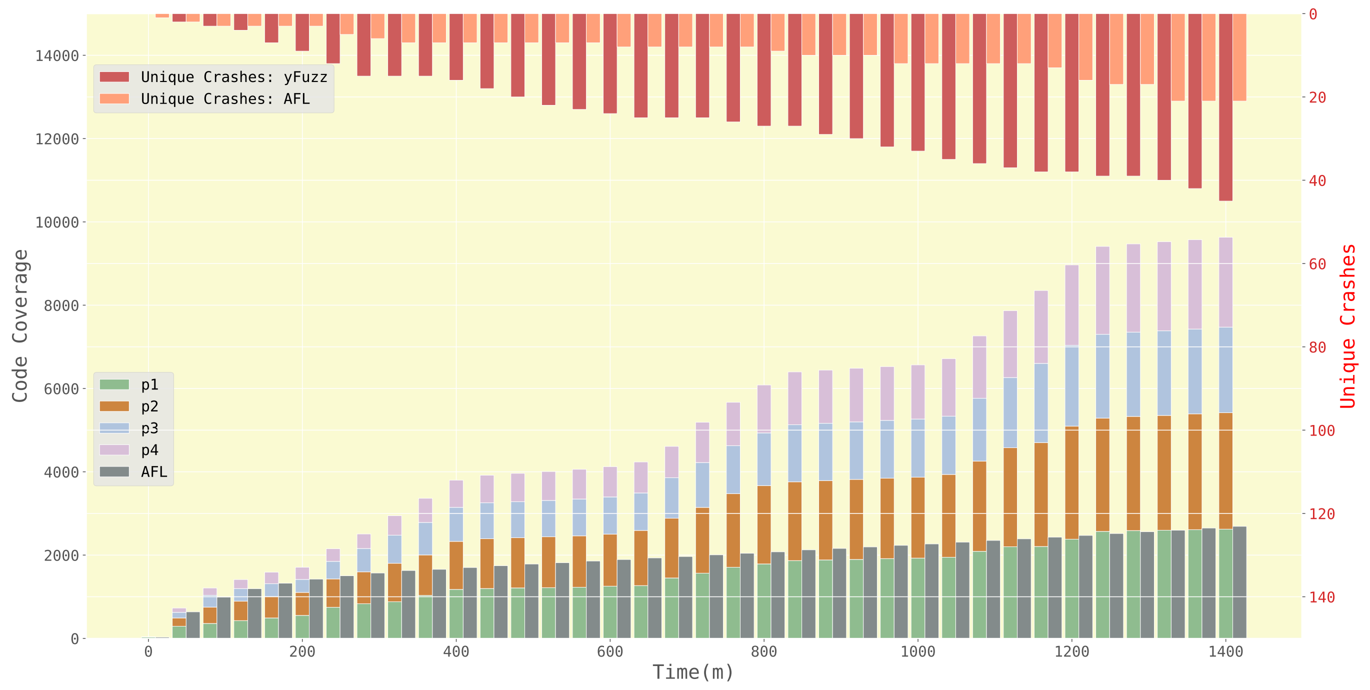

Fig. 11. AFL vs yFuzz : bottom bars show the composition of code coverage and top bars show the number of unique crashes. Note that we subtract the initial code coverage (7.9% in this case) explored by the seed $p1$ from each bottom bar to show the increments of code coverage. And the code coverage is represented as the number of discovered edges.

coverage bitmap (the global variable $virgin\_bits$ in AFL) without relaunching the AFL. The experiment results are shown in Table II. We can see that the total code coverage of 24 hours' fuzzing (for each packet, the fuzzing time is 6 hours) is 7677/64kB = 11.71%, which is higher than any of the four single-packet fuzzing shown in Table I, due to the unique code coverage. Further, we analyze the bitmap (which is used to store the code coverage information in AFL), and get the unique edges explored by each packet, as shown in Table II column $U1$, $U2$, $U3$ and $U4$. Fig. 9 also visualizes the unique edges corresponding to each packet.

In summary, the experiments conducted in this section has shown that:

- By only fuzzing one packet, the code coverage is limited. Fuzzing early-stage packets results in higher code coverage.
- Different packet fuzzing can discover unique code. Which is to say, even though late-stage packet fuzzing achieves less code coverage, it still discovers the code that cannot be discovered by early-stage packet fuzzing. (And early-stage packet fuzzing also discovers unique code that cannot be explored by late-stage packet fuzzing).

The two conclusions above has proven the need of stateful fuzzing, which is to fuzz different packets interactively and heuristically.

C. yFuzz : progression and regression

After the profiling stage (as mentioned in Section IV), yFuzz starts to perform progression and regression based on the protocol state changes and the probability (based on code coverage and fuzzing queue during profiling stage). In the case of AFL, fuzzing $p1$ or $p2$ results in better code coverage and unique crashes as shown in Table I. In addition, AFL tends to stop discovering new code soon after a short amount of time when there is no interesting testcases. yFuzz is able to "escape" the fuzzing stages that are no longer profitable and flexibly switch between different states to discover new code. We compare a typical run of yFuzz and AFL in Fig. 11, as well as the code coverage breakdown of yFuzz for each fuzzing stage. The left $y-axis$ marks the absolute number of edges (defined as the execution pair of basic blocks as mentioned in Section II) explored by the fuzzers and the right $y-axis$ marks the number of unique crashes. We show the code coverage discovered by yFuzz in four different stages: $p1$, $p2$, $p3$ and $p4$. All the four fuzzing stages of yFuzz grow considerably along the timeline while the default AFL (the grey bar) yield slow growth. As a result, yFuzz finds almost double amount of unique crashes compared with AFL.

On average of four 24-hour fuzzing, yFuzz is able to discover 19.27% code (of a total size of 64kB shared memory). In particular, fuzzing packet $p1$, $p2$, $p3$ and $p4$ contributes 10%, 4.65%, 1.73% and 2.79% code coverage (of a total size of 64kB shared memory) respectively. In other words, fuzzing $p1$ contributes a percentage of 10/19.27 = 52.41% of the entire discovered code. Similarly, fuzzing $p2$, $p3$ and $p4$ contributes 24.13%, 8.98% and 14.48%. And the air time spent on each fuzzing stage is 7.2, 11, 1.4 and 4.4 hours. In terms of unique crashes, yFuzz founds 43 unique crashes during 24 hours (on average), while AFL found 11 when fuzzing $p1$ (for 24 hours) or 14 when fuzzing $p3$ (for 24 hours).

D. Runtime overhead of state transitions

Even yFuzz outperforms the default non-stateful fuzzing using AFL, we still want to evaluate the runtime overhead of yFuzz . In particular, 1) in the multi-state forkserver, yFuzz has extra code that compares the conditions for progression/regression. 2) and in the fuzzer side, when progression/regression happens, extra code is applied to update the $multiQ$. We want to answer the question that does yFuzz slow down the fuzzing process because of state condition checking and transition.

From Table I, we can derive that when fuzzing single packet $p1$, the average number of TP execution is 42.05/24 = 1.75 (million/hour), when the fuzzing time is 24 hours. On average of 4 times of 24-hour fuzzing, yFuzz finishes 1.74 million TP executions per hour, which indicates a negligible slowdonw of 0.57%. Our explanation is that, 1) the overhead of condition checkings in the multi-state forkserver can be ignored since we use $likely()$/$unlikely()$ macros to help the branch prediction, and the number of progression and regression are extremely minor to the total number of TP executions, so the branch prediction is correct most of the time; 2) the vast majority of the time spent in fuzzing is the generation and execution of invalid testcases as indicated in [20], which greatly downplay the overhead (such as $multiQ$ updates) caused by yFuzz .

VI. Related Work

Program fuzzing has enjoyed success in hunting bugs in real-world programs with researchers devoting tremendous efforts into it.

Code-coverage guided fuzzing: Plenty of works focus on smarter testcase mutation/selction or search heuristics, to help the fuzzer generate inputs that explore more/rare/buggy execution paths [29], [34], [26], [16], [19], [30]. AFLFast [6] models testcase generation as a Markov chain. It changes the testcase power scheduling policy (scoring and priority mechanism) of default AFL, to prevent AFL spending too much time on the high-frequency testcase, and assigning more resource to low-frequency paths. Similarly, AFLGo [5] uses simulated annealing algorithm to assign more mutation time to testcases that are "closer" to the target basic block, to quickly direct the fuzzing towards the target code area. These works help AFL to find the target paths faster by changing the mutation time assigned to each testcase, but cannot find new paths, e.g., new vulnerabilities. yFuzz , on the other hand, can not only optimize the power schedule based on the protocol states, but also can explore new paths that the default AFL could never explore by stateful progressions. CollAFL [11] proposes more accurate code coverage approaches to avoid the collision in AFL, and implements three new search strategies to guide the fuzzer to explore branches that are "untouched" and have more memory accesses.

Symbolic execution and tainting: Techniques such as tainting and symbolic execution are also employed to complement greybox fuzzing [21], [18]. Angora [8] implements byte-level tainting to locate the critical byte sequences (that determines branch control flows) from the input, then use gradient descent algorithm to solve branch condition to explore both branches. SYMFUZZ [7] utilizes tainting and symbolic execution to determine the dependencies between input bytes and program CFG, in order to decide which bytes to mutate (optimal input mutation ratio) during fuzzing. Drill [28] uses concolic execution to solve constraints of magic numbers (to guide fuzzing) then apply fuzzing inside each code compartment (to mitigate path explosion).

Machine Learning: Some works take advantages of machine learning techniques to model/improve the fuzzing [13], [31], [9]. Angora [8] and NEUZZ [27] adopt gradient descent-based searching policies (instead of code-coverage) to guide the input mutation. NEUZZ builds a feedforward neural network to mimic the code coverage behavior of the TP. The neural network is trained by testcases and bitmaps (as ground truth) generated by AFL, to find the critical bytes in testcases. When new testcases are executed, NEUZZ only mutate the critical bytes to reduce redundant testcase generation. SemFuzz [33] combines NLP and semantics-based system call fuzzing to automatically generate exploits based on online bug reports such as CVEs and git logs. LipFuzzer [37] converts voice commands into computational linguistic data (using an adversarial Bayesian networks) then perform NLP-based language template fuzzing to explore the semantic inconsistencies in voice assistant services.

Scalability: Wen Xu et al. [32] aims to improve the scalability of fuzzers (such as AFL and LibFuzzer) on multi-core servers by three techniques: a) using a scalable system call $snapshot()$(instead of $fork()$) to repeatedly execute the TP. b) providing a memory file system (with limited size) to the fuzzer, avoiding time-consuming small file operations. c) sharing the testcase logs in memory among different fuzzer instances.

Program transformation: Another interesting line of work transforms the testing programs for fuzzing [17], [20], [23]. T-Fuzz [23] dynamically traces the testing programs to locate and remove the checks once the fuzzer gets stuck. Untracer [20] creates customized testing programs with software interrupts at the beginning of each basic block. Instead of tracing every testcase for coverage information (as in AFL), Untracer enables the the testing program to signal the fuzzer once new basic blocks are encountered, thus greatly reducing the overhead caused by redundant testcase tracing.

Protocol fuzzing: Few greybox fuzzers are designed specifically for protocols (in general, stateful programs). Sulley [1] and its successor Boofuzz [24] are two popular white-box protocol fuzzers that generate packets based on protocol specifications then send them to target ports for fuzzing. Unlike greybox fuzzers that instrument the testing program for code coverage and tainting information, the whitebox fuzzers typically only instrument to monitor process/network failures. Thus they lack the guidance for smarter testcase generation and power scheduling. Blockbox protocol fuzzers assume no knowledge of the protocols and try to reverse engineer them through packet/program state analyses [15], [3], [12], [10]. While the blackbox fuzzers have wider scope of use than the whitebox fuzzers, they inevitably suffer the incomplete/inaccurate packet reverse-engineering analyses, and both of these two types of protocol fuzzers fail to adopt the useful techniques from the greybox fuzzing. yFuzz , on the other hand, is a state-aware greybox protocol fuzzer that leverages coverage-guidance and stateful protocol fuzzing to efficiently explore deep into each protocol states.

VII. Conclusion

In this paper, we identify the challenges in fuzzing stateful protocols/programs and demonstrate the limitation of existing greybox fuzzers when fuzzing protocols. In order to achieve higher code coverage for protocol fuzzing, we propose a progressive stateful protocol fuzzer, yFuzz , to capture the state changes in protocols, and heuristically explore code spaces that are related to multiple protocol states. We implemented yFuzz upon the popular greybox fuzzer, AFL and evaluate yFuzz on OpenSSL (v101 and v110). Our experimental results show that yFuzz can achieve 1.73X code coverage and 2X unique crashes when comparing to only fuzzing the first packet during the protocol communication (which is adopted by current greybox fuzzer).

Meanwhile, we are also aware that yFuzz could be further improved by combining other techniques as follows, which are actually our on-going works.

Flexible power schedules: Though we adopted several power schedule policies to demonstrate the effect of yFuzz 's multi-state fuzzing, we realize that there could be more options to improve the fuzzing performance for specific protocols. For example, 1) the selection of $TPstate$ is flexible: as mentioned previously, we choose the number of packets seen in one TP execution as the value of $TPstate$. Apart from this, $TPstate$ could also be the size of a particular packet, execution time of the message flight(s), different dynamic state transitions and so on (and arbitrary combinations of them). 2) the conditions for progression and regression are also adjustable: we monitor several variables during the fuzzing (such as $TPstate$, number of fuzzing cycles, number of progressions and regressions) to decide when to progress and regress the fuzzing state. In addition to these, if the user has a particular target of fuzzing, the conditions could be modified in the function $has\_new\_state()$ accordingly. For example, if the user wants to limit the size of the fuzzing queue, then progression or regression could be forced to happen when the size of fuzzing queues reaches their limits.

Partial fuzzing: we could also integrate partial fuzzing (i.e., instead of fuzzing the whole packet, we could only fuzz the interesting packet fields) to yFuzz . In OpenSSL, the independent fields are typically the extension fields in $client\_hello$ and $server\_hello$. By only fuzzing these independent fields (and also modify packet length to make the packet consistent and valid), we could save more time and avoid redundant testcase generations.

Instrumention of TP: By working with source code, the instrumentation of TP shall be sufficiently simple and convenient. However, we realize that yFuzz would be a more general and powerful fuzzer if it can work with blackbox program binaries. Thus, we investigate the techniques such as dynamic tainting (to locate the basic blocks that generate/process packets) and binary rewriting (to carry out the instrumentation on binaries) to make yFuzz functional on program binaries.

References

[1] Pedram Amini and Aaron Portnoy. Sulley fuzzing framework, 2010.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[3] Bernhards Blumbergs and Risto Vaarandi. Bbuzz: A bit-aware fuzzing framework for network protocol systematic reverse engineering and analysis. In *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, pages 707–712. IEEE, 2017.

[4] Hanno Bock. How heartbleed could've been found, 2015.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[7] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[8] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[9] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 95–105. ACM, 2018.

[10] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *24th $USENIX$ Security Symposium 15)*, pages 193–206, 2015.

[11] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

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

[13] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. IEEE Press, 2017.

[14] Google. Addresssanitizer, 2018.

[15] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[16] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 37–47. ACM, 2018.

[17] Ulf Kargén and Nahid Shahmehri. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 782–792. ACM, 2015.

[18] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. Cab-fuzz: Practical concolic testing techniques for {COTS} operating systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 689–701, 2017.

[19] Caroline Lemieux and Koushik Sen. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. *arXiv preprint arXiv:1709.07101*, 2017.

[20] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. *arXiv preprint arXiv:1812.11875*, 2018.

[21] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1475–1482. ACM, 2018.

[22] OpenSSL. Official tesing programs for openssl fuzzing, 2019.

[23] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[24] J Pereyda. boofuzz: Network protocol fuzzing for humans. *Accessed: Feb*, 17, 2017.

[25] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[26] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.

[27] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620*, 2018.

[28] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[29] Robert Swiecki. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. *URl: https://github. com/google/honggfuzz (visited on 06/21/2017).*

[30] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. *arXiv preprint arXiv:1812.01197*, 2018.

[31] Valentin Wüstholz and Maria Christakis. Learning inputs in greybox fuzzing. *arXiv preprint arXiv:1807.07875*, 2018.

[32] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328. ACM, 2017.

[33] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[34] Michal Zalewski. American fuzzy lop, 2014.

[35] Michał Zalewski. Fuzzing random programs without execve(), 2014.

[36] Zardus and Mrsmith0x00. Preeny, 2019.

[37] Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. Life after speech recognition: Fuzzing semantic misinterpretation for voice assistant applications.