Exploring Effective Fuzzing Strategies to Analyze Communication Protocols

ANONYMOUS AUTHOR(S)

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 protocol fuzzer, ***yFuzz***, to explore the code related to different protocol states. yFuzz is built on AFL (a mainstream greybox fuzzer), and 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 a compiler-instrumented protocol program. Our experimental results on OpenSSL show that yFuzz improves the code coverage by 73% and increases the number of identified unique crashes by 100% when comparing against AFL fuzzing the first packet during a protocol handshake.

1 INTRODUCTION

Vulnerabilities in network protocols (such as Heartbleed in OpenSSL [20] and Remote Code Execution in SNMP [12]) 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.

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 details are not known beforehand). 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 [28] for LibFuzzer [33] and AFL [45]. A whitebox protocol fuzzer such as *sulley* [1] and *boofuzz* [30] 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 [2, 16, 19] 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. Blackbox fuzzing can also infer the state machine of protocols and identify flaws through the inferred state machine automatically [14]. 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, greybox fuzzers [4, 5, 21, 25, 33, 37, 39, 45**?**] 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 [26, 29] for efficient greybox fuzzing.

Challenges: Despite recent progress on fuzzing tools, a number of fundamental limitations and challenges still exist for stateful protocol fuzzing. 1) Communication protocols are typically implemented through state machines on servers/clients with state transitions driven by critical protocol events such as packet exchange. Although proxies can be employed to mutate and fuzz packets on the fly [3], 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. 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. Protocol fuzzers need to identify the dependence and adapt its fuzzing strategies accordingly. 3) Finally, protocol implementations need to be properly instrumented/transformed in order to be directly fuzzed by popular tools such as AFL [45] and VUzzer [32]. For instance, socket communications need to be logged and converted to direct data operations for higher efficiency. We explain in detail the inefficiency of stateless and individual-packet fuzzing in Section 2.2.

Our approach: 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 protocol execution states, (ii) intelligently switch 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 is built on an industry-level greybox fuzzer-AFL. yFuzz consists of 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 decide the next fuzzing state. The state-aware fuzzer, multi-state forkserver and instrumented TP work in concert to identify testcases that change the protocol states, change fuzzing states, and explore the program execution space efficiently. The fuzzing state may move forward and backward to identify the sweet spot for highest fuzzing yield, until the fuzzer can no longer find interesting testcases. We implement a prototype of yFuzz (as an open-source tool available at [44]). Evaluations using real-world protocols like the OpenSSL library show that yFuzz can achieve significantly improvement in terms of better code coverage and ability to find unique crashes.

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 demonstrates how 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 can improve the code coverage by 73% within same period of time (24 hours) and discover 2 times of unique crashes compared with the default AFL.

2 BACKGROUND

2.1 Overview of AFL

AFL [45] is a popular coverage-guided greybox fuzzer. It maintains a queue 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 rounds of mutations 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 the 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 program execution. It does so by recording basic block transitions in the TP. A basic block is a sequence of binary code that only has one entry, one exit and no branch within the block. Each basic block of the 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 the TP). For each execution, the 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.

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., repeating the library initialization and dynamic linking. Such repeated program initialization is unnecessary and could occupy a large ratio of the total execution time. Hence, AFL designs a forkserver to call $execve()$ only once. The forkserver logic is shown in Fig. 1. The fuzzer will launch after some configurations and 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 the TP by users at desired positions beforehand)[2]. In $\_\_AFL\_INIT()$, the TP will enter an infinity $while$ loop. Every time the fuzzer finishes one mutation of the input and generates a new testcase, it will send a signal to the forkserver, informing 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 the TP and sends it to the fuzzer. In Fig. 1, 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 a 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

[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.

[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. The behavior of default $\_\_AFL\_INIT()$ is shown in Algorithm 2 in Appendix

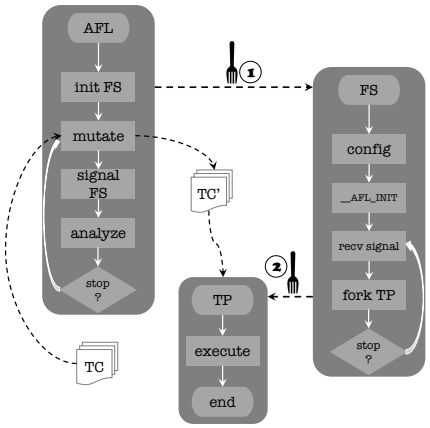

Fig. 1. Simplified AFL forking workflow. FS: forkserver, TC/TC': testcase, TP: testing program

is already done. Note that if the function *__AFL_INIT*() is not specified by the user, AFL will automatically put it right before the *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.

2.2 Motivating example

Though the forkserver mechanism "speeds up the fuzzing of many common image libraries by a factor of two or more" [46], 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 maintains 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 packets yield similar code coverages at around 10% (using the same amount of time), fuzzing the third and fourth packet suffers 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 simply fuzzing any single packet will not be able to explore all the code (as shown in Fig. 7). The details will be explained in Section 5.

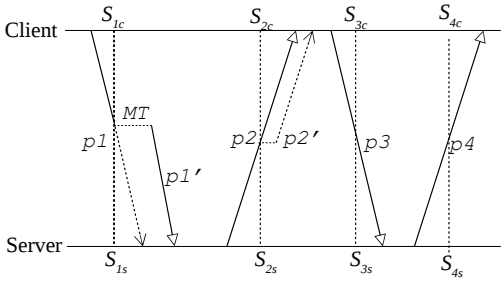

Fig. 2. Forking example of a four-stage TLS handshake

The forkserver framework in AFL cannot handle multi-state protocols. As shown in Fig. 2, there will be typically four packet flights between a TLS client and server 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*) +$finished$ and $p4$: *change_cipher_spec* + $finished$. 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 takes 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 $p1'$ needs 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 TLS). When an interesting $p1'$ occurs, it is bound with the current state of client S_{1c} as shown in Fig. 2. 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.

In yFuzz, we implement a 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 the TP to work on the same state and fuzz the target packet. The code coverage mechanism in yFuzz is similar to that in AFL.

3 SYSTEM DESIGN

3.1 Overview

yFuzz achieves stateful protocol fuzzing by combining a state-aware fuzzer(Section 3.3), a multi-state forkserver (Section 3.4) and an instrumented TP(Section 3.2) as shown in Fig. 3. 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. 2, 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 the 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 3.4). When the forkserver is ready, it will keep listening to the fuzzer, waiting for signals to fork and generate a new process of the TP. 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. (A summary of yFuzz workflow is also described in Appendix Algorithm 3 due to space limit.)

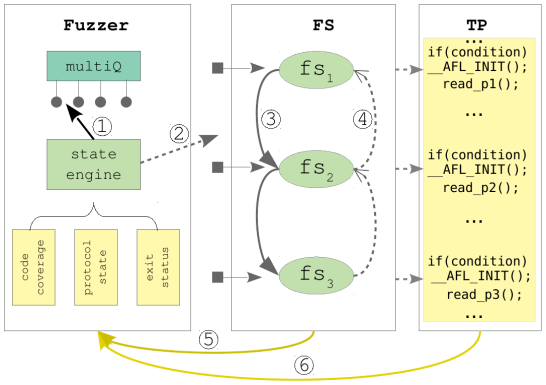

Fig. 3. System Overview of yFuzz. FS: forkserver, multiQ: queues for storing different types of testcases, TP: testing program

3.2 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 2, 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 TLS client/server model can also be developed for fuzzing [3], 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 the TP, Fig. 3, but the ideal and actual implementation of yFuzz are not limited by the number of states a protocol may have. In the TP, we also add a variable $TPstate$ to indicate the current protocol state of the TP. The function $\_\_AFL\_INIT()$ in Fig. 3 will be invoked conditionally, when the $TPstate$ matches the target fuzzing state of the fuzzer (which is $fstate$, as will be explained in Section 3.4). While the forkserver is only initialized once in AFL, yFuzz conditionally initializes the forkserver multiple times for different fork points in the 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.

3.3 State-aware fuzzer

The fuzzer passes forking and fuzzing state information to forkserver, based on the execution status of the TP. It collects protocol state and code coverage information from the TP after each execution, and in turn, analyze such information to decide the forking state of forkserver and the 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

[3]In general, tools such as *preeny* [47] can be used to convert socket communications into file operations through preloading customized libraries, if the source code of the TP is not available

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 2.

After each execution of the TP, the fuzzer analyzes the protocol state, exit status of the TP as well as code coverage, as denoted by arrow ⑤ and ⑥ in Fig. 3. In AFL, there is a 64kB shared memory between the fuzzer and the TP to track the code coverage. yFuzz also shares the protocol states between them (as explained in Section 3.5). The protocol state is updated per execution of the TP. 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 queue, and switch to the new queue, as denoted by arrow ①. Meanwhile, it sends the state information to forkserver (as denoted by arrow ②).

The state engine utilizes 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, as explained in Section 4.

3.4 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. 4, 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 3.5), 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 the 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. 3 (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

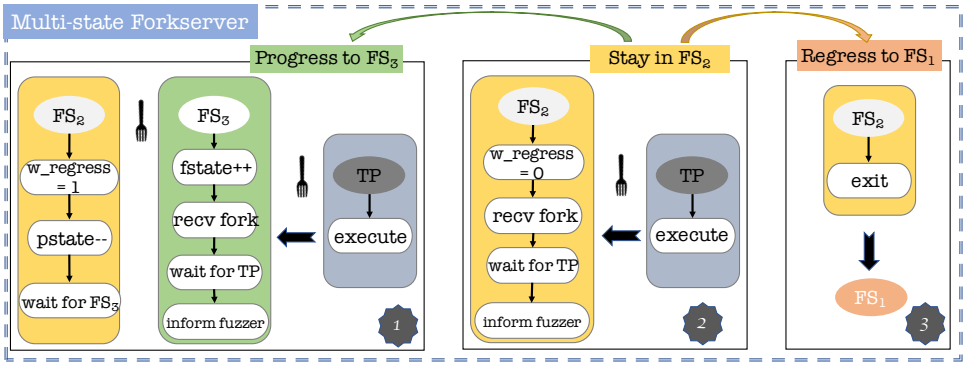

Fig. 4. Multi-state forkserver in yFuzz. Suppose current fuzzing state is FS_2 and the next fuzzing state could be one of the three: staying in FS_2, progressing to FS_3 and regressing to FS_1.

$fstate == pstate == 3$. 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. Note that though Fig. 4 only shows one path of state transition, the actual transition involves multiple paths. The transition has a tree-based structure where each node represents a fuzzing state. Each node could have multiple children. When a *progression* happens, the fuzzing state switches to one of the children of the current node; And when a *regression* happens, the fuzzing state goes back to its parent node.

3.5 Communication and Coordination

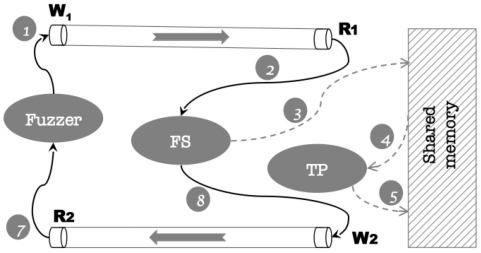

Fig. 5. Communication among the fuzzer, forkserver and TP. R/W in the pipes stand for the packet read/write. The solid arrows denotes messages passed by pipes and dashed arrows denotes values shared through shared memory.

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. 5, 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 Section 3.4, the forkserver will read the forking signal from the fuzzer through R_1, then it forks the TP at the right position and the TP will execute. The forkserver also sends the process ID of the TP to the fuzzer through W_2. After execution, the TP writes the value of *TPstate* (as well as code coverage information) to the shared memory with the fuzzer (as denoted by arrow ⑤). The forkserver will get the exit status of the 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 the TP based on code coverage, the TP exit status and protocol state *TPstate*, to decide the next fuzzing state (which is one of stay, progression or regression). Then the fuzzer will start 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 needs 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 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 3.4. Thus, the condition for reading the *pstate* value is as shown in Algorithm 1.

Algorithm 1: Conditions of reading the *pstate* from the pipe of multi-state forkserver in yFuzz. Other operations such as forking and state transitions are intentionally omitted.

```
looped = 0;
while TRUE do
    if (looped || !fs_init) && !w_regress then
        read pstate;
        ...;
        fs_init = 1;
    looped = 1;
    if fstate == pstate/*stay*/;
     then
        w_regress = 0;
        ...;
    else if fstate < pstate/*progression*/;
     then
        ...;
        w_regress = 1;
    else
        /*regression*/ ...;
```

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 the TP to decided fuzzing state. The forkserver carries out the state transition for the TP. The policies of state transitions (i.e., when to *stay*, *progress* or *regress*) will be discussed in Section 4 and evaluated in Section 5.

4 IMPLEMENTATION

We implement yFuzz based on AFL to utilize its coverage-guided testcase generation and the infrastructure of communication. Our code consists of 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. 6 with the components unique to yFuzz being shaded.

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 *state engine* is *has_new_state*(), where the information from the forkserver and the 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-clang-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 3.5). The state transition implements the three branches (*staying*, *progression* and *regression*) mentioned in Section 3.4. And the *recvsignal* components stands for the logic of reading from/writing to the pipes and shared memory.

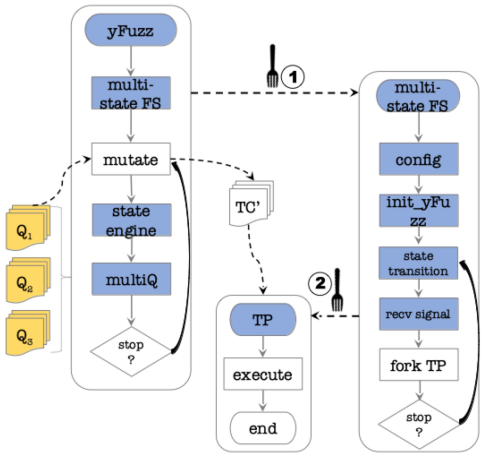

Fig. 6. Implementation of yFuzz based on AFL

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. Both the server and the client are instrumented for fuzzing. This step is done manually. (2) In addition to the code coverage and exit status, the TP will share information such as *TPstate* and *pstate* with the state engine in the fuzzer. (3) The fuzzer and the 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.

The logic of reading *pstate* is shown in Algorithm 1. 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 3.4.

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. 2 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. 2, 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. All the fuzzing states form a tree-based structure, where each node is a fuzzing state and the edges represent the transition. We agree that there might be multiple states for the same packet, e.g., one node could have multiple children. However, the purpose of profiling is to let each packet build their children nodes, so that we have an approximation of the number of children nodes each node has. Based on this information, we can have a "smarter" starting schedule for the actual fuzzing process (after profiling). The goal of profiling is not to get all and correct states of the protocol, but to have a broad view of the structure of the tree. 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 3 line 33) or the index of current queue entry exceeds $max\_entries$ (Algorithm 3 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.

5 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?

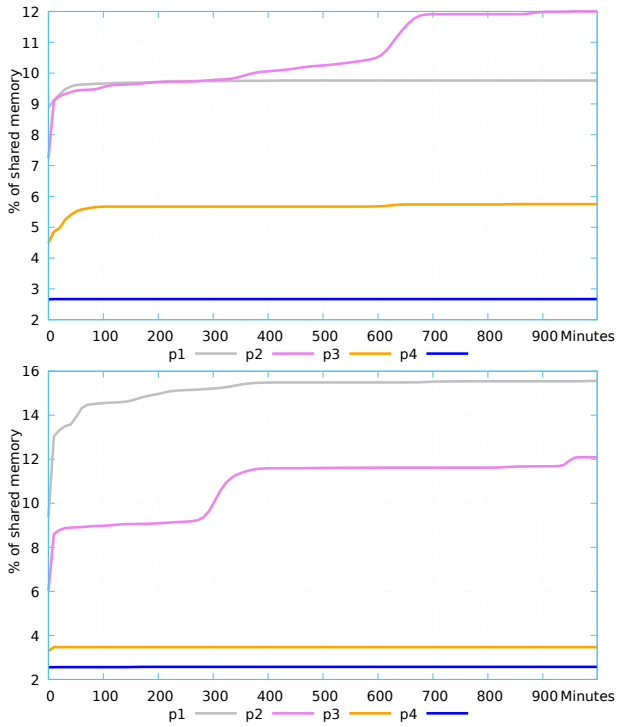

Fig. 7. Code coverage trend of fuzzing single packet at four different stages with OpenSSL 1.0.1f and 1.1.0f.

5.1 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 1.0.1f, 1.1.0f) as our benchmark. As mentioned in [3], we first compile OpenSSL using $afl\text{-}clang\text{-}fast$ to generate the static libraries which are invoked from the instrumented TP to perform handshake. We add $init\_yfuzz()$ after each packet is generated, under the condition "$pstate == TPstate$". In the 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 [18] (a compile-time tool that is able to discover memory corruption bugs) to consider more crash conditions.

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

5.2 Effect of single-packet fuzzing

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 [3].

In the case of OpenSSL version 1.0.1f, fuzzing different packet results in different code coverage as shown in Fig. 7. Fuzzing the first and second packet typically can yield more code coverage than fuzzing the third and

Table 1. Statistics of fuzzing single packet (OpenSSL 1.0.1f) 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

fourth packet. The average numbers are shown in Table 1. 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.

Table 2. 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	Hours	Covered	U1	U2	U3	U4
1.0.1f	6	7677	563	955	30	386
	10	8879	373	962	29	360
	24	8896	312	966	32	359
1.1.0f	6	10721	1472	755	26	296
	10	10093	2123	81	15	293
	24	11272	2054	738	22	295

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

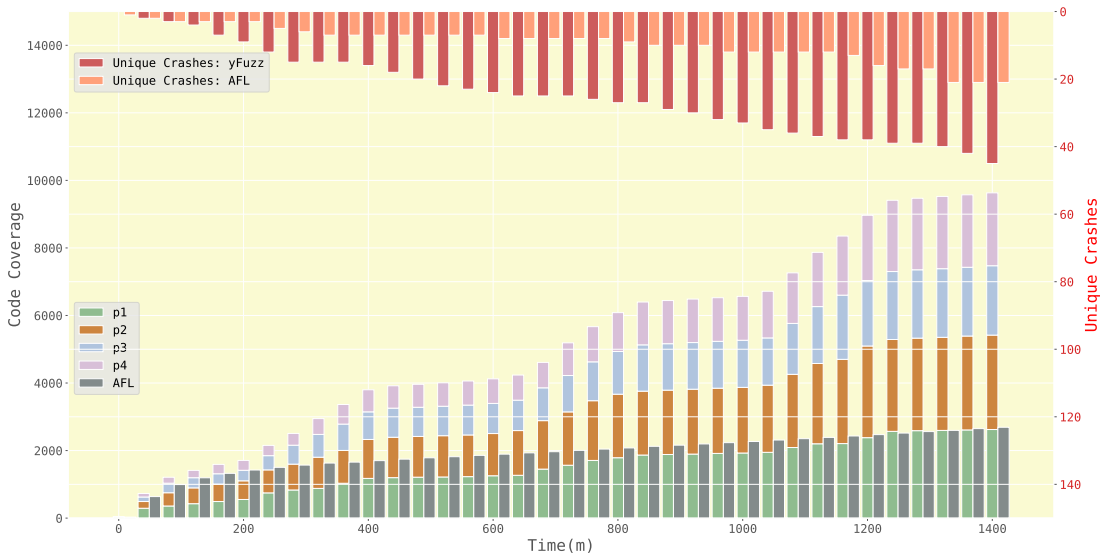

Fig. 8. AFL vs yFuzz: bottom bars show the composition of code coverage and top bars show the number of unique crashes. Note that we subtract the initial code coverage (7.9% in this case) explored by the seed $p1$ from each bottom bar to show the increments of code coverage. And the code coverage is represented as the number of discovered edges.

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 (or 10, 24) hours, we force the progression to fuzz the next packet, by clearing the code coverage bitmap (the global variable *virgin_bits* in AFL) without relaunching AFL. The experiment results are shown in Table 2. 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 1, 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 2 column $U1$, $U2$, $U3$ and $U4$.

In summary, the experiments conducted in this section have 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 have proven the need of stateful fuzzing, which is to fuzz different packets interactively and heuristically.

5.3 yFuzz: progression and regression

After the profiling stage (as mentioned in Section 4), 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).

Table 3. Comparison between yFuzz, AFLNET and TLS-fuzzer for detecting total crashes.

Time/h	1	6	12	18	24
yFuzz	283	548	1162	1846	2443
AFLNET	0	465	988	1354	1773
TLS-fuzzer	338	338	338	338	338

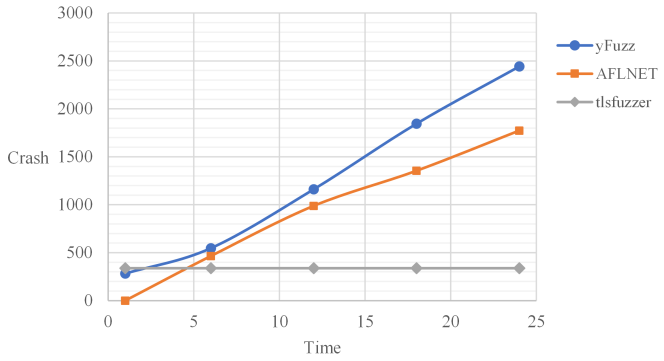

Fig. 9. Total crashes detected by yFuzz, AFLNET and TLS-fuzzer over 24 hours.

In the case of AFL, fuzzing $p1$ or $p2$ results in better code coverage and unique crashes as shown in Table 1. 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. 8, 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 2) 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).

5.4 Comparison with other baselines

We compare yFuzz with two baselines to demonstrate the effectiveness of our design in finding total crashes. The first baseline is AFLNET [31], which is a grey-box fuzzer for protocol implementations. Similar to yFuzz, it takes a mutational approach and uses sate-feedback to guide the fuzzing process in addition to code-coverage

feedback. However, unlike yFuzz, AFLNET does not support a multi-state forkserver (with staying, progressing, and regressing actions) for implementing different state-aware fuzzing strategies. The second baseline is TLS-fuzzer [23]. It is a test suite for SSL/TLS implementations by leveraging the written scripts to verify correct error handling while using simple fuzzing techniques for testing. Despite slight differences in testing environments, we compare yFuzz with AFLNET and TLS-fuzzer over OpenSSL with respect to the number of total crashes found through fuzzing. We note that unlike yFuzz, which can be applied on both the client-side and server-side, the two selected baselines can only be performed on the latter. Therefore, we collected the number of total crashes over time on the server-side in Table 3 and showed how the number of crashes grow over time for each baseline in Fig. 9.

We ran controlled experiments for a total of 24 hours and collected the data every 6 hours (except the first group, which is recorded 1 hour after the starting of the test.). As shown in Table 3, AFLNET detects less crashes than yFuzz at any time during the experiments. In the end, yFuzz is able to find 38% more crashes than AFLNET. Even though AFLNET is also a stateful fuzzing tool, there is no memorization of the protocol states. In contrast, yFuzz can memorize "interesting" protocol states to flexibly schedule resources to the states that are more promising to locate more latent crashes. As shown in Fig. 9, the gap between yFuzz and AFLNET also widens, since different fuzzing actions (i.e., staying, progressing, and regressing actions) enabled by yFuzz allows fuzzing strategies with both high coverage and concentration on regions of interest. When it comes to another fuzzing tool TLS-fuzzer. While it utilizes fuzzing techniques for testing (e.g., randomization of available inputs), the scripts are generally written in a way that verifies correct error handling with limited fuzzing capabilities. The experiment shows that it only generates a small number of valid input mutations, and therefore limits the performance of TLS-fuzzer in finding more crashes. Increasing fuzzing time from 6 to 24 hours, it fails to find more crashes. yFuzz outperforms both baselines in detecting total crashes.

5.5 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 1, we can derive that when fuzzing single packet $p1$, the average number of the 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 [26], which greatly downplay the overhead (such as *multiQ* updates) caused by yFuzz.

6 RELATED WORK

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

Protocol fuzzing: Few greybox fuzzers are designed specifically for protocols (in general, stateful programs). As discussed in Section 1, most of the existing protocol fuzzers are either whitebox [1, 30] or blackbox [2, 14, 16, 19]. The whitebox fuzzers typically only monitor process/network failures and lack the guidance for smarter testcase

generation and power scheduling. Blackbox fuzzers inevitably suffer the incomplete/inaccurate packet reverse-engineering analyses. 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. While grey-box fuzzers like [31] for protocol implementations can also be stateful, they simply use sate-feedback to guide the fuzzing process in addition to code-coverage feedback. In contrast, yFuzz leverages a multi-state forkserver to support different fuzzing actions (i.e., staying, progressing, and regressing actions) and to enable different fuzzing strategies with both high coverage and concentration on regions of interest. Compared with our preliminary work [10], yFuzz makes a number of extensions. In particular, it provides a complete implementation of the communication channels among the fuzzing engine, the forkserver and the testing program, as well as the multi-state forkserver to enable various fuzzing strategies through the process of progression and regression.

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 [15, 21, 25, 33, 37, 45**?**]. AFLFast [5] 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 [4] 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.

Symbolic execution and tainting: Techniques such as tainting and symbolic execution are also employed to complement greybox fuzzing [24, 27]. Angora [7] 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 [6] utilizes tainting and symbolic execution to determine the dependencies between input bytes and program control flow graph, in order to decide which bytes to mutate (optimal input mutation ratio) during fuzzing. Drill [35] 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, 17, 38, 43**?**]. Angora [7] and NEUZZ [34] 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.

Program transformation and rewriting: Program transformation and rewriting have also been applied in the area of fuzzing and program security. Program transformation aims to facilitate fuzzing by transforming testing programs [22, 26, 29]. T-Fuzz [29] dynamically traces the testing programs to locate and remove the checks once the fuzzer gets stuck. Untracer [26] 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. Program slicing/rewriting techniques have been used to identify and remove undesired code in the program/protocol to reduce the attack surface through dynamic tainting [8, 9, 11, 40] or static code clone analysis [41, 42].

7 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. Our experimental results show that yFuzz can achieve almost double code coverage and unique crashes when comparing to only fuzzing the first packet during the protocol communication (which is adopted by current greybox fuzzer).

Our current design of yFuzz has the following limitations that we are going to improve.

Instrumention of TP: yFuzz would be a more general and powerful fuzzer if it can work with blackbox program binaries. Thus, we plan to investigate techniques such as dynamic tainting (to locate the basic blocks that generate/process packets) and static binary rewriting to make yFuzz functional on program binaries. Also, currently inserting forking points need to be done manually, which requires understanding of the implementation details of the protocol. we will investigate mechanisms to ease the efforts.

Flexible power schedules: There could be more options to improve the fuzzing performance for specific protocols. For example, 1) the selection of $TPstate$ is flexible: In addition to choosing the number of packets seen in one TP execution as the value of $TPstate$, $TPstate$ could also be the size of a particular packet, execution time of the message flight(s), different dynamic state transitions and so on (and proper 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.

REFERENCES

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

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

[3] Hanno Bock. 2015. *How Heartbleed could've been found.* https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html

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

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

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

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

[8] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. Damgate: Dynamic adaptive multi-feature gating in program binaries. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. 23–29.

[9] Yurong Chen, Tian Lan, and Guru Venkataramani. 2019. Custompro: Network protocol customization through cross-host feature analysis. In *International Conference on Security and Privacy in Communication Systems*. Springer, 67–85.

[10] Yurong Chen, Tian Lan, and Guru Venkataramani. 2019. Exploring effective fuzzing strategies to analyze communication protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. 17–23.

[11] Yurong Chen, Shaowen Sun, Tian Lan, and Guru Venkataramani. 2018. Toss: Tailoring online server systems through binary feature customization. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. 1–7.

[12] Cisco. 2017. *SNMP Remote Code Execution Vulnerabilities in Cisco IOS and IOS XE Software.* https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20170629-snmp

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

[14] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of *TLS* Implementations. In *24th USENIX Security Symposium 15)*. 193–206.

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

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

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

[18] Google. 2018. *AddressSanitizer.* https://github.com/google/sanitizers/wiki/AddressSanitizer

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

[20] Heartbleed. 2014. *The Heartbleed Bug.* https://heartbleed.com

[21] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. 2018. Adaptive Grey-Box Fuzz-Testing with Thompson Sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*. ACM, 37–47.

[22] Ulf Kargén and Nahid Shahmehri. 2015. 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*. ACM, 782–792.

[23] Hubert Kario. 2015. tlsfuzzer. https://github.com/tlsfuzzer/tlsfuzzer.

[24] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. 2017. CAB-Fuzz: Practical Concolic Testing Techniques for {COTS} Operating Systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 689–701.

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

[26] Stefan Nagy and Matthew Hicks. 2018. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. *arXiv preprint arXiv:1812.11875* (2018).

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

[28] OpenSSL. 2019. *Official tesing programs for OpenSSL fuzzing.* https://github.com/openssl/openssl/tree/master/fuzz

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

[30] J Pereyda. 2017. boofuzz: Network Protocol Fuzzing for Humans. *Accessed: Feb* 17 (2017).

[31] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.
[32] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.
[33] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.
[34] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2018. Neuzz: Efficient fuzzing with neural program learning. *arXiv preprint arXiv:1807.05620* (2018).
[35] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.
[48]]honggfuzz Robert Swiecki. [n. d.]. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. *URl: https://github.com/google/honggfuzz (visited on 06/21/2017)* ([n. d.]).
[37] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2018. Superion: Grammar-Aware Greybox Fuzzing. *arXiv preprint arXiv:1812.01197* (2018).
[38] Valentin Wüstholz and Maria Christakis. 2018. Learning Inputs in Greybox Fuzzing. *arXiv preprint arXiv:1807.07875* (2018).
[39] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2313–2328.
[40] Hongfa Xue, Yurong Chen, Guru Venkataramani, and Tian Lan. 2019. Hecate: Automated customization of program and communication features to reduce attack surfaces. In *International Conference on Security and Privacy in Communication Systems*. Springer, 305–319.
[41] Hongfa Xue, Guru Venkataramani, and Tian Lan. 2018. Clone-hunter: accelerated bound checks elimination via binary code clone detection. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 11–19.
[42] Hongfa Xue, Guru Venkataramani, and Tian Lan. 2018. Clone-slicer: Detecting domain specific binary code clones through program slicing. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. 27–33.
[43] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2139–2154.
[44] Tian Lan Yurong Chen and Guru Venkataramani. 2021. *Open Source yFuzz tool.* https://www.dropbox.com/sh/wb1u25f11fbjmnn/AAAP-lrAQ-1gD1VJTGZdERSIa?dl=0
[45] Michal Zalewski. 2014. *American fuzzy lop.* http://lcamtuf.coredump.cx/afl/
[46] Michał Zalewski. 2014. *Fuzzing random programs without execve().* https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html
[47] Zardus and Mrsmith0x00. 2019. *Preeny.* https://github.com/zardus/preeny
[48]]lipfuzz Yangyong Zhang, Lei Xu, Abner Mendoza, Guangliang Yang, Phakpoom Chinprutthiwong, and Guofei Gu. [n. d.]. Life after Speech Recognition: Fuzzing Semantic Misinterpretation for Voice Assistant Applications. ([n. d.]).

8 APPENDIX

Algorithm 2: Behavior of default *__AFL_INIT*(). The logic is simplified.

```
while TRUE do
    receive forking signal from fuzzer;
    pid = fork();
    if pid < 0 /*fork failed*/;
     then
        exit();
    if pid == 0 /*in child process*/;
     then
        return; /*continue execute the TP*/
    /*in parent process*/;
    send pid to fuzzer;
    wait for child to exit;
    send child exit status to fuzzer;
```

Algorithm 3: Workflow of yFuzz: **FS:** forkserver, **TP:** testing program

```
Data: multiQ[];
Initialization: Qid = 0 (or specified by user);
while fuzzing not stopped do
    Fuzzer: current_entry = multiQ[Qid]→current_entry;
    while fuzzing current_entry do
        Fuzzer: testcase = mutate(current_entry);
        Fuzzer: send_to_FS(Qid, forking signal);
        FS: receive(Qid, forking signal);
        FS: fork/progress/regress according to Qid;
        if FS forked then
            TP: execute and send_to_FS(exit status);
        else if FS progressed then
            FS: executes to the next forking point;
            FS: fork and wait;
            TP: fork at next point;
        else /*FS regressed*/
            TP: exit;
            FS: go back to last forking point;
        Fuzzer: collect execution information;
        Fuzzer: if testcase is interesting then
            add_to_Q(Qid);
        Fuzzer: if new protocol state occurs then
            Qid+=1;
            proceed_fuzzing();
            break;
        Fuzzer: if give up current fuzzing state then
            Qid-=1 ;
            regress_fuzzing();
            break;
    Fuzzer: current_entry = current_entry→next;
    if current_entry is NULL then
        multiQ[Qid]→current_entry = multiQ[Qid]→head;
        fuzzing_cycle[Qid] += 1;
```