Exploring Effective Fuzzing Strategies to Analyze Communication Protocols

ABSTRACT

In recent years, coverage-based greybox fuzzing has become popular forvulnerability detection due to its simplicity and efficiency. However, it is less powerful when applied directly to protocol fuzzing due to the unique challenges involved in fuzzing communication protocols. In particular, 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 stateful protocol fuzzing strategies for communication protocols to explore the code related to different protocol states. Our approach contains a state switching engine, together with a multi-state forkserver to consistently and flexibly fuzz different states of an compiler-instrumented protocol program. Our experimental results on OpenSSL show that our approach achieves an improvement of 73% more code coverage and 2× unique crashes when comparing against fuzzing the first packet during a protocol handshake.

KEYWORDS

Protocol fuzzing;Greybox fuzzing; Stateful fuzzing

ACM Reference Format:
. 2019. Exploring Effective Fuzzing Strategies to Analyze Communication Protocols. In *3rd Workshop on Forming an Ecosystem Around Software Transformation (FEAST '19), November 15, 2019, London, UK*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 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.

Existing protocol fuzzers can be broadly categorized into three classes: whitebox, blackbox and greybox fuzzers. 1) A whitebox or blackbox protocol fuzzer is typically 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. A whitebox fuzzer assumes that the specifications of the protocol is known while a blackbox fuzzer reverse engineers the protocol by packet/program analysis. 2) The greybox fuzzer works together with a proper testing program (TP) provided for the network protocol. It feeds the mutated inputs to the TP,

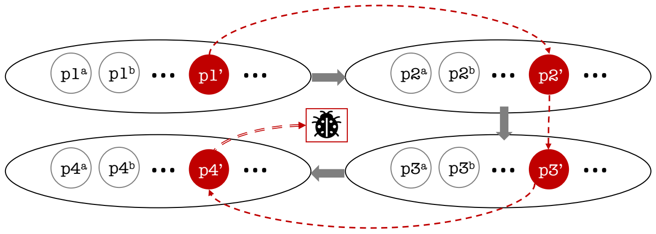

Figure 1: Fuzzing a four-packet-flight communication protocol: Different packets/fields are independent, e.g., each variation of packet $p2$ may lead to a different subsequent packet $p3$, $p4$. The bug can only be triggered when following the exact sequence $p1' \rightarrow p2' \rightarrow p3' \rightarrow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

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 [25] and AFL [33].

Limitations: Despite recent progress on fuzzing tools, a number of fundamental limitations still exist. While both whitebox fuzzers (such as [1, 24]) and blackbox fuzzers (such as [3, 12, 15]) perform "blind" fuzzing and fail to leverage some useful program execution information, the blackbox fuzzers particularly suffer from the inaccuracy of protocol reverse engineering. Greybox fuzzers [5, 6, 16, 19, 25, 28, 29, 31, 33] usually need a well-constructed TP with input interface to perform mutation and execution. Popular greybox fuzzers such as AFL can only fuzz the first packet by default.

Challenges: Protocol fuzzing possesses the following unique challenges. 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.

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 a progressive fuzzer for stateful communication protocols. In particular, our approach consists a *stateful fuzzer* and an *instrumented testing program (TP)*. The stateful 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 chooses 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. For example, in Fig. 1, we 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 for stateful protocol fuzzing. It consists of a stateful fuzzer and an instrumented TP, which work in concert to identify, replicate, and switch between different protocol states while maintaining execution consistency during fuzzing.
- We enable flexible power schedules[1] to fully capitalize the potential of our fuzzer design. In this paper, we implement and evaluate a new power schedule that continuously focuses on fuzzing the (current) most rewarding protocol states.
- Our experimental results of fuzzing the OpenSSL library show that we are able to achieve 1.73X code coverage and discover 3X unique crashes (on average) compared with the default AFL.

2 BACKGROUND

AFL [33] 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 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 initialization [34]. 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

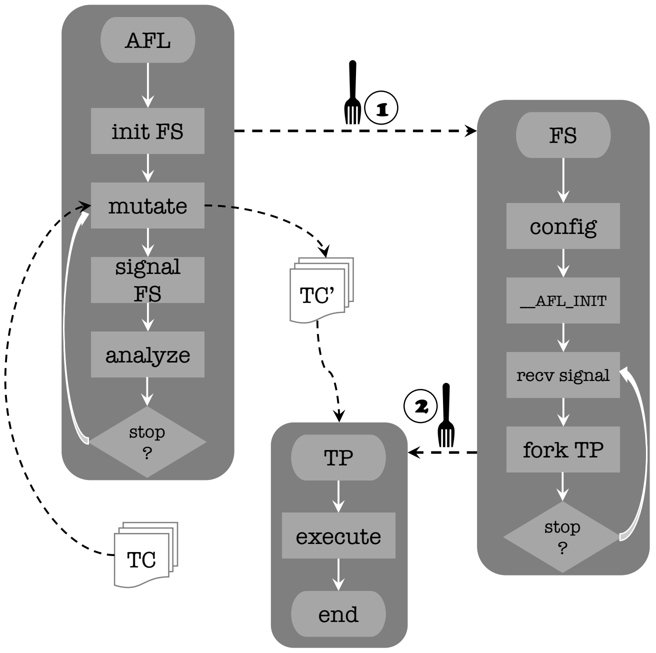

Figure 2: Simplified AFL forking workflow. FS: forkserver, TC/TC': testcase, TP: testing program

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

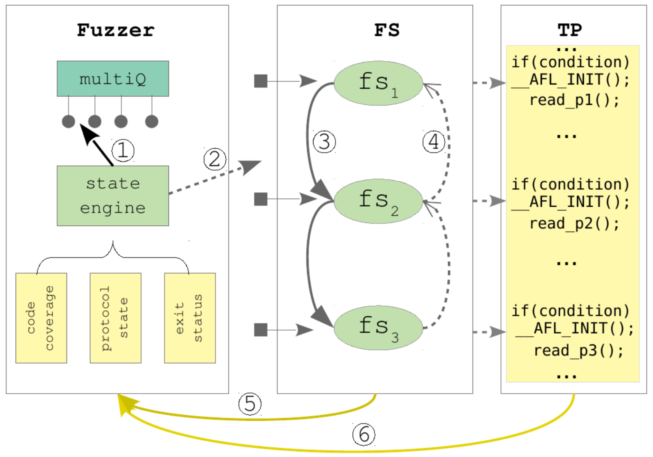

Figure 3: System Overview of our Stateful Fuzzer Design. FS: forkserver, multiQ: queues for storing different types of testcases, TP: testing program

call *execve*() only once. The work flow of forkserver is shown in Fig. 2.

The fuzzer calls *fork*() to generate the forkserver. The forkserver will perform some configuration first and then call *execve*() to execute the TP. The TP then executes 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. Every time the fuzzer finishes one mutation of the input and generates a new testcase, it will send a forking signal to the forkserver to generate a cloned TP that executes the mutated input. 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.

3 SYSTEM DESIGN

Our approach achieves stateful protocol fuzzing by combining a stateful fuzzer(Section 3.2) and an instrumented TP(Section 3.1) as shown in Fig. 3. The fuzzer contains an array of queues and a multi-state forkserver(FS). Each queue is used to only store the testcases that belong to the same fuzzing stage. For the example in Fig. 1, 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. When the forkserver is ready, it will keep listening to the fuzzer, waiting for signals to fork and generate a new process of TP. We will explain each module separately in the following subsections.

[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.

3.1 Testing program

For better understanding, we first show the changes of the TP for protocol fuzzing, 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 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. 3, but the idea and actual implementation of our approach are not limited by the number of states a protocol may have. While the forkserver is only initialized once in AFL, we conditionally initialize the forkserver multiple times for different fork points in TP.

3.2 State-aware fuzzer

The state engine passes forking and fuzzing state information to the 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. 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.

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. 3. In AFL, there is a 64kB shared memory between fuzzer and TP to track the code coverage information. Our design approach also shares the protocol states using the shared memory and pipes. 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

[3]In general, tools such as *preeny* [35] can be used to convert socket communications into file operations through preloading customized libraries, if the source code of TP is not available

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 4.

3.3 Coordination

In the example shown in Fig. 1, suppose we are mutating the first packet $p1$ and the forkserver is in state fs_1. At first the mutated $p1$ has wrong packet format and will not pass the format checking and the server sends back error message and stop the handshake. We use $TPstate$ to track the protocol state changes in TP. 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 state engine signals the forkserver to switch to fs_2. The forkserver at fs_1 forks another instance fs_2 to continue the fuzzing. Meanwhile, fs_1 will be blocked by fs_2 until fs_2 switches back (when the state engine decides to regress the fuzzing state).

In summary, our proposed stateful fuzzing design performs program fuzzing flexibly at multiple execution points with "memory" of precedent program states, by incorporating a state engine, 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 4 and evaluated in Section 5.

4 IMPLEMENTATION

Our implementation is based on AFL that utilizes its coverage-guided testcase generation and the infrastructure of communication. The multi-state fuzzer contains multiple queues (for testcases representing different stages of packets), a state engine (that analyzes the yield information and decides the next fuzzing state) and a stateful forkserver (that is able to switch between different forking states).

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 $TPstate$) with the state engine in the fuzzer. (3) The fuzzer and TP also communicate to record the packets when progressions are performed, such that we can keep track of the chain of packets that lead to vulnerabilities.

Search Policy: Based on the structure of the multi-state forkserver, we implement a DFS-like searching policy to transit among different fuzzing states. Taking Fig. 1 as an example, when interesting $p1'$ occurs, the fuzzer will use the program state of $p1'$ and starts to fuzz $p2$. If interesting $p2'$ is generated, then we 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 the fuzzer will regress to previous fuzzing state ($p4 \rightarrow p3$, $p3 \rightarrow p2$ or $p2 \rightarrow p1$). When the fuzzer 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. The fuzzer 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. 1, suppose our fuzzer is handling $p1$, 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. The fuzzer 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, our design 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, our fuzzer performs random progression based on the probabilities determined during profiling. Periodically, we update the probabilities by jointly consider the code coverage (and queue entries) in the profiling and testing stages. We 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, we set other thresholds for regression such as *max_Q_cycles* and *max_entries*. When the current fuzzing state finishes *max_Q_cycles* or the index of current queue entry exceeds *max_entries*, we will enforce regression to prevent wasting too much resources upon current fuzzing state.

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

5 EVALUATION

In this section, we evaluate our stateful fuzzer design to answer the following questions: (i) What is the performance of our fuzzer 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 our fuzzer due

[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.

Table 1: 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

to state forking and replay? (iv) What are the benefits of fuzzing strategy that targets higher yield on code coverage?

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 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, we invoke functions from *libssl.a* and *libcrypto.a* to perform the ssl handshake. We add $init\_yfuzz()$ after each packet is generated. 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.

5.2 Effect of single-packet fuzzing

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	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 using the design proposed by Bock et al [4]. By assigning different values to *packetID* in line 11, we can utilize the default forkserver in AFL to conveniently fuzz different packets.

In the case of OpenSSL version 101, fuzzing different packet results in different code coverage. 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 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.

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 coverage bitmap (the global variable *virgin_bits* in AFL) without relaunching the 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 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. That is, 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).

These two observations show the need of stateful fuzzing approach, and demonstrate the usefulness of fuzzing different packets interactively and heuristically.

5.3 Progression and Regression policies

After the profiling stage (as mentioned in Section 4), our fuzzer 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 1. In addition, AFL tends to stop discovering new code soon after a short amount of time when there is no interesting testcases. Our design is able to "escape" the fuzzing stages that are no longer profitable and flexibly switch between different states to discover new code.

On average of four 24-hour fuzzing, our stateful fuzzer design 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, our new fuzzer design 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).

6 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 [11, 16, 19, 25, 28, 29, 33]. 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. Our stateful fuzzer design, 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 [18, 21]. 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 [27] 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 [9, 13, 30, 32, 36]. Angora [8] and NEUZZ [26] 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: 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 whitebox 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. Blackbox protocol fuzzers [3, 10, 12, 15] have the same limitation. Our stateful fuzzer, 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.

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 to capture the state changes in protocols, and heuristically explore code spaces that are related to multiple protocol states. We implemented our design upon the popular greybox fuzzer, AFL and evaluate using OpenSSL (v101 and v110). Our experimental results show that we can achieve 1.73× code coverage and 2× unique crashes when comparing to only fuzzing the first packet during the protocol communication (which is adopted by current greybox fuzzer).

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

[26] 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.

[27] 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.

[28] 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)*.

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

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

[31] 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.

[32] 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.

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

[34] Michal Zalewski. fuzzing binaries without execve, 2019.

[35] Zardus and Mrsmith0x00. Preeny, 2019.

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