State Selection Algorithms and Their Impact on The Performance of Stateful Network Protocol Fuzzing

Dongge Liu*, Van-Thuan Pham*, Gidon Ernst†, Toby Murray*, and Benjamin I.P. Rubinstein*
* The University of Melbourne, Melbourne, Australia
Email: donggel@student.unimelb.edu.au, {thuan.pham, toby.murray, brubinstein}@unimelb.edu.au
† LMU Munich, Munich, Germany
Email: gidon.ernst@lmu.de

***Abstract*—The statefulness property of network protocol implementations poses a unique challenge for testing and verification techniques, including Fuzzing. Stateful fuzzers tackle this challenge by leveraging state models to partition the state space and assist the test generation process. Since not all states are equally important and fuzzing campaigns have time limits, fuzzers need effective state selection algorithms to prioritize progressive states over others. Several state selection algorithms have been proposed but they were implemented and evaluated separately on different platforms, making it hard to achieve conclusive findings. In this work, we evaluate an extensive set of state selection algorithms on the same fuzzing platform that is AFLNet, a state-of-the-art fuzzer for network servers. The algorithm set includes existing ones supported by AFLNet and our novel and principled algorithm called AFLNetLegion. The experimental results on the ProFuzzBench benchmark show that (i) the existing state selection algorithms of AFLNet achieve very similar code coverage, (ii) AFLNetLegion clearly outperforms these algorithms in selected case studies, but (iii) the overall improvement appears insignificant. These are unexpected yet interesting findings. We identify problems and share insights that could open opportunities for future research on this topic.**

***Keywords*-Fuzzing, network protocol, Monte Carlo tree search**

I. Introduction

Network protocols, such as the Simple Mail Transfer Protocol (SMTP), Real Time Streaming Protocol (RTSP), and Secure Socket Layer Protocol (SSL), are important. They enable consumer communication and recreational services (e.g., email and entertainment services) as well as business and government critical services (e.g., banking and cyber physical systems). A single critical vulnerability in these services could lead to catastrophic consequences. In 2001, the Code-Red worm infected more than three hundred thousands computers in less than 14 hours via Hypertext Transfer Protocol (HTTP) queries and caused at least $2.6 billion in damages globally [1]. In 2017, the infamous WannaCry ransomeware [2] exploited the Server Message Block (SMB) protocol, causing hundreds of millions of dollars damage. These incidents call for a thorough examination of network protocol implementations, especially network servers as they are usually Internet-facing. Given a server's Internet Protocol (IP) address and its port number, attackers—from anywhere on Earth—can send crafted requests aiming to discover vulnerabilities and develop harmful exploits.

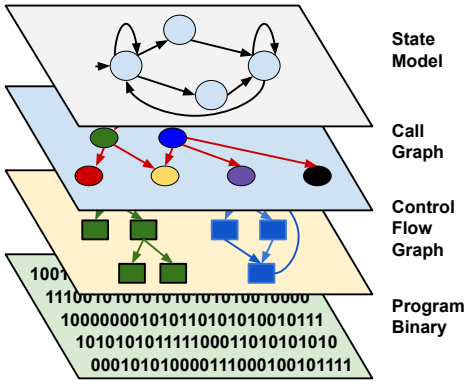


Fig. 1: Four abstraction layers of a server under test.

Fuzzing (a.k.a Fuzz Testing) [3], [4]—as recommended by the U.S. National Institute of Standards and Technology (NIST) [5]—is an effective technique for security testing. It has been used to discover thousands of vulnerabilities in large real-world systems [6], [7]. However, fuzzing network servers is challenging because servers are mostly *stateful*. That is, the server under test (SUT) takes sequences of messages (requests sent from the client) and its behaviour depends on both the current message and the current internal server state which is determined by earlier messages. State-of-the-art stateful fuzzers (e.g. Peach [8], BooFuzz [9], Pulsar [10], AFLNet [11], MACE [12])—including black-box, grey-box, and white-box approaches—tackle that challenge by modelling the SUT's state space and rely on the so-called *state model* to explore the space with *state selection* and test generation algorithms. As depicted in Figure 1, the state model is another abstraction layer of the SUT; it positions on top of the SUT's call graph, control flow graph, and the program binary.

There is a large body of research evaluating exploration algorithms at the program call graph and control flow graph levels to improve the effectiveness and efficiency of fuzzing [4], [13], [14], [15], [16]. One critical research topic is to study the impact of path, function, and basic block selection algorithms on fuzzing performance. Meanwhile, research on state selection algorithms—that work at the state model layer—is overlooked. For instance, in the recent AFLNet paper [11], only one state selection algorithm was evaluated. Similar is

true in many other papers [10], [12], [17]. Researchers tend to evaluate their algorithms, as implemented in a whole system, in comparison with other systems e.g. grey-box fuzzing versus black-box fuzzing or greybox-box fuzzing versus white-box fuzzing. In this paper, we evaluate an extensive set of state selection algorithms on *the same fuzzing platform*.

To conduct this research, we need to choose a representative base fuzzer that has existing state selection algorithms and supports an interface to easily implement new ones. Based on this criterion, we chose AFLNet [11], the first greybox fuzzer for stateful network protocol fuzzing, and built on top of American Fuzzy Lop (AFL) [18], one of the most popular and effective fuzzers today. Other fuzzers we considered do not support dynamic state selection algorithms [8], [9], or their source code is not publicly available [12].

We divided our investigation into two phases. In the first phase, we conducted a preliminary study of three existing state selection algorithms of AFLNet. Surprisingly, the results of 24-hour experiments on six subjects in the ProFuzzBench benchmark [19] show that the three algorithms achieved very similar results in terms of code coverage. Specifically, the maximum difference between average branch coverage of all benchmarks is 1.15%. Our deep dive into the implementation of AFLNet and those algorithms revealed problems leading to those unexpected results. First, AFLNet's state machine is too coarse-grained resulting in unreliable estimation of each state's potential. Second, its existing algorithms lack a principled way to balance the exploration-exploitation trade-off.

To address the identified problems, in our second phase, we designed and implemented a novel state selection algorithm namely AFLNETLEGION based on Legion [20], which is a successful variant of the famous Monte Carlo tree search [21]. We evaluated AFLNETLEGION in comparison with the existing AFLNET algorithms. The final results show that AFLNETLEGION clearly outperforms the baseline algorithms in selected case studies, demonstrating its principled design has advantage on programs that extend the motivating example. However, the overall improvement is insignificant. These are unexpected yet interesting findings.

By analyzing the fuzzing artefacts, we identified two potential problems behind these unexpected results. The most critical problem could be the low fuzzing throughput (i.e., number of executions per second)) of AFLNET. On average, it achieved only 20 execs/s, which is hundreds of times lower than the normal throughput achieved by *stateless* fuzzers like AFL [18]. With such a low throughput, systematic algorithms like AFLNETLEGION might not be able to fully unlock its potential.

This paper makes the following contributions:

- We conducted the first study, to the best of our knowledge, to evaluate an extensive set of state selection algorithms for stateful network protocol fuzzing. By analyzing the results, we identify problems and share insights that could open opportunities for future research in this topic.
- We extended LEGION's algorithm to protocol state selection and made it available to support future research.

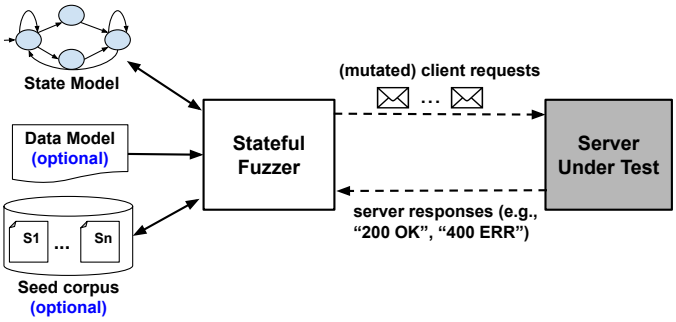


Fig. 2: Stateful Network Protocol Fuzzers

II. BACKGROUND AND RELATED WORK

A. Stateful Network Protocol Fuzzing

Fuzzing is a process of repeatedly generating (random) inputs and feeding them to the system under test to cover more code and discover abnormal behaviours (e.g., crashes, memory access violations) [4]. Since 1989, when the term fuzzing was coined by Prof. Barton Miller [3], it has received tremendous attention from both industry and academia, demonstrated by hundreds of published papers at top venues in Security and Software Engineering [4]. The majority of these papers have focused on fuzzing stateless systems, whose behaviour depends only on the current input, such as media processing libraries and utility programs [22], [23]. The *statefulness* property of network protocol servers pose a unique challenge for fuzzing.

To fuzz test stateful network servers, many techniques have been proposed including black-box, grey-box, and white-box approaches [8], [9], [11], [12], [24], [10], [25], as depicted in Figure 2. Unlike stateless fuzzing approaches, statefull fuzzers rely on a *state model* that partitions the state space of the SUT to assist the test generation process. By leveraging the state model and monitoring the SUT's responses, the fuzzer knows what is the current SUT state and generates messages, which can be valid or invalid with respect to the message format expected by the SUT at that state. These messages can be generated in a fully random way [3], from existing messages via mutations [11], or from a given data model/input grammar [8], [9]. Since not all states are equally important, the fuzzer also consults the state model to decide which states it should focus on. The decision is informed by *state selection algorithms*.

B. State Modelling Algorithms

Existing techniques tend to model state space using state machines. A state machine is a graph consisting of a finite number of states and transitions between states [26]. The server response codes (e.g., "200 OK", "400 ERR") are commonly used to abstract the server states. Many works use a simple state machine, where the destination of each transition is solely determined by the transition action and the origin state of the transition, regardless of other preceding states before the origin. For example, KIF [25] encodes a SIP

system to a state machine; SNOOZE and MACE [27], [12] uses it to record network operation information; and TLSHACKER [28], [29] analysed state machine models of TLS protocol implementations. PRISMA [24] defines the state machine model of the SUT in a probabilistic manner with a hidden Markov model [30], which further labels each transition with its execution probability.

To construct state models, many stateful fuzzing techniques rely on manually written models [8], [9], [31], [27], which requires the specification of the network protocols. Although this improves the probability of generating valid messages, the reliability of the model is subject to developers' understanding of the model and how well the specification describes actual implementations.

The state models can also be automatically constructed via either offline or online model learning. Offline learning is designed to infer a state model on a sufficient set of network traces and hence its accuracy relies heavily on that set. PRISMA [24] proposed to cluster samples of network traffic to infer a state machine of the SUT, which also abstracts the message template (a.k.a message format) from the sample traffic. PULSAR [10] combines offline learning with fuzzing so that it can generate messages from fuzzing templates with the state machine model and its embedded message rules.

Online learning is proposed as an improvement on offline learning, it can learn and refine the server state model at runtime. For example, Angluin's L* algorithm [32] is an online algorithm that has been applied in several research works [28], [12], [33] to gradually and dynamically infer a state machine model of the SUT through a number of messages. As generative fuzzing techniques, their message generation requires the SUT-specific data model or message grammar. In particular, [28], [12] require a test harness to translate between abstract requests and actual packets; while [33] proposes to learn it automatically.

C. State Selection Algorithms

PEACH, BOOFUZZ, SNOOZE [8], [9], [27] leave this burden to developers: developers have to embed their state selection strategies into the manually written state models. PRISMA [24] corresponds each state to a template and randomly chooses one according to the transition probability of its Markov model. PULSAR [10] has multiple templates in one state and prefers templates with more fuzzable fields. MACE [12] uses a priority queue to favour states traversed by past executions that visited a large number of unexplored basic blocks.

D. AFLNet: Stateful Grey-box Fuzzing

AFLNET is a greybox fuzzer for stateful protocol implementations. Unlike existing protocol fuzzers, it takes a mutational approach and uses state-feedback, in addition to code-coverage feedback, to guide the fuzzing process. AFLNet is seeded with a corpus of recorded message exchanges between the server and an actual client. No protocol specification or message grammars are required. It acts as a client and replays variations of the original sequence of messages sent to the server and retains those variations that were effective at increasing the coverage of the code or state space. To identify the server states that are exercised by a message sequence, AFLNet uses the server's response codes.

Regarding state selection policies, in its default state selection algorithm called FAVOR, AFLNET [11] uses an intuitive formula to consider states that are less often targeted, trying to balance exploiting the benefit of states that appear interesting with exploring the potential of the ones that seem to be less progressive. It also supports two more state selection algorithms called RANDOM and ROUND-ROBIN. While the former algorithm randomly selects states, the latter stores states in a kind-of circular queue and selects them in turns.

Once a state is selected, AFLNET chooses a seed input (i.e. a sequence of messages) that can reach that state. The seed selection algorithms are also named FAVOR, RANDOM, and ROUND-ROBIN. Like AFL [18], the FAVOR seed selection algorithm prioritises seed inputs that are small, cover more code, and take less time to execute. Meanwhile, the RANDOM and ROUND-ROBIN algorithms follow the same logic of RANDOM and ROUND-ROBIN state selection algorithms as described above.

E. Monte Carlo Tree Search and Legion

The *Monte Carlo tree search* (MCTS) algorithm [21] has proven its efficiency in exploring large search space, such as the state space of complex games like Go [34]. It iteratively refines a tree model of the search space via four steps: *selection*, *expansion*, *simulation*, and *back-propagation*. The tree can facilitate a *best-first* exploration policy, as four steps can estimate the productivity of various exploration directions and dynamically adjust the estimation with the statistics of simulation results.

LEGION [20] proposed a principled Monte Carlo tree search based algorithm for coverage-guided ordinary software testing. Its variation of MCTS attempts to learn the most promising program states to investigate at each search iteration based on past observation on code coverage. LEGION considers the symbolic tree of the program as the search space, and explores it with four steps that are marginally varied from the original MCTS to adapt to concolic execution.

Its selection step descends down the tree until a *simulation child* is selected. A simulation child is added to each node of the original MCTS search tree to allow simulation from their parent's program state, even if they are intermediate nodes.

Simulation is the second step in LEGION. It first generates inputs that preserve the path selected by solving the path constraint of the simulation child's parent. Simulating from intermediate nodes is designed to reduce the computation cost of constraint solving. For example, we do not have to solve the complex constraints of deep program states, if solving their ancestors' simpler constraints turns out to be as productive. LEGION will then execute the program under test with the inputs generated. Departing from the original MCTS, LEGION

records the whole execution path to determine the outcome (reward) of the current selection in the next step.

The third step is expansion, which checks if each new execution path exists in the current tree and adds the whole path to the tree. It records a reward to the current selection if the path is new. Each iteration ends with propagation. It increases the selection count of all nodes in the selection path by one, and adds the rewards to the discovery count of all nodes in the execution path. Note that some execution paths may fail to preserve the selection path, due to glitches in constraint solving in practice.

The design of LEGION also benefits light-weight (i.e., without symbolic execution) stateful protocol fuzzing techniques like AFLNET. For example, simulation from intermediate tree nodes is necessary, because when symbolic execution is not available, we cannot confirm if all child protocol states of a node are covered. AFLNETLEGION takes advantage of this design with its own modifications to adapt to protocol modelling and fuzzing, we explain the detail in Section IV.

III. PRELIMINARY STUDY

In this first phase of our investigation, we evaluated the three existing state selection algorithms namely RANDOM, ROUND-ROBIN, and FAVOR of AFLNET. We discussed the details of these algorithms in Section II-D.

A. Experimental Setup

Benchmarking programs. We conducted this preliminary study on six subjects (Table I) from ProFuzzBench [19], the largest public benchmark for network protocol fuzzers. We intentionally selected these *stateful* protocols for evaluation, because they are consistent with this paper's interest, namely the impact of selection algorithms on fuzzing stateful protocol implementations. Moreover, their setups produce stable results.

Subject	Protocol	Description
LightFTP	FTP	File Transfer Protocol
ProFTPD	FTP	File Transfer Protocol
Exim	SMTP	Simple Mail Transfer Protocol
OpenSSH	SSH	Secure Remote Login and File Transfer
OpenSSL	TLS	Secure socket connection
Live555	RTSP	Real-time media streaming

TABLE I: Subject programs

Seeds, Timeout and Repetition. We used the seed corpora provided by the ProFuzzBench benchmark. Regarding timeout, we followed the suggestions from [35] and ran our experiments for 24 hours. We also repeated the experiments 5 times to mitigate the influence of randomness.

Performance Measurement. Since AFLNET is a coverage-based approach, we focus on reporting its achieved code coverage. There is a (arguably) common understanding that covering more code leads to higher chance of discovering software faults. We used ProFuzzBench's scripts to fully automate the execution of the target servers inside Docker containers, and generate code coverage reports.

Platform. All experiments were conducted on the same host machine, running Ubuntu 18.04 64-bit with a 2.50 GHz Intel Xeon Platinum 8180M CPU and 128 GB of RAM.

B. Preliminary Results and Analysis

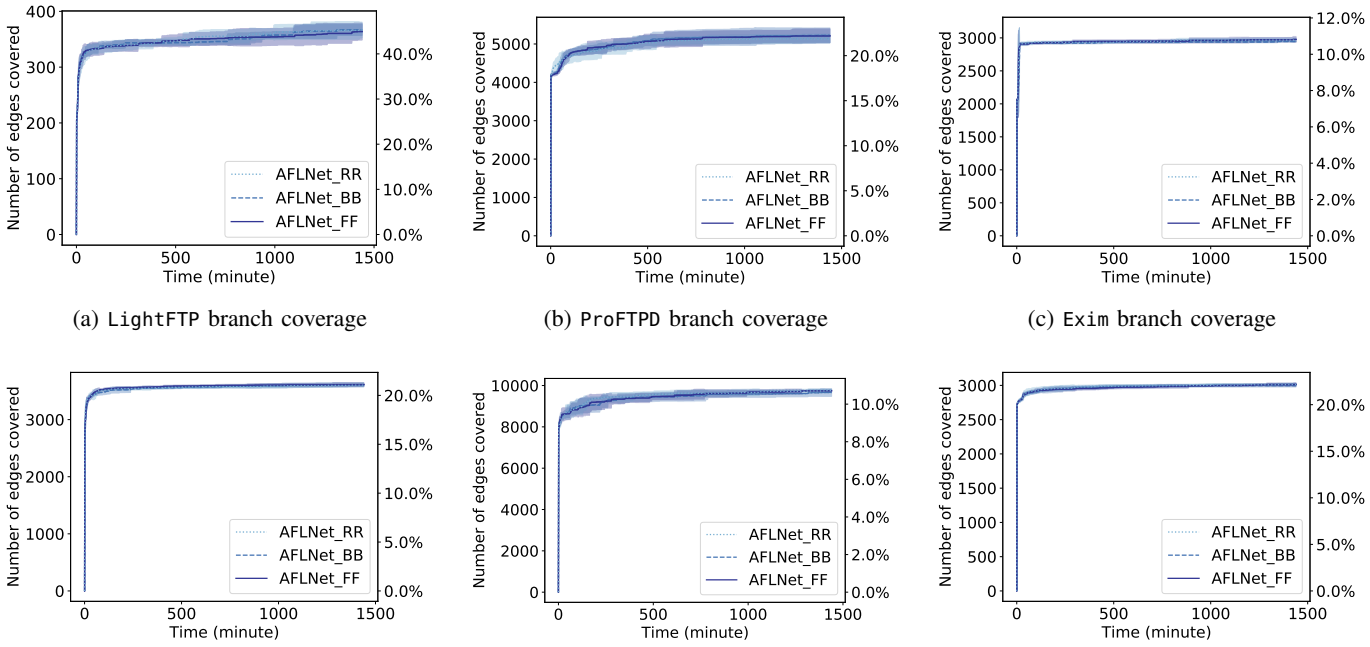
Our experiments show that distinct state selection algorithms exhibit minimum impact on the coverage performance. Fig. 3 plots three algorithms' average absolute and percentage branch coverage performance (respectively on the left and the right y-axes) on all benchmark subjects. The RANDOM, ROUND-ROBIN, and FAVOUR algorithms are respectively denoted by AFLNET_RR, AFLNET_BB, and AFLNET_FF, drawn in light blue dotted lines, blue dashed lines, and dark blue solid lines. Each mean curve is surrounded by its 95% confidence interval.

In general, all algorithms tend to exhibit similar coverage performance (indicated by their overlapping curves) and stability (shown by the similar widths of their confidence intervals) on each benchmark. For each subject, the differences between all algorithms' average final branch and live coverage values are within 1.2%. A 2-sample Student's t-test [36] further confirmed that such differences between their performance are statistically insignificant.

In addition to the close final coverage results, all algorithms also take very similar coverage trajectories on each subject. At the very beginning, all algorithms can quickly and stably cover similar amount of branches and lines with very tight confidence intervals. Then diverged performance starts to emerge between different algorithms (i.e., larger distances between mean curves) and between trials of the same algorithm (i.e. wider confidence intervals). Take benchmark subject `ProFTP` as an example, the RANDOM baseline outperformed the carefully designed FAVOUR for a short period immediately after the initial stage. Similarly, in `Exim`, the confidence interval of RANDOM expanded for a very short period, resulting in a spike-like-region at the top-left corner of Fig. 3c. Note that it is a fluctuation of the confidence interval (rather than mean) of RANDOM, caused by some trials covering much fewer branches than average at those timestamps. Nonetheless, the differences between curves are almost always within the confidence interval, indicating the divergence between algorithms is insignificant compared with the differences between trials of the same algorithm.

It is worth noting that, at the end of most experiments, three algorithms' curves visually overlap on each other with a tight confidence interval, implying their stably close performances. Two exceptions are the `FTP` benchmarks, where in `LightFTP` (Fig. 3a) RANDOM performed closely to ROUND-ROBIN and outperformed FAVOUR, and in `ProFTPD` (Fig. 3b) the performances of all were not as stable as the other benchmarks across the whole 24 hour experiment.

The indistinguishable coverage performance between FAVOUR and the other two distinct alternatives raises the following question:



(a) `LightFTP` branch coverage (b) `ProFTPD` branch coverage (c) `Exim` branch coverage

(d) `OpenSSH` branch coverage (e) `OpenSSL` branch coverage (f) `Live555` branch coverage

Fig. 3: AFLNET Overall results

Is the unexpected result caused by some weaknesses of FAVOUR?

In Section IV, we identify two main weaknesses of AFLNET and then propose corresponding fixes with a principled selection algorithm.

IV. MCTS-BASED STATE SELECTION ALGORITHM

To prove the unexpected results are irrelevant to the weaknesses in AFLNET, this section proposes a principled selection algorithm to address two main weaknesses of AFLNET that may limit its performance. We term the new algorithm AFLNETLEGION, as it extends LEGION's algorithm to stateful protocol fuzzing on the basis of AFLNET.

We identify the following two main weaknesses in AFLNET's state selection algorithm. First, its coarse-grained state machine model identifies each state only by individual response codes without distinguishing their prefixes, leading to unreliable state evaluation and selection. Second, its intuitive state evaluation formula lacks a principled way to balance the exploration-exploitation trade-off, which may inaccurate estimate the productivity of server states.

Section IV-A exemplifies how the compact state model of AFLNET may limit the performance of its state selection algorithm. Section IV-B explains why it can be improved with a more informative tree model inspired by LEGION. Section IV-C and Section IV-D describe the construction detail of the tree model. Section IV-E then introduces the principled UCT function used by LEGION as an improvement to the intuitive state evaluation formula of AFLNET.

A. Motivating Example

We will start with an example that spotlights the weaknesses of AFLNET and the corresponding improvements allowed by AFLNETLEGION. Fig. 4 is a case study illustrating how the effect of its state selection algorithm can be diminished by a coarse-grained state machine model. It uses the Name List (`NLST`) FTP command to explain the importance of selecting the *interesting* state in stateful protocol fuzzing. The command will instruct the server to list the file names that match with its parameter. Here we use an asterisk ($*$) as its parameter to test the `GLOB` library. The library is in charge of matching files and directories with wildcard characters (e.g. $*$) or other patterns. In this example, two sequences send the same requests in different orders: Sequence 1 (Fig. 4a) on the left globs with a `NLST` request without the existence of any file or directory; Sequence 2 (Fig. 4b) on the right creates a directory and a file before sending the same `NLST` command. As a result, the `NLST` command in sequence 2 is more interesting than sequence 1 to fuzz, because its mutants are more probable to cover more code of the `GLOB` library when the server attempts to match the parameter (e.g. *). A rigorous model of the server should distinguish these two sequences so that the selection algorithm can later learn that the state of sequence 2 is more productive in fuzzing than 1.

Fig. 5a illustrates the state machine of AFLNET after receiving the server's responses to two sequences of requests. The initial server state is represented by a dummy code 0, and each following state is identified with the response code of the latest command. Note that the `FTP` server returns the same code (`250`) to successful `NLST` and `STOR` commands,

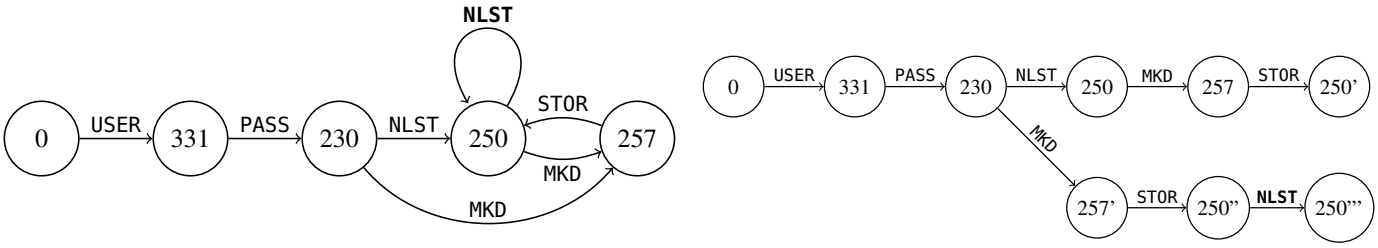
```
USER foo
PASS bar
NLST *  #Globbing matches nothing
MKD  temp
STOR sample.text
```

(a) Request sequence 1: `Globbing` before creating directory and file

```
USER foo
PASS bar
MKD  temp
STOR sample.text
NLST *  #Globbing matches new directory and file
```

(b) Request sequence 2: `Globbing` after creating directory and file

Fig. 4: Two sample request sequences involving globbing with `NLST`



(a) AFLNET's state machine model

(b) AFLNETLEGION's tree model

Fig. 5: Two models of the request sequences in Fig. 4
the bold font **`NLST`** is the interesting one from AFLNETLEGION

hence AFLNET's state machine is unable to distinguish them, and will think both request sequences produce the same state (`250`), where it can send the interesting `NLST` from sequence 2. Similarly, with more request sequences generated later, it may further mislead the selection algorithm to consider more noise request sequences as the prefixes of the interesting `NLST`, such as `USER -> PASS -> NLST`. As a result, the boring noise request sequence 1 lowers the estimated value of the state (`250`) and consequently reduces the chance of selecting the interesting state from sequence 2.

B. Algorithm

Observing the first weakness of AFLNET's algorithm, AFLNETLEGION proposes to assist state selection with a more fine-grained tree model illustrated by Fig. 5b. The state model construction is the same as AFLNET, except every node is identified with a unique response code sequence from the root to itself. Each also stores the same statistics and request sequences as AFLNET.

The tree model allows a more fine-grained server state modelling, by unfolding the states from AFLNET's model into tree branches. For example, the interesting `NLST` was sent from node `250"`, which is clearly distinguished from other nodes with the same response code but different prefixes (e.g. `250`, `250'`, `250"'`), so that they will be evaluated and selected independently. Consequently, no noise request sequence will undermine the actual statistics of each node or complicate its state selection. Section IV-C and Section IV-D explain the detail of this model.

C. Tree Nodes

Tree nodes are designed to allow AFLNETLEGION to fuzz the server from past server states preferred by the its MCTS-based algorithm. Each state of the server under test is identified by a sequence of response codes received during fuzzing.

Recall from the example in Section IV-B, the root of the tree represents the initial state of the server (denoted by a dummy code '0'), and every child node corresponds to a response code received after its parent code. Each tree node stores *interesting* request sequences that can reproduce the corresponding server state. A request is considered interesting to a node if it finds a new child of the node. This is because, under the MCTS paradigm, we consider all requests that produce the same response code from the same server state as an identical action, and only save (the first) one of them to avoid duplication. To facilitate state selection, each node also stores its selection count and discovery count, which respectively record the number of times it has been selected and the number of new child it has found. The same two statistics of each request sequence are also saved to support their selection.

Inherited from LEGION, AFLNETLEGION's nodes come in three types for different purposes. We use hollow nodes to represent the server states where we can launch fuzzing (e.g. right after executing `STOR sample.text` in Fig. 4b). In this way, we can fuzz interesting requests (e.g. `NLST *`) from the same server state as they were previously sent. A simulation node is attached to each hollow node, representing the option of launching fuzzing from the latter. Fig. 6 draws hollow nodes and simulation nodes respectively as circles and squares. In some cases, one request can trigger multiple response codes in a row, in which case we can only launch fuzzing from the server state represented by the last code in them. The last code of them is represented by a hollow node while the others are represented by redundant nodes. The latter is omitted from Fig. 6 as they do not affect fuzzing.

Note that the solid node from LEGION is not applicable in AFLNETLEGION, since symbolic execution is not available in the scope of this paper. Consequently, we can never know if all children of a node have been found, which highlights the necessity of a principled selection algorithm (Section IV-E) that can upper-bound the expected loss in coverage due to not selecting the best server state.

D. Tree Construction

Fig. 6 illustrates how AFLNETLEGION explores the tree-structured search space in its variation of MCTS: Each iteration of the search algorithm proceeds in the same four steps as LEGION with minor differences tailored to stateful protocol fuzzing. In Fig. 6, bold lines highlight the actions in each stage. Solid thin lines and nodes represent response sequence and server states that have been covered and integrated into the tree. Dashed lines and nodes draw the undiscovered response sequences and server states. The four steps execute in the following order:

Selection. It consists of two sub-steps in a row: Tree node selection and seed request sequence selection. The tree node selection descends from the root node by recursively applying the *selection policy*, until it reaches a simulation node. Then it applies the selection policy on all the request sequences stored in the tree node to find a seed for fuzzing. We leave the rule of the selection policy to Section IV-E.

Simulation. It replays the request to restore the server state corresponds to the parent hollow node (e.g. Node $250''$ in Fig. 5b), and then fuzzes the following requests (e.g. `NLST` $*$).

Expansion. It records all newly discovered response sequences and their request sequences into the tree. Inherited from LEGION, AFLNETLEGION maps each observed response code sequence to a path of tree nodes. When a new response code subsequence is found, it adds each of the new code as a new child node of its predecessor, and saves the corresponding request sequence to the parent of the first new node and every new descendants. As mentioned, when a single request triggers multiple response codes in a row, only the last code of such subsequence will be created as a hollow node while all its ancestors of the same subsequence (if any) will be added as redundant nodes. Again, a simulation node will be attached to each hollow node.

Backpropagation. It updates the statistics recorded in the tree to assist future decision making. For each mutant executed in the simulation stage, AFLNETLEGION increases the *past selection count* of each node in the selection path by one to discourage selecting the same nodes too often. For each new response code sequence observed, it increases the *past discovery count* of the parent of the first new node and all subsequent new nodes, to encourage selecting them in the future. The usage of these two counts are discussed in Section IV-E.

E. Selection Policy

To mitigate the second weakness of AFLNET, AFLNETLEGION's selection policy takes charge of choosing tree nodes and seed request sequences based on their statistics in the selection stage of MCTS.

It estimates the productivity of a server state regarding finding more uncover states. It relies on the same two statistics as AFLNET, namely, *past discovery count* and *past selection count*. Again, the former represents the number of response code sequences found under the target program state, more discovery indicates higher productivity of the state. The latter stands for the number of times the target state has been selected for fuzzing, less selection implies higher uncertainty about the state. Together, they attempt to balance exploitation of the states that appear to be most productivity based on past experience, against exploration of less-well understood parts of the protocol whose productivity (response code sequence discovery) are less certain.

$$UCT(N) = \begin{cases} \infty & S = 0 \\ \frac{D}{S} + \rho\sqrt{\frac{2\ln P_S}{S}} & S > 0 \end{cases} \quad (1)$$

Different from AFLNET (which relies on intuitive formulas), AFLNETLEGION inherites the UCT function (Eq. (1)) from LEGION to evaluate both tree nodes and request sequences: For node evaluation, D, S, and P_S respectively denotes *past discovery count* of the node, *past selection count* of the node, and *past selection count* of the nodes parent. For seed evaluation, D, S, and P_S respectively denotes *past discovery count* of the seed, *past selection count* of the seed, and *past selection count* of the seed's belonging simulation node. The discovery count records the number of new response sequences found by selecting that state/seed, the selection count records the number of times the state/seed has been selected for fuzzing.

With both main weaknesses mitigated, the next section will evaluate the performance of AFLNETLEGION.

V. EVALUATION

This section addresses two research questions:

(1) Is there any statistical evidence to support that AFLNETLEGION's state selection algorithm may be better than AFLNET? To answer this question, we conduct a detailed *case study* to analyse the coverage performance of AFLNET and AFLNETLEGION on ten sample code blocks carefully selected from the `GLOB` library of protocol `ProFTPD`, as an extension to the motivating example in Section IV-A.

(2) How do different modelling and selection algorithms impact *overall* coverage performance? To answer this question, we compare the *overall coverage performance* of 3 combinations of state and seed selection algorithms based on AFLNETLEGION against each of the best-performing AFLNET algorithm from the 6 benchmark subjects from Section III, following the same experiment setup.

A. Case Study

Analysing the coverage of the `GLOB` library can tell us if AFLNETLEGION's state selection algorithm is capable of improving coverage performance. Although the library appears

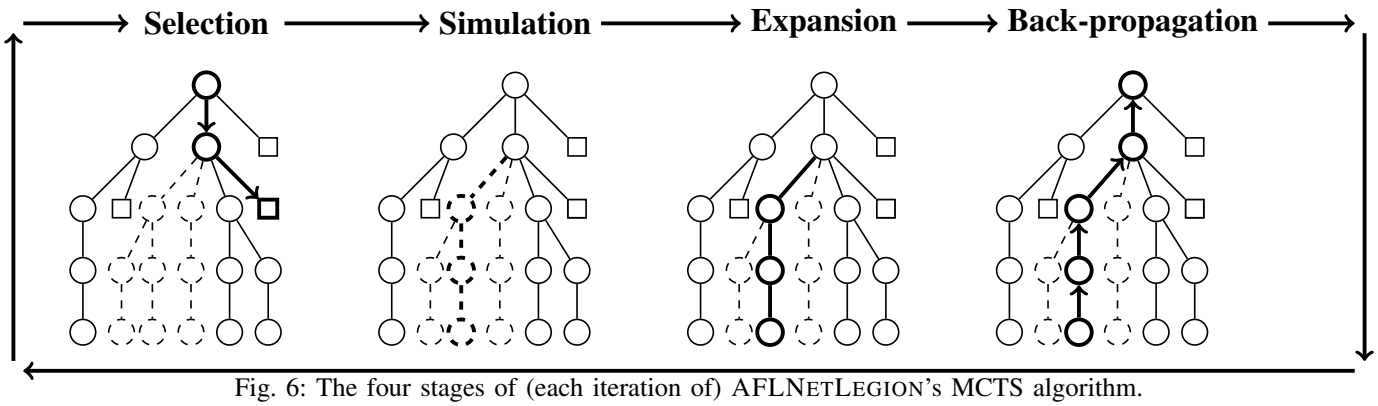


Fig. 6: The four stages of (each iteration of) AFLNETLEGION's MCTS algorithm.

easy to *reach*, there are many special cases that are difficult to *fully cover*. The former ensures most algorithms can generate some coverage statistics for comparison and analysis, while the coverage performance of the latter cases tells how efficient/effective the algorithm is. Recall the motivating example, the first three lines from Fig. 4a is a minimum test case to reach this model without testing many lines in the library. Alternatively, Fig. 4b is able to test `GLOB`'s functionality of expanding the wildcard by creating some files beforehand. Similarly, many other code blocks also correspond to particular cases, such as

- Case 1: Failing to allocate more memory
- Case 2: Globbing a file without path prefix
- Case 3: Globbing a file with a path prefix
- Case 4: Having a wildcard in a directory name
- Case 5: Globbing with metacharacter "$*$" or "?"
- Case 6: Globbing with metacharacter "[]"
- Case 7: Escaping a wildcard metacharacter
- Case 8: Successfully expanding a directory with a metacharacter
- Case 9: Globbing a directory that contains "\\"
- Case 10: Successfully finding at least 1 match

The case study will compare 12 hours performance of the selection algorithms of AFLNET and AFLNETLEGION on the ten cases above.

B. Overall Evaluation

While the case study highlights performance on particular interesting code blocks, the overall evaluation zooms out to the big picture. It investigates the overall impact of state selection algorithm on different protocol implementations.

C. Experiment Setup

We used the same experimental setup as described in Section III for this evaluation. Additional details follow.

Case Study-Specific Setup. Desipte AFLNETLEGION exhibits better overall performance than AFLNET in `ProFTPD`, the case study chooses only ten code blocks to investigate because their consistency with the example in Section IV-A, and their statistical evidence are typical to show why AFLNETLEGION can perform better. We repeated each experiments for 50 trials to support a statistically meaningful t-test evaluation.

Algorithms and Models. In addition to the best-performing algorithm of AFLNET on each benchmark program from Section III, we also add three new algorithms based on AFLNETLEGION. AFLNETLEGION uses the same backend as AFLNET, all differences come from their model and algorithm discussed in the previous sections. Its default algorithm AFLNETLEGION_UU selects both states and seeds with the highest UCT scores. An alternative, AFLNETLEGION_UR, uses UCT for state selection and selects seeds randomly. The third algorithm AFLNETLEGION_RR select both states and seeds randomly. The source code of AFLNET[1] and AFLNETLEGION[2] used in this paper are publicly available.

D. Experiment Results

1) Case Study: Fig. 7 compares the coverage performance of AFLNET and AFLNETLEGION on the ten representative cases in `GLOB` library. For both algorithms, we count the number of trials that each case is covered and use the percentage value (out of 50) to represent their likelihood of covering that case in an average trial. For example, 1 trial of AFLNET and 10 trials of AFLNETLEGION covered the third case, hence the probabilities are respectively 2% and 20%.

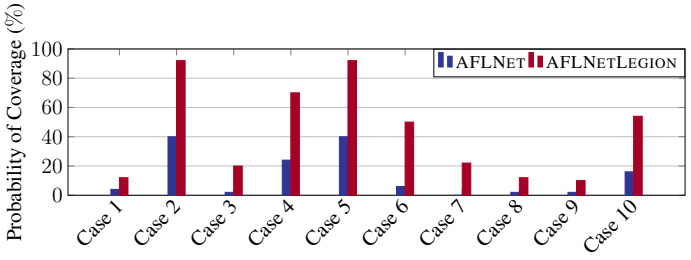


Fig. 7: Coverage performance on the `GLOB` library

AFLNETLEGION achieves a better overall coverage performance on the whole library because its higher probabilities in covering all ten interesting cases. For each case, AFLNETLEGION at least doubled AFLNET's coverage probability. These

[1]https://github.com/Alan32Liu/aflnet/tree/SANER2022
[2]https://github.com/Alan32Liu/AFLNet_Legion/tree/SANER2022

cases are difficult to cover either because of their a) *non-trivial prerequisites* or b) *rareness*.

Covering cases with non-trivial prerequisites requires the ability to accurately select server states that satisfy the requisites to launch fuzzing. As discuss in Section IV-B, AFLNETLEGION fulfils this requirement via unfolding the compact state machine into a tree model, where each node is distinguished by all the response codes received in the current session. In this way, it reduces noise in state selection (i.e., states that share the same response code but did not fulfil the prerequisites), allows a more accurate measurement of past reward of each state, and constitutes a reliable estimation of the potential of each server state. For example, case 8 requires globbing an existing directory, which will be missed if a fuzzer's algorithm failed to select a server state that has some directories created beforehand. Mixing such states with noises will largely lower the chance of covering this case. As a result, an average AFLNET trial has merely 2% probability to covers this case, while AFLNETLEGION increases this probability by 6 times.

Covering rare cases requires the ability to balance selecting states and request sequence that appear to be most rewarding and ones where rewards are less certain. Such ability allows intentionally generating enough mutants from the states that do not seem productive at first. For example, case 7 requires the globbing pattern to contain a `"["` followed by a `"]"` some characters later. Without knowing this specific grammar, AFLNETLEGION was able to cover this line more than once in an average trial. As a comparison, although AFLNET's heuristic uses the same statistics (i.e. discovery and selection count), none of its 50 trials was able to cover this case.

Both abilities constitute AFLNETLEGION's advantage in server state estimation and selection. With an accurate and reliable estimation of the complexity of the `GLOB` library, AFLNETLEGION on average generated over 14 times more test cases for it than AFLNET. Recall Section IV-A once gave an example to test `GLOB` library with `NLST` requests. AFLNETLEGION on average generated over 15 times more `NLST` requests than AFLNET, despite that the total number of all requests generated by both algorithms are of the same magnitude. AFLNETLEGION intentionally focused on `NLST` commands because its algorithm can identify the existence of interesting cases hidden inside the library, and notice mutants of `NLST` requests can cover them.

This experiment proves that we can *expect* AFLNETLEGION to perform better with its algorithm. In Section V-B, we will compare this algorithm against various alternatives regarding the impact on the overall coverage performance in 24-hour experiments.

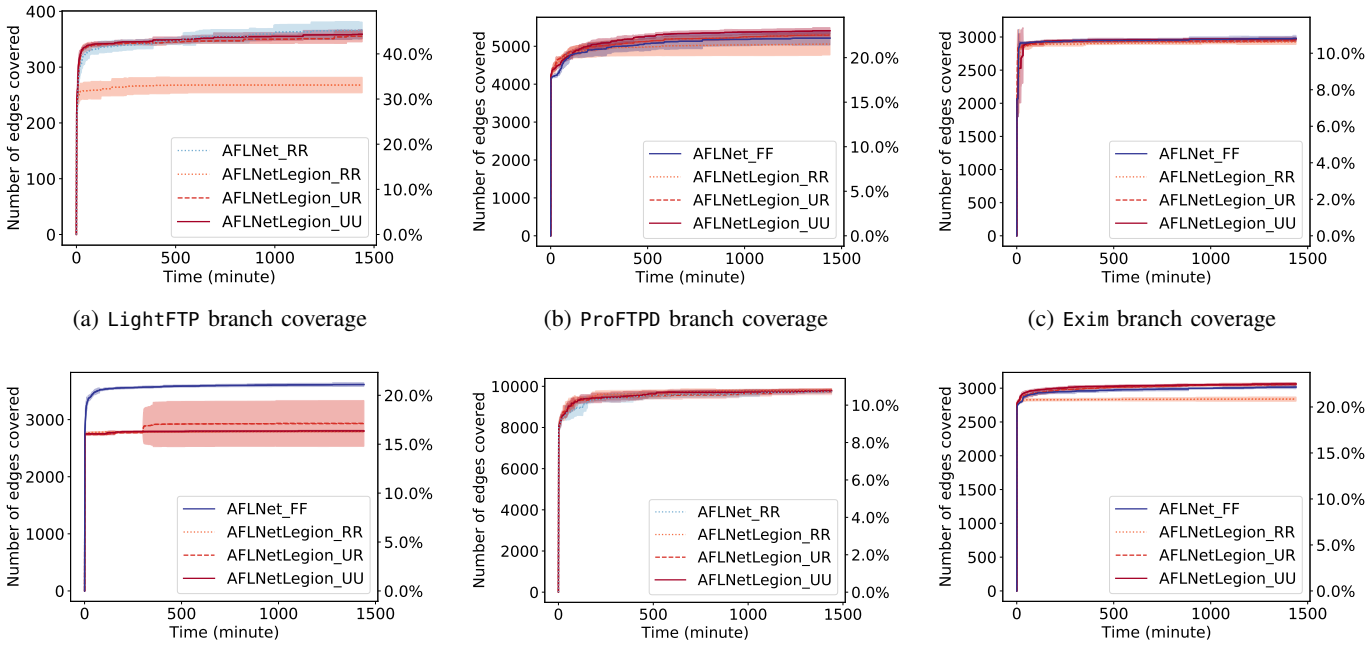
2) Overall Comparison: Despite the promising results from the case study, AFLNETLEGION failed to demonstrate largely better overall performance than its derivatives or the best performing algorithm from AFLNET. Fig. 8 repeats the experiment of Fig. 3 with its best performing algorithm against the three algorithms based on AFLNETLEGION.

In five out of six cases (except `OpenSSH`, discussed later), AFLNETLEGION_UU covered very similar amount of code as the best-performing AFLNET algorithm at the end. For example, the difference between their final branch coverage are proven to be statistically insignificant via t-tests. in `LightFTP`, `OpenSSL`, and `Exim`, AFLNETLEGION_UU outperformed the best-performing AFLNET algorithm by less than 2.5% branch coverage in `ProFTPD` and `Live555`, and largely under-performed it by 22% in `OpenSSH`.

In some cases, three algorithms based on AFLNETLEGION can discover branches and lines faster at the beginning. For example, AFLNETLEGION_UU and AFLNETLEGION_UR exhibits minor advantage at the first few minutes on `LightFTP`, `ProFTPD`, and `OpenSSL`. But at the rest time, all algorithms' performances are not visually distinguishable.

AFLNETLEGION_RR is the worst performing algorithm in almost all cases as we expected. Recall that the selection algorithm of AFLNETLEGION always descends down the tree from the root and stops when a simulation child is selected. With random selection, state selection at each level of the tree becomes a Bernoulli process, making the probability of selecting a simulation node at tree depth n approximately governed by a binomial distribution, where the likelihood of selecting that node is exponentially lower with a larger n. For example, in `OpenSSH`, the number of times that a child node is selected for fuzzing is almost half of its parent. The low probability of selecting deep tree nodes limits AFLNETLEGION_RR's ability to replay a long sequence of requests before mutation, reducing its ability to explore the behaviour of deep server states.

We also analysed why AFLNETLEGION's algorithms lose effectiveness on `OpenSSH` with the statistics of 40 experiments. Firstly, it violates AFLNETLEGION assumption on the sufficiency of winning (i.e., finding unique response code sequences). The average probability of receiving a unique sequence from `OpenSSH` is less than 1.2% for AFLNETLEGION and approximately 2.5% for AFLNET. As a comparison, with the same volume of experiments (40 trials) of `ProFTPD`, the same probabilities are approximately 50% and 31% for AFLNETLEGION and AFLNET respectively. Recall that AFLNETLEGION's UCT algorithm evaluates a state with its exploitation value (i.e., average unique sequence found by each selection) and exploration value (i.e., inverse selection frequency compared with its siblings). An extremely low winning rate will make the exploitation value negligible and purely determines the potential of each state by how often it has not been selected, which further lowers the winning probability. The exploration value of the algorithm optimistically estimates the potential of states within a confidence bound, which will be ineffective if the winning rate is outside the bound. This also made AFLNETLEGION's performance very unstable, ranging from uncovering 6 to 86 unique response sequences on average of every 10 trials. Secondly, MCTS's advantage in sequential decision making is useless in this particular benchmark program. While this advantage helps AFLNETLEGION to select which requst seqeunce prefix to preserve, most of the unique response sequences from `OpenSSH`



(a) `LightFTP` branch coverage (b) `ProFTPD` branch coverage (c) `Exim` branch coverage

(d) `OpenSSH` branch coverage (e) `OpenSSL` branch coverage (f) `Live555` branch coverage

Fig. 8: AFLNETLEGION overall results.

are found by not preserving any prefix. For example, FAVOUR found 89% of the new sequences by selecting the first state for 13501.25 times in an average trial.

Another reason why AFLNETLEGION failed to demonstrate good performance like LEGION is due to the unavailability of symbolic execution. Symbolic execution plays a crucial rule in optimising LEGION's performance in pruning nodes that have only one child or are not expected to find more execution paths, but is not available to lightweight techniques like AFLNETLEGION.

VI. DISCUSSION

By analyzing the results and other artefacts (e.g., fuzzing logs), we have identified two limitations of AFLNET that could prevent the state selection algorithms from fully unlocking their potentials. Understanding these problems opens opportunities for future research in this interesting yet challenging topic of stateful network protocol fuzzing.

Low fuzzing throughput (i.e. speed). On average, AFLNET and AFLNETLEGION achieved only 20 executions per second, which is hundreds of times lower than the normal throughput we could get while fuzzing stateless systems (e.g., media processing libraries). Stateful servers take longer time to reset and communications over the network is also much slower than file reading and writing operations. Having a low throughput in a given limited time budget, even if a state is correctly selected by a systematic algorithm like AFLNETLEGION, the fuzzer could not be able to generate enough test inputs to explore that state.

State-aware and structure-aware mutations. Servers under test normally expect well-structured inputs (i.e., the messages must adhere to some data format/input grammar). It means messages generated by random mutation operators currently supported by AFLNET are likely rejected by the servers. Therefore, they may not go deep to explore more code paths. Structure-aware fuzzing approaches like AFLSmart [37] and Nautilus [38] could be helpful. However, unlike stateless systems that require only one input model/grammar, stateful servers would require more—one for each state—because at each state, the expected input format could be different. Moreover, there could be dependencies across messages, making input generation for stateful servers more challenging.

VII. CONCLUSION

In this paper, we have discussed the challenges in stateful network protocol fuzzing and the importance of state selection algorithms to explore the state space. We have also shared our investigation into the effectiveness and efficiency of six different state selection algorithms. After analyzing the experimental results and fuzzing artefacts, we have identified potential problems, such as low fuzzing throughput and low quality generated inputs, preventing state selection algorithms from fully unlocking their potentials. Our future plan is to find solutions for those hindering problems and extend our research to cover more algorithms.

VIII. ACKNOWLEDGMENT

This research was partially supported by the Commonwealth Scientific and Industrial Research Organisation's Data61 under the Defence Science and Technology Group's Next Generation Technologies Program. We also thank Google Cloud for giving us research credits to advance this work with their computing platform.

REFERENCES

[1] D. Moore, C. Shannon, and K. Claffy, "Code-red: a case study on the spread and victims of an internet worm," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002, pp. 273–284.

[2] Website, "What is wannacry ransomware?" https://www.kaspersky.com.au/resource-center/threats/ransomware-wannacry, accessed: 2021-06-18.

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, pp. 32–44, 1990.

[4] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, pp. 2312–2331, 2021.

[5] P. E. Black, B. Guttman, and V. Okun, "Guidelines on minimum standards for developer verification of software," Oct 2021. [Online]. Available: http://dx.doi.org/10.6028/NIST.IR.8397

[6] Website, "Google clusterfuzz," https://google.github.io/clusterfuzz/, accessed: 2021-06-18.

[7] ——, "Microsoft onefuzz," https://github.com/microsoft/onefuzz, accessed: 2021-06-18.

[8] M. Eddington, "Peach fuzzer platform," https://wiki.mozilla.org/Security/Fuzzing/Peach, 2017.

[9] J. Pereyda., "boofuzz: Network protocol fuzzing for humans." https://boofuzz.readthedocs.io/en/stable/, 2017.

[10] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 330–347.

[11] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: a greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 460–465.

[12] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery." in *USENIX Security Symposium*, vol. 139, 2011.

[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[14] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[15] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 543–553, 2016.

[16] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 350–360, 2018.

[17] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *arXiv preprint arXiv:2110.06253*, 2021.

[18] Website, "American fuzzy lop (afl) fuzzer," http://lcamtuf.coredump.cx/afl/technical_details.txt, accessed: 2021-06-18.

[24] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, "Learning stateful models for network honeypots," in *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, 2012, pp. 37–48.

[19] R. Natella and V.-T. Pham, "Profuzzbench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[20] D. Liu, G. Ernst, T. Murray, and B. I. Rubinstein, "Legion: Best-first concolic testing," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 54–65.

[21] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai." in *AIIDE*, 2008.

[22] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, pp. 49:1–49:29, 2020.

[23] J. S. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[25] H. J. Abdelnur, R. State, and O. Festor, "Kif: a stateful sip fuzzer," in *Proceedings of the 1st international Conference on Principles, Systems and Applications of IP Telecommunications*, 2007, pp. 47–56.

[26] K. H. Rosen, "Discrete mathematics and its applications," 1984.

[27] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: toward a stateful network protocol fuzzer," in *International conference on information security*. Springer, 2006, pp. 343–358.

[28] J. De Ruiter and E. Poll, "Protocol state fuzzing of {TLS} implementations," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 193–206.

[29] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1492–1504.

[30] A. M. Fraser, *Hidden Markov models and dynamical systems*. SIAM, 2008.

[31] "Sulley: A pure-python fully automated and unattended fuzzing framework." https://github.com/OpenRCE/sulley, 2017.

[32] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and computation*, vol. 75, no. 2, pp. 87–106, 1987.

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

[34] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, p. 484, 2016.

[35] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[36] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[37] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[38] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.