SYNTAX-AWARE GREYBOX PROTOCOL FUZZING

FUZZING DE PROTOCOLE GREYBOX SENSIBLE À LA SYNTAXE

A Thesis Submitted to the Royal Military College of Canada
by

Sébastien Généreux, B.Eng.
Major

In Partial Fulfillment of the Requirements for the Degree of
Master of Applied Science in Electrical and Computer Engineering

September 2025

To my wife and children.

Acknowledgements

I would like to thank my supervisors, Dr. Vincent Roberge and Mr. Brian Lachine for their guidance.

Abstract

Fuzzing helps find potential vulnerabilities in applications by sending crafted inputs to them and observing their response. Code coverage achieved by those inputs is maximized to increase the likelihood of finding bugs. In the case of fuzzing network applications there is an additional layer beyond code coverage; state-space. Most bugs in network applications require them to be in a particular state; the bug is said to be stateful. To explore that state space, protocol fuzzers must model the state space in some way and try to abide by protocol syntax. AFLNet has become the protocol fuzzer most often compared to or extended in recent research. Efforts to make AFLNet faster have achieved significant code coverage improvements. Meanwhile, efforts to improve AFLNet's state-modelling and state-exploration features have shown mixed and limited results.

The aim of this research is to determine how adding syntax-awareness in AFLNet's mutation process changes the code coverage impact of state feedback features in AFLNet. This is based on the hypothesis that the current input crafting process generates mainly invalid inputs which do not make it beyond the network application's initial protocol parsing logic. To limit scope, the syntax-aware mutation process and validation are done for the File Transfer Protocol only. An ablation study of AFLNet features is completed across multiple target applications to measure the current code coverage impact of state feedback features. The same experiments are repeated using our syntax-aware fuzzer, AFLNet-Packmute, and then compared to the results of the ablation study. Measuring and discussing the change in code coverage impact achieves the aim of the research. We find that syntax-awareness increases the effectiveness of two AFLNet state feedback features, while decreasing the effectiveness of one.

Résumé

Le fuzzing permet de trouver des vulnérabilités potentielles dans les applications en leur envoyant des entrées et en observant leur réponse. La couverture du code obtenue par ces entrées est maximisée pour augmenter la probabilité de trouver des bogues. Dans le cas du fuzzing d'applications réseau, il existe une couche supplémentaire au-delà de la couverture du code : l'espace d'état. La plupart des bogues dans les applications réseau exigent qu'elles soient dans un état particulier ; on dit qu'elles sont des bogues avec état. Pour explorer l'espace d'état, les fuzzers de protocole doivent modéliser l'espace d'état et essayer de respecter la syntaxe du protocole. AFLNet est devenu le fuzzer de protocole le plus souvent comparé à ou étendu dans les recherches récentes. Les efforts visant à rendre AFLNet plus rapide ont permis d'améliorer considérablement la couverture du code. Contrairement, les efforts visant à améliorer les fonctions de modélisation et d'exploration de l'état d'AFLNet ont donné des résultats mixtes et limités.

L'objectif de cette recherche est de déterminer comment l'ajout de la prise en compte de la syntaxe dans le processus de mutation d'AFLNet modifie l'impact sur la couverture du code des caractéristiques de retour d'information sur l'état dans AFLNet. Cet recherce se fonde sur l'hypothèse selon laquelle le processus de mutation actuel génère principalement des entrées non valides qui ne vont pas au-delà de la logique initiale d'analyse du protocole. Pour limiter le champ d'application, le processus de mutation sensible à la syntaxe et la validation ne sont effectués que pour le protocole File Transfer. Une étude d'ablation des caractéristiques de retour d'état d'AFLNet est réalisée sur plusieurs applications cibles afin de mesurer leur impact sur la couverture du code actuel. Les mêmes expériences sont répétées en utilisant notre fuzzer sensible à la syntaxe, AFLNet-Packmute, puis comparées aux résultats de l'étude d'ablation. La mesure et l'analyse de l'impact de la couverture du code nous permettent d'atteindre l'objectif de la recherche. Nous constatons que le processus sensible à la syntaxe augmente l'efficacité de deux caractéristiques de retour d'état d'AFLNet, tout en diminuant l'efficacité d'une seule.

Contents

List of Tables

List of Figures

Acronyms

AFL	American Fuzzy Lop
ASAN	Address Sanitizer
CSV	Comma Separated Values
DAAP	Digital Audio Access Protocol
DICOM	Digital Imaging and Communications in Medicine
DNS	Domain Name Service
DTLS	Datagram Transport Layer Security
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
IPSM	Implemented Protocol State Machine
LLM	Large Language Model
PCAP	Packet Capture
RTSP	Real-Time Streaming Protocol
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
SSH	Secure Shell
TLS	Transport Layer Security
VM	Virtual Machine
XML	Extensible Markup Language

1 Introduction

Network applications are attractive attack targets because they are accessible remotely and provide services to many devices. This makes it important to find vulnerabilities in network applications so that they can be patched or mitigations be taken before they are exploited. An effective way to find potential vulnerabilities is fuzz testing (fuzzing) [1],[2]. Traditional binary fuzzers send inputs, known as fuzz, to the application, known as the target, over stdin or directly to functions. The target is then monitored for crashes or hangs.

Fuzzing network targets presents unique challenges. They manage multiple concurrent connections in a stateful and structured way. Protocol fuzzers address these challenges by sending fuzz as packets over TCP/IP sockets to the target. A protocol fuzzer also explores the state-space of the target to find stateful bugs, the bugs most commonly found in network targets [3],[4]. AFLNet [5] is the protocol fuzzer most often extended or compared against by recent protocol fuzzing research [6]–[12].

Each fuzz AFLNet creates comprises a series of network packets sent to the target. It does not have knowledge of protocol-syntax. It instead bases fuzz on initial network packet exchanges between a client and server application. AFLNet models the internal state of a target based on the response codes in the response packets. It heuristically selects which target state should be fuzzed, and selects an input that reaches that state. AFLNet favours inputs likely to find new coverage, and only modifies the portion of the input after the selected state is reached. The input is modified and is sent to the target as fuzz. The process that modifies the input into fuzz is called the mutation process and includes changes to the sequence of messages sent.

The target is compiled with special instrumentation that coarsely tracks what code has been executed as a result of the fuzz. Fuzz that introduce new or interesting code coverage are added to the set of inputs that seed new fuzzing rounds. Besides using code coverage feedback to maximize coverage of code, AFLNet concurrently explores the state-space of the target through

state feedback features.

1.1 Motivation

Research extending AFLNet has improved fuzzing speed (target executions per second), used alternative state feedback features, and added external knowledge to improve fuzz created [6]–[12]. To encourage research in protocol fuzzing, particularly AFLNet derivative fuzzers, a code coverage-based benchmark has been created called ProFuzzBench [13]. Research shows code coverage is correlated to, and moderately agrees with, bug finding ability for fuzzers [14]. The more code that is covered the more likely a fuzzer is to find bugs.

AFLNet fuzzing speed improvements have resulted in significant increases to final code coverage achieved. Since network targets are stateful, we expect that state feedback features in AFLNet and its derivatives should also contribute to code coverage improvement. Research extending state feedback features of AFLNet, however, have had limited impact. AFLNet's state representation method, using response codes, is very coarse and contains limited information depending on the protocol. Thus, research has looked at alternative state representations, such as NSFuzz [3], using source code state variables, or StateAFL [10], using long-lived memory regions. The alternative state representations perform worse, on par, or only slightly better than AFLNet depending on the target. AFLNet-Legion[11] proposes a new target state selection algorithm and a new tree model to replace AFLNet's Implemented Protocol State Machine (IPSM) which results in insignificant code coverage improvement and finds that random target state selection performed equally to AFLNet's heuristic-based approach. The authors of [11] believe this result is due to insufficient target executions over their 24-hour experiment and due to fuzz lacking state or structure awareness. A recently published study confirms that state feedback, when added to the existing coverage feedback, does not add code coverage value [15].

The statefulness of network protocols is a challenge which motivates research in protocol fuzzing. That state feedback aspects of protocol fuzzers based on AFLNet seem to provide limited contribution to code coverage, motivates this research.

1.2 Statement of Deficiency

AFLNet's authors did not initially conduct an ablation study, meaning it was not demonstrated how meaningful each of their separable design features were towards the total code coverage achieved. An ablation study has been published since this research was proposed [15], though it does not look at the sub-features that make up state feedback as a whole and does not consider the impact of region mutations. They find that the addition of state feedback atop code coverage feedback does not increase the code coverage achieved and that perhaps response codes are not adequate representations of target state.

This conclusion is in some ways supported and challenged by previous research. Alternative state representations and changes to state selection features have had limited impact on code coverage improvement [3],[10]–[12] meaning using response codes may not be the root issue. The authors of [11] suggest this limitation might be addressed through faster fuzzing, syntactically valid fuzz or fuzz aware of state-specific inter-message dependencies (semantics).

We hypothesize the root cause limiting the effectiveness of state feedback features is AFLNet's mutation process creating syntactically invalid fuzz. Fuzzer state feedback would have limited benefit to fuzzing if the fuzz being sent to the target fail to pass initial protocol parsing and cannot reach deeper in the application logic. The direct impact of invalid fuzz is reduced fuzzing performance as measured by code coverage. The indirect impact, we believe, is limited effectiveness of protocol fuzzer state feedback features.

1.3 Aim

The aim of this research is to determine how adding syntax-awareness in AFLNet's mutation process changes the code coverage impact of its state feedback features. The extent to which syntax-awareness impacts code coverage is determined by measuring the effectiveness of AFLNet's state feedback features before and after adding syntax-awareness in the mutation process. The change in effectiveness is used to achieve the aim. Effectiveness is measured as the code coverage impact of a particular AFLNet state feedback feature relative to a parent configuration without that specific feature. To limit scope, the syntax-aware mutation process is implemented for the File Transfer Protocol (FTP) only.

The features measured are: basic state feedback, favoured state selection, and favoured input selection. Basic state feedback in AFLNet, as defined by

this research, is when target state is modelled and specific states are fuzzed, without specific care for how states and inputs are selected. Favoured state and input selection are state-informed modes for each state selection and input selection which try to select the best state to fuzz and the best input to derive fuzz from. We are interested in the performance of these favoured modes relative to randomly picking seeds and states.

Adding protocol-syntax awareness to the mutation process allows fuzz to make it past initial protocol parsing logic in the target and thereby reach aspects of the target which depend more heavily on internal state.

1.4 Research Activities

The research activities for this research were conducted over the following 4 phases:

1. **Conduct an ablation study.** An ablation study measures the effectiveness of separable features in AFLNet using its current mutation process. This phase confirms findings in previous research and draws new conclusions about the value-added to code coverage of each separable AFLNet feature. Region mutations, though not a state feedback feature, are added for completeness.
2. **Develop AFLNet-Packmute: Protocol-syntax Aware Mutation Process.** An AFLNet derivative, AFLNet-Packmute, is developed which produces syntactically correct fuzz. This is based on the hypothesis that invalid fuzz limits the effectiveness of AFLNet state feedback features. AFLNet-Packmute allows some invalid fuzz to be created to fuzz initial parsing code and error handling code in the target.
3. **Verification.** Analysis verifies whether AFLNet-Packmute behaves as designed. That is, does it create only valid fuzz when instructed to do so, and invalid fuzz when permitted to do so? This phase includes activities that verify the correctness of analysis scripts and function behaviour.
4. **Validation.** Validation first involves measuring the code coverage impact of state feedback features in AFLNet-Packmute. Next, the change in impact is measured between these results and those obtained in the ablation study phase, conducted with AFLNet. Finally, using this measurement we determine how the addition of syntax-awareness into the AFLNet mutation process has changed the code coverage impact of state feedback features, achieving the aim of this research.

1.5 Results

This research finds that adding syntax-awareness to AFLNet's mutation process increases the code coverage impact of two of its state feedback features while reducing the impact of one state feedback feature. Favoured seed and state selection, where selection prioritizes seeds or states with more potential to find new coverage, see an increase in code coverage impact of +0.84% and +0.77% respectively. The basic process of selecting states and fuzzing the portions of message sequences exercising that state sees a reduction in code coverage impact. We believe this may be due to syntactically valid fuzz reaching further into the target's logic, causing slower fuzzing speeds and therefore reduced coverage. These research findings suggest previous work done to improve on AFLNet's state feedback features could have their effectiveness improved through syntax-awareness, and that targets where the maximum code coverage is achieved by a fuzzer configuration with state feedback enabled could see even greater maximum coverage.

1.6 Organization

Chapter 2 introduces terminology, tools and related work important to understanding this research. Chapter 3 then describes in detail the design of all research activities as well as the methodology used to arrive at that design. Next, Chapter 4 presents the results or outcomes of conducting the research as designed, including achieving the aim of the research. Lastly, Chapter 5 concludes, outlining contributions made by this research, its limitations and areas for future work.

2 Background

In this chapter the information necessary to understand the research conducted is presented, alongside existing research which directly relates to it. First, a discussion on binary fuzzing will introduce terminology and techniques used in the foundational fuzzing use-case. Protocol fuzzing is then discussed as a distinct fuzzing use-case, with further detail on the AFLNet fuzzer and ProFuzzBench benchmark. Next, best-practices in evaluating fuzzers explains the experimentation parameters used in Chapter 3. Lastly, a discussion on related works situates this research within the existing body of knowledge.

2.1 Binary Fuzzing

Barton Miller first introduced the concept of fuzz testing in 1990 when he observed that you could affect the reliability of UNIX utilities by sending them random bytes [16]. This became known as fuzzing, and the inputs sent to the targeted application are known as fuzz. Fuzzing is now an extensive research area, with nearly 300 papers published in the field between 2018 and 2023 in select top venues [17].

Fuzzers can be categorized based on the level of introspection into the target application they require. Blackbox fuzzers require a compiled binary of the target, no source code. Greybox fuzzers require target source code in order to add lightweight instrumentation that is used to evaluate or guide fuzzing. Whitebox fuzzers also require source code but use more intensive code analysis, such as constraint solving, to develop fuzz. The grey-white trade-off is primarily between fuzz quality and fuzzing speed.

Another fuzzer categorization is whether it is generating new fuzz from scratch, called generation-based, or mutating a seed to create fuzz, called mutation-based. Mutation-based fuzzers store their starting candidate inputs as seeds in a corpus. A mutation-based fuzzer is said to be evolutionary if

it uses feedback from the target's execution to guide fuzzing efforts. This guidance could inform which mutations to use, which seeds to select, and whether new fuzz should be added to the corpus. The fuzzer which popularized mutation-based evolutionary fuzzing is called AFL [18].

2.1.1 AFL

AFL's design principles are speed, reliability and ease of use, on the basis that fast stochastic fuzzing is better than slow precise fuzzing. Central to AFL is coarse but fast code coverage feedback in the form of a 64 KB map shared between the fuzzer and the instrumented target. At compile-time, each code branch is instrumented to update the coverage map with branch-tuple hit counts as shown in Listing 1. Each position in the map represents a branch-tuple, and its value indicates how often the branch-tuple is visited during a single target execution. The location address bit shift on line 3 is necessary to differentiate a flow of branch-tuple 1-2 from 2-1. Otherwise, the XOR operation would give the same result.

Listing 1 AFL branch-tuple coverage pseudocode [18]

```
1 cur_location = COMPILE_TIME_RANDOM;
2 shared_mem[cur_location ^ prev_location]++;
3 prev_location = cur_location >> 1;
```

This coarse coverage map will tell AFL if the fuzz sent is interesting; whether it introduced coverage not yet reached by existing seeds or changed the code execution path in an important way. Interesting fuzz are added to the corpus of seeds for later fuzzing rounds. The corpus is managed in-memory via a queue of entry structures. Each entry stores pointers to the next entry, the filename of the seed associated with the entry and metrics associated with that seed.

Fuzzing is conducted in a loop. In one iteration, a seed is selected from the corpus, assigned an energy score, then mutated and sent to the target according to that energy score before returning to select a new seed. Energy scores are assigned using a power schedule which considers the execution speed of the seed, the size of its bitmap, when in the fuzzing session's lifespan it was found, and a few others factors. Mutation involves applying a random number (between 1 and 128) of randomly selected mutations one after the other on the same buffer. Examples of mutations include bit flips, integer value addition/subtraction, insertion of random bytes or dictionary words, and block manipulations.

When fuzz causes a crash or hang of the target, it is saved to be investigated later. The conditions which cause crashes can be broadened by compiling the target with a sanitizer enabled. The most common sanitizer used is called Address Sanitizer (ASAN), which will terminate a program on occurrence of memory violations such as use-after-free or buffer overruns, and output a report. This broadens the bug classes that AFL fuzzing can find and is built into common compilers such as GCC [19] and clang [20].

2.2 Protocol Fuzzing

Protocol fuzzers historically tended to be generation-based since network application targets expect structured and state-aware input to execute most of their functionality. Notable ones include Peach [21] and BooFuzz [22]. These fuzzers need user-provided protocol and state models in the form of Extensible Markup Language (XML) files or Python code objects.

This trend has shifted since the advent of AFL and the popularity of mutation-based evolutionary fuzzers [23]. Now the state-of-the-art is to pair a mutation-based fuzzer with self-learning of target state space and/or protocol syntax [23]. This reduces the amount of tailored configuration needed to onboard a new target for fuzzing. Two common sources of initial inputs are pre-recorded packet captures (PCAPs) or to run a client application and have the fuzzer proxy packets. When using PCAPs, the fuzzer acts as the client. Many mutation-based protocol fuzzers introduce new mutations that effect the overall packet sequence being sent and some try to preserve inter and intra packet dependencies [24],[25],[26].

2.2.1 AFLNet

Based on AFL, AFLNet was created by Van-Thuan Pham *et al.* and was published in 2020 [5]. They propose that a network target's state can be derived from the server response code last received from the target. Building on AFL's [18] coverage-guided evolutionary greybox fuzzing approach, it derives fuzz from application-layer data, called a message, and explores the target application's state to maximize coverage. Its main contributions include:

- Operates on sequences of messages (as opposed to individual ones).
- Message sequence mutations (called region mutations).
- State feedback. It explores target state-space through self-learning of server response codes. This includes state-space representation through an IPSM, target state selection, state-aware seed parsing and state-aware seed selection.

The AFLNet authors found significant fuzzing improvement relative to a basic network-enabled version of AFL, AFLNwe [27], and BooFuzz [22], a generation-based blackbox protocol fuzzer.

AFLNet is seeded with raw TCP streams. Figure 2.1 shows how network traffic is first captured between a client application and target server and then the raw TCP stream is saved to a file which becomes one of the initial seeds. The more target server features exercised in the initial seeds the better for fuzzing.

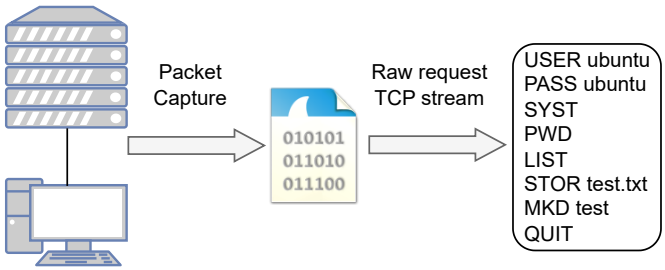

Figure 2.1: AFLNet process to create initial seeds

Algorithm 1 describes AFLNet's main procedure, which must be provided initial seeds and a target application. A queue of entries (seeds), bitmap and IPSM are globally held (line 1). First AFLNet will perform a dry-run where the initial seeds are sent to the target server one message at a time and the target's behaviour observed (lines 2-6). This phase builds the initial IPSM and populates the corpus with initial seeds. Next, AFLNet selects a target state to fuzz, selects a seed that exercises that state, creates fuzz by mutating that seed and lastly sends it to the targets (lines 7-10). These are repeated in a loop until fuzzing is done. Each step will be discussed in detail.

In the dry-run phase, the *AddToQueue()* function reads the initial seeds from disk and adds them as entries to the queue. When a seed is added to the queue it is parsed into individual messages using protocol-specific request extraction code that uses message terminators or length headers to do the splitting. AFLNet records the start and end byte of each of these messages and assigns them a zero-indexed ID. This data is called a *region* and is saved in the queue entry structure. In the case of FTP, it is as simple as splitting on each message terminator `\r\n`. Figure 2.2 shows the regions for an example FTP seed. This seed has eight regions. The region start and end position values are inclusive, and capture any message terminators.

Using this region data, the seed is parsed into a linked list of individual

Algorithm 1 AFLNet Simplified Main Program Loop

Input : Initial Seeds T, Target Program P

1: $Queue, Bitmap, IPSM \leftarrow 0$
2: **for** seed in T **do**
3: addToQueue($seed, Queue$)
4: $Message \leftarrow$ parse(seed)
5: $Response \leftarrow$ send($Message, P, Bitmap$)
6: updateStateMachine($IPSM, Response$)
7: **while** true **do**
8: $selectedState \leftarrow$ selectState($IPSM$)
9: $selectedSeed \leftarrow$ selectSeed($selectedState, Queue$)
10: fuzz($P, selectedSeed, selectedState, Bitmap, Queue, IPSM$) ▷ *fuzz target a few thousand times with this seed*

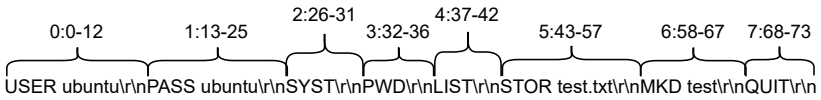

Figure 2.2: AFLNet FTP regions example

messages. Each message is combined with the internet and transport layer settings of the fuzzing session to build network packets which are sent to the target (Algorithm 1 lines 4-5). Using the same example seed, Figure 2.3 shows the exchange. Upon connection, the server responds with code 220 to indicate it is ready. Next response code 331 is sent to acknowledge a login attempt for user *ubuntu* followed by response code 230 to confirm successful login. Response code 221 confirms the session is over.

As AFLNet receives response packets it collects them in a single response buffer. Protocol-specific functions extract the sequence of response codes from the response buffer (Algorithm 1 line 6). This forms the state sequence traversed by the target as a result of the fuzz. The state transition sequence for our working example would be 0-220-331-230-215-257-150-226-150-226-257-221, where state '0'is the default initial state. Figure 2.4 shows the IPSM from this first sequence. Each transition is labelled with the region ID from the seed in Figure 2.2 which caused the transition. Regions 4 and 5 result in two responses from the server, code 150 to indicate the file status and code 226 to indicate the transfer is completed. Future state sequences are added to the IPSM if a new state or state transition is discovered.

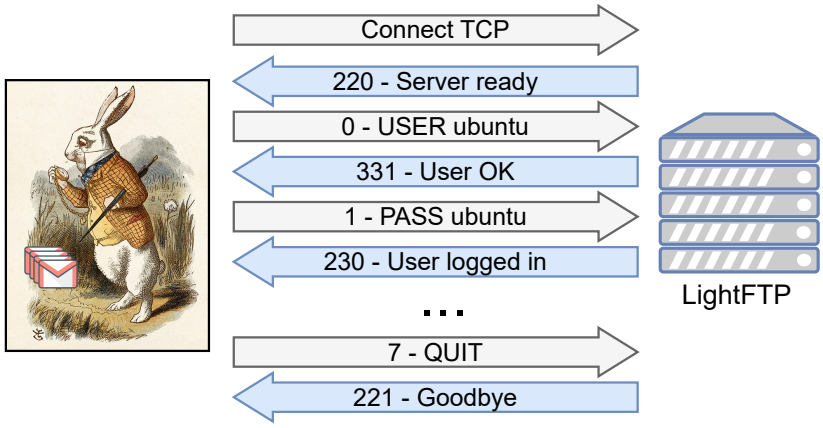

Figure 2.3: AFLNet sending FTP fuzz. Grey arrow are client sent messages and blue arrows are server sent

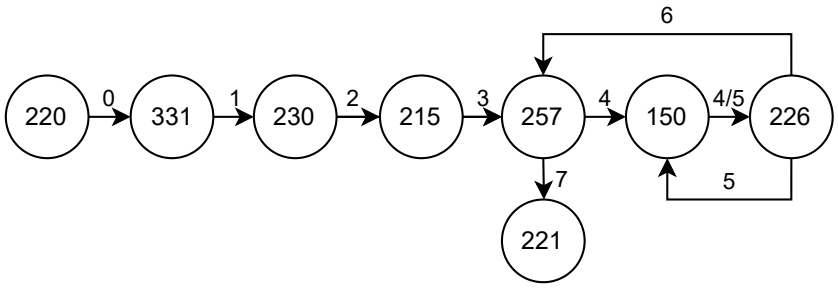

Figure 2.4: AFLNet implemented protocol state machine (IPSM) example

In addition to updating the IPSM, AFLNet dynamically tracks target state information as a list of structures, one per state. Listing 2 describes important fields. AFLNet uses these fields to select seeds to fuzz specific states and to heuristically select the best state to fuzz.

After the dry-run phase, fuzzing begins. An iteration of fuzzing comprises state selection, seed selection and fuzzing using that seed. Figure 2.5 illustrates the process. Using the IPSM and associated state data, AFLNet decides which state to fuzz. In this example, FTP state 257 is selected which is associated with the creation or printing of a directory (*MKD* and *PWD* commands). Using the state structure with ID 257, AFLNet selects a seed from

Listing 2 AFLNET target state info struct (partial)

```
typedef struct{
        u32 id; /*server response code*/
        u32 fuzzs; /*total fuzz which visit this state*/
        u32 paths_discovered; /*total interesting fuzz
                                found while fuzzing this state*/
        u32 selected_times; /*# of fuzzing rounds this
                              state has been selected*/
        u32 score; /*how favoured this state is for selection*/
        void **seeds; /*points to seeds reaching this state;
                        can cast to queue_entry*/
} state_info_t;
```

the corpus that traverses that state. The seed selected is parsed to identify the first message sent once in that state. The message is mutated to create fuzz and sent to the target. Compile-time instrumentation provides code coverage feedback and response packet parsing provides the state sequence. If either of these feedbacks determine the fuzz is interesting, it is added to the corpus as a seed and a corresponding queue entry is made.

AFLNET supports three state selection methods; random, round-robin, and *FAVOR*. The *FAVOR* mode will conduct round-robin for the first five state selections then try to pick the best state to fuzz using a score assigned based on equation (2.1).

$$score = ceil(1000 * pow(2, -log10(log10(fuzzs + 1) * selected\_times + 1)) * pow(2, log(paths\_discovered + 1))) \tag{2.1}$$

fuzzs and *selected_times* are used to measure the rarity of a state within this fuzzing session, while *paths_discovered* measures the potential for a state to find new code paths. Score then is a function of its potential for new code path discovery discounted by how much it's been selected and fuzzed so far. A state is selected with a probability proportional to its share of the sum of scores across all states.

For seed selection, AFLNET also supports three modes; random, round-robin and *FAVOR*. The *FAVOR* mode will conduct round-robin until the state selected has at least 10 seeds reaching it, then start favouring seeds created while the currently select state was being fuzzed, is an initial seed or has not yet been fuzzed for this state. State and seed selection only apply if state

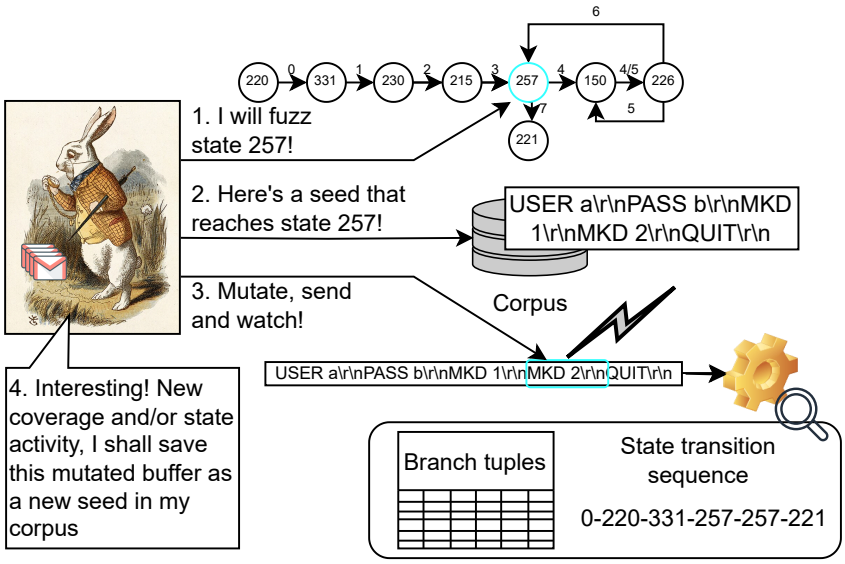

Figure 2.5: AFLNet fuzzing illustration

feedback is enabled, otherwise AFLNet does not model state, and selects seeds as AFL does, based on the order of the queue entries with preference for seeds that have not yet been fuzzed. In the case state feedback is off, the seed is still parsed into messages, but the portion of the seed that is mutated is randomly selected; as little as one message up to the entire seed.

Algorithm 1 line 10 is where the selected seed and state results in fuzz being created and sent to the target. Most of this research is implemented in this part of AFLNet, so we will look at it in detail. Algorithm 2 describes the AFLNet fuzzing function. There are three segments to this function; seed parsing, the havoc stage and the splice stage. Each of these will be explained in detail.

First, the selected seed (message sequence M) is parsed into a prefix, candidate and suffix sequence. This is done by converting the seed into a linked list of messages and assigning pointers to the messages corresponding to:

- messages from the start of the seed up to and including the message that moves the target into the desired state for fuzzing;
- messages sent while the target remains in that state. This forms the candidate buffer which will be mutated; and
- messages after the target leaves the desired state.

Algorithm 2 AFLNet fuzzing function

Input : Target Program P, Message sequence M, Target state s, *Corpus*, *IPSM*

```
M_1, M_2, M_3 ← PARSE(M, s) ▷ 1 - messages to place program in state s, 2 - messages up to change in state, 3 - remaining messages
in_buf ← M_2
orig_buf ← in_buf
out_buf ← COPY(in_buf)

HAVOC_STAGE:
for i from 1 to energy(M) do
    stacks ← RANDOM(128)
    for j from 1 to stacks do
        mutation ← RANDOM(21)          ▷ 17 if region mutations disabled
        out_buf ← MUTATE(out_buf, mutation)
    COMMONFUZZSTUFF(out_buf, Corpus, Bitmap, P, IPSM)
    out_buf ← COPY(in_buf)             ▷ restore out buffer

SPLICE_STAGE:
if splice_count++ <15 then
    in_buf ← orig_buf                  ▷ restore in buffer
    splice_target ← PICKTARGET()
    new_buf ← SPLICE(splice_target, in_buf)
    in_buf ← new_buf
    out_buf ← COPY(in_buf)
    goto: HAVOC_STAGE
splice_count ← 0
```

Figure 2.6 shows the parsing of a seed for selected state 257. To find the candidate sequence, AFLNet looks at the state sequence traversed by the target up to each region in the seed to find the first message after the target is in the desired state and how many more regions are sent before a new state is encountered. This region-specific state sequence is stored in the region struct. Since the *MKD* command moves the target to state 257, the following message is the start of the candidate sequence. As implemented, AFLNet considers looping states to be 'new states' and so the next state 257 reached after sending *MKD 2* terminates the sequence. This implementation means the candidate sequence is usually a single message. The message(s) comprising the candidate sequence are then copied twice into byte buffers, of

which three pointers keep a reference:

- *orig_buf* retains the original copy;
- *in_buf* will be re-assigned during the splicing stage (Algorithm 2 lines 13-20) and thus must be restored to the *orig_buf* at the start of each splicing stage; and
- *out_buf* will be modified by the fuzzing loop havoc stage (Algorithm 2 lines 5-12) at the end of which it is restored to *in_buf*.

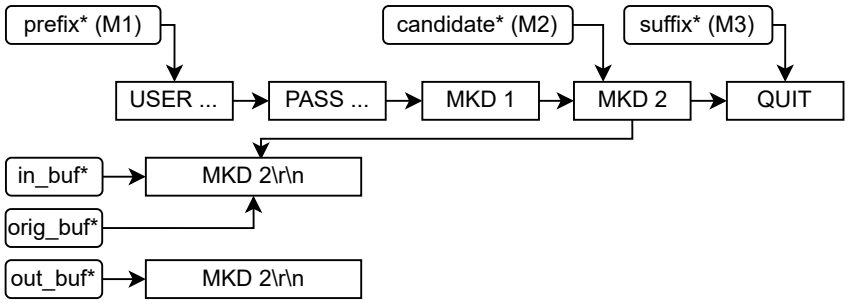

Figure 2.6: AFLNet parsing of seed into prefix, candidate and suffix sequences

On each havoc stage, between 1-128 mutations are applied to the *out_buf* in a stacked fashion. A mutation is applied to the candidate buffer and then another mutation is applied to that same buffer, stacking the mutations atop one another. Table 2.1 shows all the mutation options and indicates the likelihood of that mutation relative to the base-case mutation; random bitflip. Most of these are inherited from AFL except for region mutations where entire messages are involved. Interesting values are kept in a configuration file for byte, word and dword sizes. These include signed or unsigned overflow values, large values, or common size values. Cloned bytes are taken from the buffer itself to then be inserted into or overwritten into the buffer. Dictionary extras are provided as a newline separated list of words when the fuzzer is first launched. These dictionaries are ideally protocol specific keywords or common fields.

Once one havoc stage is completed, AFLNet enters the splicing stage. The splicing stage is essentially an attempt to further exhaust achievable code coverage using the selected seed by combining it with portions of other seeds and repeating the havoc stage. Figure 2.7 illustrates a splicing mutation. First, a splicing target is selected. This is a random seed other than the currently selected seed and is at least 2 bytes long. The fuzzer scans both

Table 2.1: AFLNet Mutations (#x likelihood relative to bit random bit flip)

Fuzzer	Mutation
AFL	Flip a random bit
AFL	Set byte (1x), word(1x) or dword(1x) to interesting value (random endianess)
AFL	Add random value to byte(1x), word(1x) or dword(1x) (random endianess)
AFL	Subtract random value from byte(1x), word(1x) or dword(1x) (random endianess)
AFL	Set random byte to random value (1x).
AFL	Delete bytes (2x).
AFL	Overwrite bytes with constant bytes (0.25x), cloned bytes (0.75x), or dictionary extra (1x)
AFL	Insert bytes with constant bytes (0.25x), cloned bytes (0.75x), or dictionary extra (1x)
AFL	Splicing. Combine with bytes from other another seed
AFLNet	Replace out buffer with random new region (1x)
AFLNet	Prepend a random region to out buffer (1x)
AFLNet	Append a random region to out buffer (1x)
AFLNet	Duplicate entire out buffer (1x)

the splice target and *in_buf*, noting the positions of the first and last differing bytes; shown in bold red. Next, a position between the first and last differing byte is randomly selected. A new buffer is created combining *in_buf* up to but not including the splice position and the splice target from the splice position to its end. *in_buf* is then pointed to this new buffer and *out_buf* initialized to its value to start a new havoc stage. This cycle of splice→havoc is repeated 15 times by default before the fuzzing function ends and AFLNet returns to pick a new state to fuzz.

With 3 out of 21 mutations adding messages and up to 128 mutations being applied to the same buffer, the mutated candidate buffer (*out_buf*) can become very long. To mitigate the negative impact to fuzzing speed that sending many messages would have, AFLNet constrains the mutated candidate buffer to at most n + 1 messages where n is the number of messages in the largest initial seed. This constraint is implemented in the *commonFuzzStuff()* function on line 11 in Algorithm 2. In this function, the *out_buf* is parsed into individual regions and then inserted into the linked list of messages formed at the start of the fuzzing round, replacing the message(s) originally forming the candidate

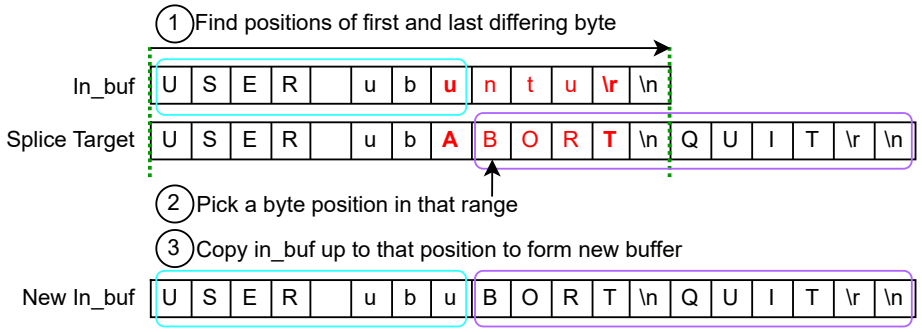

Figure 2.7: AFLNET splicing

sequence. Once n messages have been inserted, AFLNET forms one final message containing the entire remaining buffer, message terminators included. Figure 2.8 illustrates a buffer undergoing two region prepend mutations, one region append mutation and a duplicate mutation. The buffer is then inserted into the linked list of messages, respecting the max message constraint of five messages, and so the sixth message is a combination of all three remaining messages.

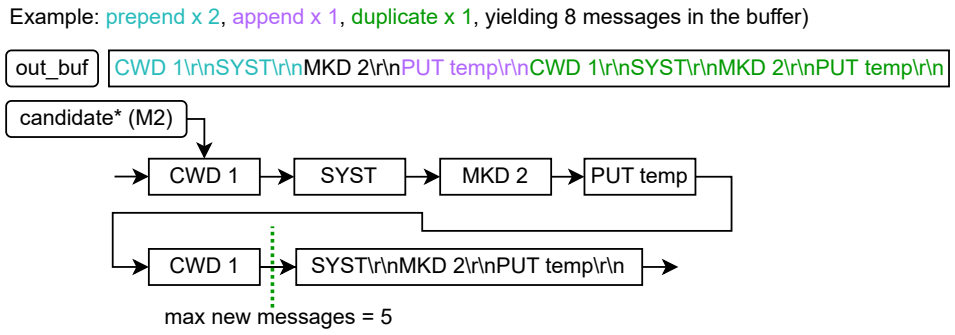

Figure 2.8: AFLNET mutation illustration with max message constraint

From this updated linked list, the *commonFuzzStuff()* function sends the fuzz to the target message by message; watching for new coverage or state feedback. Interesting seeds are added to the corpus to seed future fuzzing rounds and crashes or hangs are saved for future replaying. To aid in the

replaying, AFLNet saves the fuzz in a *replay* format which records the length of each message as a 4 byte value preceding each message. Figure 2.9 shows the ASCII and byte representation of a two message fuzz alongside its replay format.

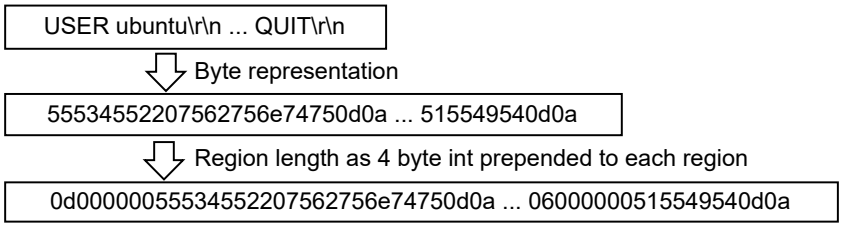

Figure 2.9: AFLNet replay format

AFLNet's features can be individually disabled or enabled using switches when the *afl-fuzz* binary is executed. This research uses these switches to conduct an ablation study of AFLNet's state feedback features. The switches relevant to this research are:

- **-R**: enable region mutations;
- **-E**: enable state feedback. When set, the fuzzer will model target state, select states to fuzz and state-specific seeds. Seeds will be parsed in order to mutate only the candidate sequence;
- **-q**: specify state selection algorithm among round-robin, random and favor. Defaults to round-robin if not specified; and
- **-s**: specify seed selection algorithm among round-robin, random and favor. Defaults to random if not specified.

2.2.2 Profuzzbench

AFLNet's main author, Van-Thuan Pham, partnered with Roberto Natella, author of AFLNet-based fuzzer StateAFL, to create a protocol fuzzing benchmark named ProFuzzBench [13]. It comprises 13 targets, which implement 10 different protocols. Targets represent a breadth of protocol state types and are related to previous academic and non-academic fuzzing work. Table 2.2 shows the 13 targets supported by ProFuzzBench.

The benchmark workflow involves build, execute and analyze phases as shown in Figure 2.10. First a container is built for each target using Docker. The target program and fuzzers being benchmarked are compiled in the container using a specific folder name and structure. The target is compiled

Table 2.2: ProFuzzBench supported targets

Protocol	Target	Description	Protocol States
RTSP	Live555	Real-time media streaming	Streaming progress
SMTP	Exim	Email transmission	Message queue, session progress
FTP	[Pro|Pure|B]-FTPd, LightFTP	File transfer	CWD, session flags, session progress
SSH	OpenSSH	Secure remote shell	User auth. progress, session configuration
TLS	OpenSSL	Secure socket connection	User auth. progress, session configuration
DTLS	TinyDTLS	Secure datagram communication	User auth. progress, session configuration
DNS	DNSmasq	Network domain names	Cached DNS records
SIP	Kamailio	Signaling protocol for real-time sessions	User registrations, session progress
DAAP	Forked-DAAPd	HTTP-based audio library streaming	Streaming progress, playlist
DICOM	Dcmtk	Image retrieval	Progress of the session

twice, once for fuzzing with ASAN enabled and a second time without ASAN for coverage analysis using *gcov*. Files necessary to execute fuzzing are copied into the container at this step, such as initial seeds, clean-up scripts, and shell commands to run fuzzing and coverage programs.

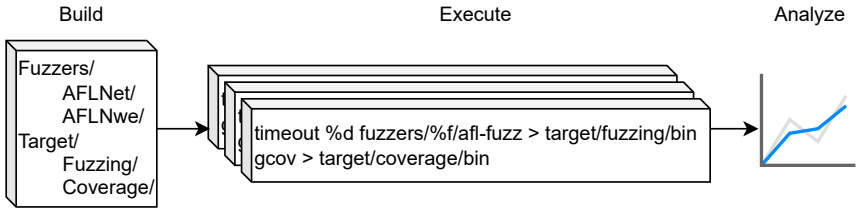

Figure 2.10: ProFuzzBench benchmark phases

To execute experiments, bash scripts are used. At their core, these scripts

spin up containers, run the fuzzing and coverage scripts in the containers and extract the output data onto the host before stopping the containers. An example command to execute an experiment is shown in Listing 3. This experiment launches 4 identical containers, named *lightftp*, and runs AFLNet against the target within the container environment using the flags in quotes. The fuzzing portion of the experiment lasts 86400 seconds. Once fuzzing is complete, ProFuzzBench calculates the coverage achieved over time throughout the fuzzing campaign. The coverage script runs seeds from the fuzzer corpus against the gcov-compiled target. For each 5 seeds sent, *gcov* is run and branch and line coverage information are recorded to disk. The last modified timestamp of the seed files sent are recorded to indicate at what time that coverage is achieved. This captures a time series of line and branch coverage achieved by the fuzzer. When the coverage portion is complete, recorded data is transferred out of the container into the designated host folder (*results-lightftp* in this case) using the designated filename (*out-aflnet* in this case).

Listing 3 Example ProFuzzBench experiment command

```
profuzzbench_exec_common.sh lightftp 4 results-lightftp \
        aflnet out-aflnet "-P FTP -D 10000 -q 3 -s 3 -E -K \
        -m none" 86400 5
```

ProFuzzBench includes analysis scripts to consolidate the experiment coverage outputs into a comma separated value file (CSV) and also to produce plots of average coverage over time. The average coverage is of all runs at each timestep. Figure 2.11 shows an example output for experiments using AFLNet and AFLNwe. Note these are average coverages which, as later discussed in Section 2.3, is not best-practice in fuzzer evaluation.

Though ProFuzzBench receives limited maintenance, users continue to contribute improvements and it is the standard benchmark used for AFLNet-related research.

2.3 Fuzzer Evaluation

Fuzzing is stochastic, meaning determining whether Fuzzer A is better than Fuzzer B, and by how much, requires following certain principles. It is recommended fuzzing experiments last at least 24 hours. Klees *et al.* observed that fuzzing performance varies throughout an experiment and that fuzzing for 24-hours is enough for initial false trends to reverse [29]. This recommendation

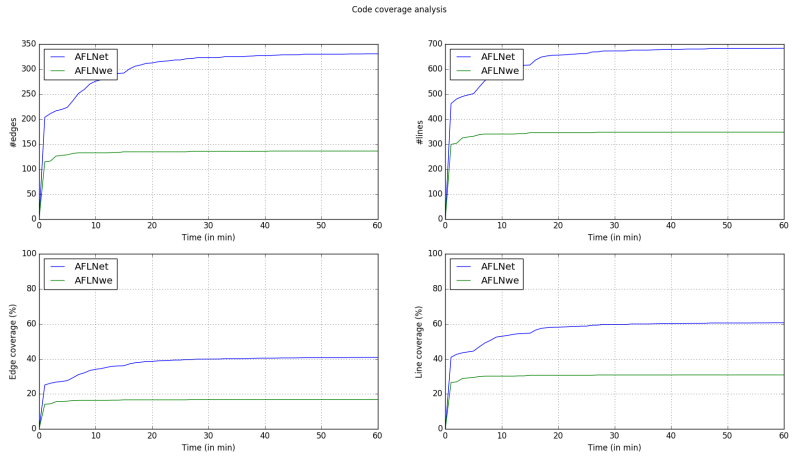

Figure 2.11: ProFuzzBench plotting script output showing coverage over time as absolute values (top) and percentages (bottom)[28]

aligns well with research by Böhme *et al.* that determined code coverage to bug finding agreement improves significantly up to 12 hours of fuzzing [14].

It is recommended that experiments are repeated at least ten times. There are two reasons. First, at least ten data samples are required to accurately complete statistical tests [17]. Second, ten data samples allows us to compare median fuzzer performance. Median fuzzer performance is preferred over average performance due to fuzzing's substantial performance difference run to run [29]. Outliers, very good or very poor runs, skew the average.

There are two main ways to measure fuzzer performance; code coverage achieved and bugs found. Code coverage is easy to measure with standard tools and can be necessary when no bugs are found. The idea that code must be executed in order for a bug to be found is intuitive. Research supports that code coverage is a good proxy for measuring bug finding ability for fuzzers [14]. Specifically, code coverage is strongly correlated to, and moderately agrees with bug finding. It remains that code coverage is not the true objective of fuzzing, which is finding bugs. Research on fuzzer evaluation recommends bug finding is best measured after-the-fact by triaging the crashing testcases found by the fuzzers and to perform detailed bug deduplication; a common occurrence with fuzzing [17]. This recommendation is time-consuming to implement, so many research works limit their evaluation to code coverage.

It is commonplace to report and assess only branch coverage as it better

captures the ability to explore new portions of code and meet constraints. *gcov* considers a branch as a transition between basic blocks in a control flow graph of the code. As expected this includes conditional programing structures, loops, and boolean conditional statements. It also includes exception handling, initialization/destruction of static code elements, and additional ones added from compiler generated code. Though *gcov* has features to reduce unreachable branches, 100% branch coverage is usually impossible [30]. In this research, when we refer to final branch coverage, we mean the total number of branches that the fuzzer succeeded in reaching over the entire session.

When comparing fuzzers using code coverage it is recommended that statistical tests, such as the Mann-Whitney-U test [31], be used to demonstrate whether the coverage difference is statistically significant and that the difference be quantified (for example: Vargha and Delaney's $\hat{A}_{12}$ effect size [32]).

How much of a statistically significant coverage increase is needed to improve protocol fuzzing? Previous research on coverage as a measure of test-suite effectiveness suggests a small increase of coverage, in one case 4%, can result in many bugs found [33]. Additional research on fuzzer evaluation indicates this is due to initial seeds giving the fuzzer easy access to many code branches early in the session [14]. For example, most branches found in their 23-hour experiments were covered in the first 15 minutes. Thus, this research considers a statistically significant coverage improvement of any size to mean the particular configuration or change has improved protocol fuzzing.

Ideally, multiple broadly representative fuzzing targets are used, and all experiments repeated for different seed sets [29]. When multiple separable design features are introduced in the same research work, an ablation study is recommended [17]. Ablation studies seek to understand the role individual components in a system have by removing other components. This allows research to look at improving specific components within fuzzing and build on previous research.

2.4 Related Work

Related to this research includes AFLNet-based research that looks to modify its state feedback features, studies that measure the code coverage value of state feedback in AFLNet, and a fuzzer that added syntax-awareness to AFLNet's mutation process.

2.4.1 Alternative State Feedback

Research has looked at replacing AFLNet's response code-based state representation with more fine-grained alternatives. StateAFL [10] models target state based on a fuzzy hash of long-lived memory regions. States are otherwise selected and fuzzed similar to AFLNet. Code is injected at compile-time to track memory allocations and to take memory snapshots when network exchanges between the fuzzer and target occur. This state representation method eliminates the need to extract response codes from target responses, making the adoption of new targets easier and supporting targets without response codes. Fuzzing new targets for the first time is likely to find bugs, making this feature important [34]. StateAFL's code has been open-sourced. They find this new state representation generally has little to no impact on branch coverage achieved relative to AFLNet, with the exception of PureFTPd and TinyDTLS which show a 2-3% improvement.

NSFuzz [3] models target state based on state variables in the application source. The authors created static analysis tools to help find state variables, which are then manually annotated. Those annotations guide compile-time instrumentation which monitors those state variables. Their premise is that most network application targets have a single main loop which receives and handles new connections according to some state variables. When a target processes a message, a hash is taken of an array of state variables to represent the target state. NSFuzz additionally implements techniques to speed up fuzzing, but they also tested a version of their fuzzer with no speed-ups to understand the contribution of the new state representation alone. They find the variant of the fuzzer with only the new state repesentation averages a 2.11% branch coverage improvement across ProFuzzBench targets, though this average is skewed by 2 targets showing a large coverage increase while 5 targets showed a coverage decrease. This research team also ran experiments with StateAFL and found an average coverage decrease of 3.38%. Overall results varied depending on the target's interaction with the state representation's strengths or weaknesses. NSFuzz is not open-source.

On the topic of target state selection, AFLNet-Legion [11] implemented a new state selection algorithm in AFLNet called Legion which is based on Monte Carlo tree search. They found that the new algorithm failed to improve coverage in a statistically significant way relative to AFLNet's favoured state selection mode. They relate this result to two limitations in AFLNet; low fuzzing throughput and AFLNet generating bad fuzz rejected by the server. AFLNet-Legion's code has been open-sourced.

SMGFuzz [12] proposes that state selection should occur after seed se-

lection, on the grounds that every state transition exists within the coverage bitmap from which seeds are selected. In addition, SMGFuzz introduces more efficient ways to store state information (as a map) and construct message sequences (no longer than the first protocol-specific termination command, such as QUIT in FTP). The average executions for one particular target increased by 40% through this semantic addition. On average, coverage improved by 12.48% across eleven ProFuzzBench targets. This average is skewed by one strong outlier. When the outlier is removed, the average code coverage change is negative. SMGFuzz has not yet been published and the code not currently publicly available.

2.4.2 Value of State Feedback

Days before this research was proposed, a preprint became available wherein the authors present AFLNet's process in detail and investigate the code coverage value added by the state feedback and seed-selection strategies in AFLNet [15]. It was published in April 2025, its source code published in May 2025, and is co-authored by the creator of AFLNet, Van-Thuan Pham. The research shares a similar goal and motivation to the ablation study conducted in this present research. The authors acknowledge an evaluation of each AFLNet component was lacking and that they aimed to understand the contribution state feedback makes towards overall fuzzer performance.

Whereas this research uses existing AFLNet switches to define different configurations, Meng *et al.* modified the *alf-fuzz* source to more neatly isolate features while holding others constant. To discover the value of state feedback they compiled separate AFLNet *afl-fuzz* binaries that isolate code-only and code+state feedback while retaining an identical seed selection approach. Using just existing switches this is not possible, because when state feedback is enabled, purely AFL-style seed selection is not available, rather the closest is FAVOR mode, where a mix of round-robin and AFL-style seed selection is used with state heuristic considerations. They ran 10x24 hour experiments using ProFuzzBench and reported average results. They find that while additional state feedback can slightly improve code coverage for most subjects, this improvement is not statistically significant. On average the code coverage improvement is of +0.01%, with targets seeing changes from -2.42% up to +1.38%.

Next, the authors researched the performance difference of selecting seeds based on the queue order as is done by AFL and AFLNet FAVOR mode (a blend of queue order and state heuristics). They found AFLNet FAVOR mode performed worse than queue order, achieving -2.04% coverage on average

across all ProFuzzBench targets (minus OpenSSL) and -5.07% on average for FTP targets. This paper concludes that perhaps response codes are not a good representation of target state and that would explain why state feedback contributes such limited code coverage value. It is relevant to mention here their definition of state feedback differs from our's; where we include state-heuristic based seed selection as part of state feedback.

2.4.3 Syntax-awareness in AFLNet

ChatAFL is a fuzzer derived from AFLNet which uses a large language model (LLM) to apply three guidance strategies to protocol fuzzing; syntax-aware mutations, enrichment of initial seeds and breaking out of coverage plateaus [6].

Regarding syntax-aware mutations, the authors find an LLM can reliably provide a protocol's set of supported messages (its syntax) while respecting a specific response format that identifies variable and fixed fields. To achieve this, few-shot learning with two shots is used to inform the LLM of the desired format while preventing it from adhering too closely to the example. The LLM is then queried multiple times for the protocol syntax. Messages reliably found in the LLM response form the protocol's syntax for guiding mutations. In their case study, using the Real Time Streaming Protocol (RTSP), the LLM rarely generated random messages and in one case would sometimes omit an optional field, but the majority response matched the ground-truth. These message templates are then placed in a corpus that is used to constrain mutations to variable fields within a message type.

It is relevant to mention here the different depths of syntax. First we can ensure the message respects the correct form (called a template by ChatAFL) and then ensure the fields contain the right type of information. Figure 2.12 illustrates this. The *USER* FTP command expects the form shown at point one. The username field meanwhile is expected to be a string comprising ASCII characters except the carriage-return and linefeed characters. ChatAFL limits its syntax to the template-level. Yu *et al.* in their protocol fuzzing tool named SGPFuzzer, explain being too strictly aligned with protocol syntax could mean no bugs are triggered and that rather we need enough validity to get past initial parsing, but some variability to trigger bugs [24]. Limiting syntax to the template-level is one way of achieving this balance.

Before the stacked mutation loop is entered, the *out_buf* is searched against the templates. If a match is found, the variable fields are tracked by byte position in an array of ranges (start and length of each byte range). These byte ranges are mutated, while the rest of the buffer is left untouched. Figure 2.13

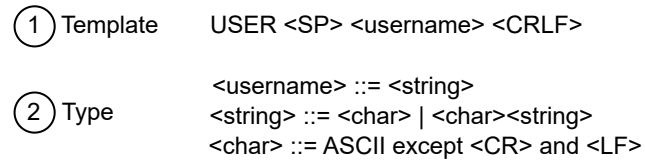

Figure 2.12: FTP syntax depths

shows a mutable byte range being parsed from the buffer *MKD 1\r\n*. The syntax corpus is queried and the buffer matches the FTP MKD message type. The template has a variable field after the "MKD " header up to the message terminator $\backslash r \backslash n$. In this example, this results in a single range starting at byte position 4 with a length of 1.

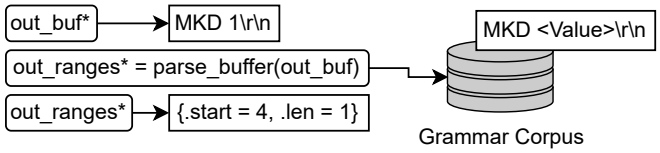

Figure 2.13: ChatAFL syntax-awareness

When performing stacked mutations, a mutable range is randomly selected to have the mutation applied to. A mutation is then randomly selected. Each mutation first checks that the mutable range is sufficiently long for the chosen mutation, if not then this stack is skipped and the loop continues to the next stack, where a new random mutable range and mutation are chosen. How each mutation is made syntax-aware is shown in Table 2.3. For AFL mutations, except for splicing, syntax-awareness is simply constraining the target of the mutation to the mutable range. ChatAFL does not modify the splicing stage. In the case of the replace buffer mutation, the new buffer is parsed to obtain its mutable ranges. If adding a new region, the existing regions must be shifted based on the size of the region added. Note ChatAFL does not add the mutable ranges of a newly added region to the list of mutable ranges in the buffer. The duplication mutation is changed to only duplicate the selected mutable range. In this case, the ranges are also replicated.

Like SGPFuzzer[24], ChatAFL's authors agree that some amount of syntactically invalid fuzz can be useful. Before entering the mutation loop a decision is made to use syntax-awareness (exploit) or not (explore). These terms usually refer to a trade-off between acquiring new knowledge about the

Table 2.3: ChatAFL syntax-awareness added to AFLNet Mutations. #x likelihood relative to bit random bit flip. Strikethrough indicates changes made by ChatAFL.

Fuzzer	Mutation	ChatAFL Syntax Awareness
AFL	Flip a random bit.	Apply to mutable byte ranges only
AFL	Set byte (1x), word(1x) or dword(1x) to interesting value (random endianess)	Apply to mutable byte ranges only
AFL	Add random value to byte(1x), word(1x) or dword(1x) (random endianess)	Apply to mutable byte ranges only
AFL	Subtract random value from byte(1x), word(1x) or dword(1x) (random endianess)	Apply to mutable byte ranges only
AFL	Set byte to random value (1x)	Apply to mutable byte ranges only
AFL	Delete bytes (2x).	Apply to mutable byte ranges only Adjust mutable ranges accordingly
AFL	Overwrite bytes with constant bytes (0.25x), cloned bytes (0.75x), or dictionary extra (1x)	Overwrite mutable byte ranges only. Clone from entire buffer
AFL	Insert bytes with constant bytes (0.25x), cloned bytes (0.75x), or dictionary extra (1x)	Insert in mutable byte ranges only. Clone from entire buffer. Shift ranges accordingly
AFL	Splicing (after havoc stage)	-
AFLNet	Replace out buffer with random new region ~~(1x)~~(2x)	Replace mutable ranges with mutable ranges of new region
AFLNet	Prepend a random region to out buffer ~~(1x)~~(2x)	Shift existing mutable ranges
AFLNet	Append a random region to out buffer ~~(1x)~~(2x)	-
~~AFLNet~~ ChatAFL	~~Duplicate entire out buffer (1x)~~ Duplicate selected mutable range (2x)	Add new mutable range and shift existing

environment (exploration) and using that knowledge to gain some form of energy (exploitation).

When exploring, ChatAFL creates a single mutable range comprising the entire buffer. For their experiments ChatAFL used a 50% explore-exploit ratio. Three design decisions make the actual ratio unclear. First, though the

decision for a round of fuzzing may be to exploit, if the parsing function fails to match the buffer to a template, it resorts to exploration. Second, when the replace-region mutation is performed it always attempts to exploit (again resorting to explore if no match is found). Third, since the splicing stage is not changed, it is likely the stage seeds future havoc stages with syntactically invalid seeds, thus causing more exploration.

These design decisions hinder a clear understanding of the explore-exploit tradeoff when applying syntax-awareness in AFLNet. Hence, in the research conducted herein we undertake a different design to adding syntax-awareness to AFLNet, while retaining ChatAFL's granularity of syntax and mutable range approach.

Tested against six ProFuzzBench targets, they found syntax-aware mutations led to an average coverage improvement of 3.04% and an average coverage speed-up of 2.02x relative to AFLNet. Among the six targets were two FTP targets, ProFTPd and PureFTPd. Though the aim of this research is not to improve on syntax-aware mutation, a comparison between ChatAFL and this research is warranted and discussed in Appendix A.

3 Methodology and Design

This chapter presents the methodology and design used to conduct this research. Recall the aim of this research is to determine how adding syntax-awareness to AFLNet's mutation process changes the branch coverage impact of AFLNet state feedback features. This research achieves its aim by measuring the impact of state feedback features before and after adding syntax awareness.

First, an ablation study of separable AFLNet features is completed. This provides the code coverage impact of each separable AFLNet feature relative to a baseline configuration (feature impact A). AFLNet is then extended with syntax-awareness to create syntactically correct fuzz. We call our syntax-aware AFLNet derivative AFLNet-Packmute. A verification phase confirms that created fuzz are indeed syntactically correct. Lastly, a validation phase measures the impact to code coverage between a baseline AFLNet-Packmute and configurations enabling each separable design feature (feature impact B). The difference between feature impact A and feature impact B allows us to achieve the aim of the research.

3.1 Methodology

Initially we were interested in the idea of sharing dynamic state information among parallel protocol fuzzers to improve favoured state selection, however in preliminary experimentation we found state selection did not make any difference in code coverage achieved. This conclusion is also reached in 2022 by a doctoral student of AFLNet's creator [11]. This led us to observe no one had yet investigated the code coverage impact of each AFLNet component. Conveniently, AFLNet's author included command-line switches to configure various features within the tool and so fuzzer configurations could be defined based on those switches to measure the code coverage contribution of each feature.

Our hypothesis was that state feedback features lack effectiveness due to AFLNet having no knowledge of protocol syntax. The AFLNet mutation process presented in Chapter 2 has a high likelihood of making the fuzz syntactically invalid, having on average 64 stacked mutations applied anywhere in a protocol message. Syntactically invalid fuzz would then be rejected early on in a networking application's logic, failing to effectively interact with its state. The objective then is to add syntax-awareness into AFLNet.

Although manually coding protocol parsers is effortful, we were convinced there must be existing tools that have gone through that effort (Wireshark exists after all). Searching for tools that give structured access to application layer protocol data and have a language compatibility with AFLNet (C) we found PcapPlusPlus [35]. It is written in C++ and thus could be compiled alongside *afl-fuzz*, is actively maintained, has very fast reported performance, and supports the parsing of fourteen different application layer protocols; giving access to each field in a given protocol packet. Next we looked to ChatAFL, which is the first tool to add syntax-awareness into AFLNet's mutation process. It's use of mutable byte ranges to constrain mutations while preserving syntax is elegant and simple, so we opted to use that but also recognized design decisions which we found inefficient or unclear (particularly regarding the exploit-explore trade-off). The design of this research looks to address those observations while using existing tooling with syntax-awareness, vice an LLM derived syntax corpus.

If an ablation study has now been published [15], why did we proceed with the ablation study in this research? We believe that using an unmodified AFLNet in the ablation study will further validate the work by Meng *et al.* and contribute additional insights into the code coverage value of state selection algorithms and region mutations. Furthermore, the ablation study in this research was needed for development and validation phases and the Meng *et al.* paper was not published in time for this research. The authors omitted one target, OpenSSL, which our ablation study includes. Lastly, unlike the existing ablation study we include state-informed seed selection as part of state feedback in AFLNet.

3.2 AFLNet Ablation Study

The goal of this phase is to measure the change in branch coverage resulting from enabling specific AFLNet features. Measuring the contribution of a particular system component is called an ablation study. This is the feature's code coverage impact. If a feature has little to no impact on code coverage,

there is a case that the feature needs to be re-designed to be useful, simply eliminated, or some other change made to AFLNet to make it impactful.

An ablation study of AFLNet components is conducted using ProFuzzBench against all its supported targets (listed in Table 2.2). The latest version of AFLNet at the time of this research is used, commit ID 6d86ca0. ProFuzzBench by default uses an old version. Features are enabled one at a time, using AFLNet's existing switches to toggle or set specific features. The fuzzing configurations are shown in Table 3.1. There is one baseline configuration, four configurations capturing various combinations of state feedback features, and five configurations adding region mutations to each.

Table 3.1: Ablation study AFLNet configurations

Name	Region Mutations	State Awareness	Seed Selection	State Selection
BASELINE			N/A	N/A
BASELINE-RGNS	✓		N/A	N/A
STFL-RND		✓	Random	Random
STFL-RND-RGNS	✓	✓	Random	Random
STFL-FVR-SD		✓	Favor	Random
STFL-FVR-SD-RGNS	✓	✓	Favor	Random
STFL-FVR-ST		✓	Random	Favor
STFL-FVR-ST-RGNS	✓	✓	Random	Favor
STFL-FULL		✓	Favor	Favor
STFL-FULL-RGNS	✓	✓	Favor	Favor

As per fuzzer evaluation best-practice outlined in Section 2.3, each configuration is evaluated over ten separate fuzzing sessions lasting 24 hours. A BASELINE experiment execution against the LightFTP target looks like Listing 4.

Listing 4 ProFuzzBench BASELINE experiment command

```
profuzzbench_exec_common.sh lightftp 10 results-lightftp \
        aflnet out-aflnet "-P FTP -D 10000 -K -m none" 86400 5
```

Ten fuzzing container instances are launched in parallel, running AFLNet with the specified flags against the LightFTP application for 86,400 seconds (24 hours) and then measuring coverage for every five testcases in the corpus. The *-K* switch tells AFLNet to send a process termination signal to the target between fuzz. The *-m none* switch tells AFLNet there is no memory limit for the spawned target processes. This is necessary due to using ASAN-enabled 64 bit binaries which allocate huge quantities of memory; though they never actually use it. The *-D* switch tells AFLNet to wait 10,000 microseconds before sending the fuzz to allow the target application to complete initialization. Each fuzzer configuration keeps these essential switches and then adds others to enable specific features or for specific protocols. The ablation study uses the target-specific timeouts included in default ProFuzzBench exepriment commands.

The use of AFLNet's existing switches to define each configuration makes this ablation study simple to execute, needing to compile only one fuzzer per target. However, we are then limited to analyzing the impact of these functional features which may not perfectly isolate AFLNet's design decisions. This limitation is present when we measure the impact of basic state feedback, where a state-aware seed selection approach must be used. Seed selection is also done AFLNet with state feedback disabled as well. Hence, for basic state feedback there are two influences at play; the change in seed selection and the addition of state modeling and selection. Fortunately, the coverage impact of state-heuristic based seed selection relative to plain AFLNet using queue order has measured by Meng *et al.*, allowing us to understand each of these 2 components at play when measuring basic state feedback.

This phase is concerned with observing the effectiveness of AFLNet features; region mutations, basic state feedback (parsing seed into candidate sequence, interpreting server response codes as states) and each FAVOR-mode seed and state selection. Effectiveness of a feature is measured by the percent change in final median branch coverage observed when a feature is enabled relative to the parent configuration.

Parent configurations have one less feature enabled than their child configuration, allowing us to measure the impact of enabling one specific feature. Figure 3.1 shows the parent:child hierarchy for all ten fuzzer configurations included in the ablation study. Arrow number 1 isolates the impact of adding state feedback while retaining a seed selection closest to the parent config and random state selection. We call this basic state feedback. This evaluates the coverage value of selecting and fuzzing specific states. STFL-FVR-SD will mutate the messages targeting specific states, while BASELINE will mutate anywhere from 1 message to the entire sequence, not targeting any specific

state. Arrows numbered 2 isolate the impact of favoured state selection. Arrows numbered 3 isolate the impact of favoured seed selection. Purple right-angled arrows indicate measurements that are accessory to the aim of this research; that is, measuring the impact of region mutations. Those results are discussed in Appendix B. Note for each numbered turquoise arrow there are two opportunities to measure the feature impact; once by comparing the configurations without region mutations enabled and once by comparing the configurations with region mutations enabled. These multiple measurements for each feature may help overcome results of limited statistical significance and strengthen our observations. The green semi-circular arrow represents the coverage value of the 3 state feedback features combined. There may be synergies or drawbacks from combining features which cause an effect to performance beyond the feature we wish to measure itself. This research did not try to measure such factors. However, using multiple measurements allowed us see a given feature's impact in the context of different fuzzer configurations, especially with or without region-mutations.

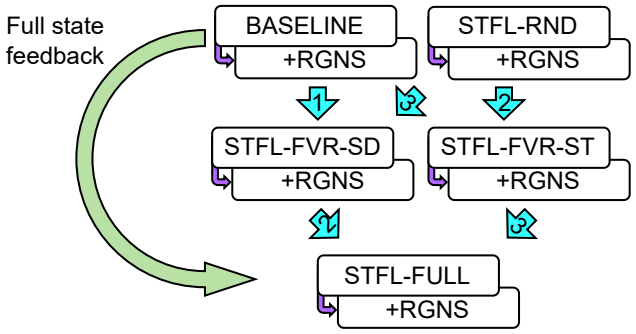

Figure 3.1: Ablation study configuration hierarchy. 1 - basic state feedback. 2 - favoured state selection. 3 - favoured seed selection.

Protocol fuzzing performance is measured by the total branch coverage achieved by the median fuzzer of a given configuration. Coverage is measured by *gcov* as described in Section 2.2.2. Though *gcov* also provides line coverage, only branch coverage will be used as is done in all related works. The percent change in median branch coverage between a child and parent configuration represents the impact or effectiveness of a feature.

As fuzzing is stochastic and we are repeating experiments a limited number of times, statistical significance will determine whether the sets of results (child and parent) belong to different distributions. The Mann-Whitney-U test [31]

with a significant level of 0.05 confirms whether the distributions are different. It is a test which helps reject the null hypothesis that the distributions are the same when the number of samples is small. Though the test does not assume the data is normally distributed it does assume they have a similar shape. The test results in a p-value which indicates the likelihood that the samples are from the same distribution. If the p-value is 5% (0.05) or lower we reject the null hypothesis that they are from the same distribution and deem they are different. Vargha and Delaney's $\hat{A}_{12}$ effect size [32] quantifies the difference. An effect size ranges from 0 to 1 and indicates the chance that a random sample from distribution A is greater than a random sample from distribution B. If the effect size is larger than 0.5 is means distribution A contains larger values, while an effect size under 0.5 indicates distribution B contains larger values. The original paper proposes how far from 0.5 the effect size must be to consider the difference in distributions negligible (0-0.06), small (0.06-0.14), medium (0.14-0.21) or large (0.21+).

Related work in this field usually includes all values in calculating summary statistics, but also includes the statistical significance metrics to indicate how strong the observations are. Secondary measurements are included to draw additional insights; speed-up (how fast fuzzer B reached the coverage of fuzzer A in the event it outperformed A), average executions per second, and fuzzer min/max run code coverage. The output of this phase is the feature impact of the non-syntax-aware AFLNet.

3.3 AFLNet-Packmute

This section outlines the design of a syntax-aware AFLNet variant, named AFLNet-Packmute which is developed for this research. It will later be used to measure the state feature impact of AFLNet when it has a syntax-aware mutation process. This section is organized in four parts;

- Adding syntax-awareness to the AFLNet fuzzing function from an algorithmic perspective;
- Adding syntax-awareness to the AFLNet mutations;
- The structural design of AFLNet-Packmute; and
- The behavioural design of key functions.

Syntax-awareness is limited to ensuring messages adhere to a valid message template for the protocol. Mutations are applied to byte ranges within the mutation buffer which correspond to variable fields in the protocol specification. These are called mutable ranges. As mutations are applied to the candidate buffer, the mutable ranges are updated accordingly. In the event

the fuzzer decides to explore, and not use syntax awareness, a single mutable range spanning the entire buffer is created. This approach to adding syntax-awareness to AFLNet's mutation process is taken from ChatAFL [6]. It neatly adds syntax-awareness to the existing mutation process with minimal changes. Our goal is to add syntax-awareness without making large changes to how AFLNet is creating fuzz.

AFLNet-Packmute does not enforce that mutable ranges respect the expected type of information by the protocol. For example, the FTP command *USER* expects a username field to only contain ASCII characters excluding the carriage-return and line-feed characters. Enforcing that level of syntax-awareness would require changes to the mutation process itself, which is outside the scope of this research.

3.3.1 Fuzzing Loop Syntax-Awareness

Algorithm 3 highlights in red the changes made to the AFLNet fuzzing function to make it syntax-aware. An exploit-explore decision point and mutable range structures have been added. For each copy of the candidate buffer in the fuzzing loop there is a corresponding mutable range structure that tracks where mutable ranges exist in each buffer. These are updated as each mutation is performed. Like the candidate buffer copies, these ranges are named *orig_ranges*, *in_ranges*, *out_ranges*. How the functions contained in Algorithm 3 have been made syntax-aware is covered in the next two subsections. Here we limit ourselves to a high-level view.

On line 5, a decision is made whether to exploit or explore this fuzzing round. If exploiting, the mutable ranges are extracted from the regions comprising the candidate sequence through the *getRanges()* function as shown on line 7 and assigned to *orig_ranges*. If exploring, a single range comprising the entire buffer is created (line 9). These *orig_ranges* are then copied to *in/out_ranges* on lines 10-11. In the stacked mutation loop, individual mutations consider the ranges of the *out_buf* and whether this round is exploiting or exploring (line 17). Between each fuzz sent to the target, the *out_ranges* are reset to match the *in_ranges* (line 20). During splicing, the *in_ranges* are restored to the *orig_ranges* as is the buffer (line 24). If exploiting, a splicing target that contains mutable ranges will be selected (line 25) and the splice will occur between a mutable range in the target to a mutable range in the *in_buf*. Splicing will update the effected *in_ranges* before the buffer and ranges are copied into the *out_* variables.

With this algorithm in mind, let's consider how mutations are made syntax-aware.

Algorithm 3 AFLNet-Packmute Fuzzing Loop

Input : Target Program P, Message sequence M, Target state s, *Corpus*, *IPSM*

1: $M_1, M_2, M_3 \leftarrow$ PARSE(M, s) ▷ *1 - messages to place program in state s, 2 - messages up to change in state, 3 - remaining messages*
2: $in\_buf \leftarrow M_2$
3: $orig\_buf \leftarrow in\_buf$
4: $out\_buf \leftarrow$ COPY(in_buf)
5: $exploit \leftarrow$ RANDOM(100) $< EXPLOIT\_RATE$
6: **if** *exploit* **then**
7: $\quad orig\_ranges \leftarrow$ GETRANGES(M_2)
8: **else**
9: $\quad orig\_ranges \leftarrow$ .starts = 0, .lengths = len(M_2), .num_ranges = 1
10: $in\_ranges \leftarrow$ COPY($orig\_ranges$)
11: $out\_ranges \leftarrow$ COPY(in_ranges)

12: HAVOC_STAGE:
13: **for** *i from 1 to energy(M)* **do**
14: $\quad stacks \leftarrow$ RANDOM(128)
15: $\quad$ **for** *j from 1 to stacks* **do**
16: $\quad\quad mutation \leftarrow$ RANDOM(21) ▷ *17 if region mutations disabled*
17: $\quad\quad out\_buf \leftarrow$ MUTATE(out_buf, *mutation*, out_ranges, $exploit$)
18: $\quad$ COMMONFUZZSTUFF($out\_buf, Corpus, Bitmap, P, IPSM$)
19: $\quad out\_buf \leftarrow$ COPY(in_buf) ▷ *restore out buffer and ranges*
20: $\quad out\_ranges \leftarrow$ COPY(in_ranges)

21: SPLICE_STAGE:
22: **if** *splice_count++ <15* **then**
23: $\quad in\_buf \leftarrow orig\_buf$ ▷ *restore in buffer and ranges*
24: $\quad in\_ranges \leftarrow$ COPY($orig\_ranges$)
25: $\quad splice\_target \leftarrow$ PICKTARGET($exploit$)
26: $\quad new\_buf \leftarrow$ SPLICE($splice\_target, in\_buf, in\_ranges, exploit$)
27: $\quad in\_buf \leftarrow new\_buf$
28: $\quad out\_buf \leftarrow$ COPY(in_buf) ▷ *prep next HAVOC*
29: $\quad out\_ranges \leftarrow$ COPY(in_ranges)
30: $\quad$ goto: HAVOC_STAGE
31: $splice\_count \leftarrow 0$

3.3.2 Mutation Syntax-Awareness

The way ChatAFL added syntax-awareness to each mutation was covered in Section 2.4. AFLNet-Packmute takes a more strict approach, seeking to minimize the breaking of syntax when choosing to exploit. Table 3.2 describes how AFLNet-Packmute applies syntax-awareness to each mutation, and the differences from ChatAFL. Note this pertains to when the fuzzing round is exploiting. When exploring, a single mutable range comprising the entire buffer is kept so that all bytes can be mutated and syntax-awareness constraints are not enforced during mutations.

Region mutations in AFLNet-Packmute differ from ChatAFL in several ways. In the case of replacing the entire buffer with a new region, ChatAFL takes any region, regardless of whether there exist mutable ranges in that region and then proceeds to parse it for mutable ranges (ie apply syntax-awareness). This is done even if the decision is to explore, essentially converting it to exploitation. Since this mutation case on average occurs multiple times each set of stacked mutations, ChatAFL's process often tries to exploit even if their exploit_rate is set to 50%. AFLNet-Packmute ensures the region selected has a mutable range if the decision was to exploit this fuzzing round.

When AFLNet-Packmute prepends or appends regions to the mutation buffer it then adds mutable ranges in those regions to the list of ranges, whereas ChatAFL does not. The splicing stage is not modified for ChatAFL, but we found it introduced syntactically invalid commands when fuzzing and so we also made it syntax-aware. This way, there is a very clear decision to exploit or explore that is carried throughout the fuzzing loop.

Another area ChatAFL's approach introduced invalid fuzz is cloning from anywhere in the mutation buffer, including message terminators. When exploiting, AFLNet-Packmute restricts the cloning of bytes from and to the mutable range to prevent message terminators from being copied. In the case of inserting to a zero-length mutable range, AFLNet-Packmute will pad the inserted bytes as needed to preserve syntax-validity.

Let's illustrate two mutations in detail to get a sense of how the ranges and mutation buffer interact. Figure 3.2 illustrates inserting cloned bytes. Point one shows the initial *out_buf* and *out_ranges* we are working with. The mutation buffer '*USER ubu\r\nLIST\r\n*' consists of two messages, each with one mutable range. For the *USER* command, the mutable range is 'ubu' found in byte positions 5 through 7, inclusive. Thus, the first entry in *out_ranges'* arrays has a start of 5 and a length of 3. The second mutable range, for the *LIST* command, is found at byte position 14, but has length 0. This is because

Table 3.2: AFLNet-Packmute syntax-aware mutations. Mutation likelihood and meaning identical to AFLNet. Mutation descriptions abridged for readability.

Mutation	AFLNet-Packmute Syntax Awareness	Description
Bit flip	Apply to mutable byte ranges only	Same as ChatAFL
Set to interesting	Apply to mutable byte ranges only	Same as ChatAFL
Add	Apply to mutable byte ranges only	Same as ChatAFL
Subtract	Apply to mutable byte ranges only.	Same as ChatAFL
Set to random	Apply to mutable byte ranges only	Same as ChatAFL
Delete	Apply to mutable byte ranges only. Adjust mutable ranges accordingly	Same as ChatAFL
Overwrite	Overwrite mutable byte ranges only. Clone from mutable ranges only	Cloning from mutable ranges only prevents the cloning of message terminators, creating new, broken messages
Insert	Insert in mutable byte ranges only. Pad 0-length mutable ranges. Clone from mutables ranges only. Shift ranges accordingly	Cloning from mutable ranges only prevents the cloning of message terminators, creating new, broken messages. Inserting into 0-length mutable range may require padding to respect the message template
Splicing	Splice from mutable range in target seed to mutable range in buffer. Update ranges as needed	Splicing reduced from entire buffer to only mutable ranges. Not considered by ChatAFL
Replace buffer	Select region with mutable range. Replace mutable ranges with those of new region	Preserved AFLNet mutation likelihood. Added constraint to pick a region with a mutable range so syntax-aware mutations can continue
Prepend to buffer	Shift existing mutable ranges. Add new mutable ranges, if any.	Preserved AFLNet mutation likelihood. Adding mutable range of new region will permit it to be immediately mutated
Append to buffer	Shift existing mutable ranges. Add new mutable ranges, if any.	Preserved AFLNet mutation likelihood. Adding mutable range of new region will permit it to be immediately mutated
Duplicate buffer	Duplicate all mutable ranges	Preserved AFLNet mutation likelihood and implementation

the *LIST* command may or may not have an argument; it is optional. In this

illustration, mutable range 1 is selected for mutation (*range_choice*=1). Since these are zero-indexed arrays, it means the range at byte position 14 will be inserted into with cloned bytes.

Point two shows the parameters that will determine the cloning operation. *clone_rng* is chosen randomly among mutable ranges with length greater than zero, which in this case only leaves one option, the first mutable range (index zero). There is no restriction on cloning from ones-self. *clone_len* is a random value between one and the length of the selected *clone_rng*, in this illustration the *clone_len* is two. Now we know from which range we are cloning to, which range we are cloning from and how many bytes we want to clone. Next we need to know exactly which bytes we are cloning. *clone_from* determines the position from which a continuous set of bytes will be cloned. It is randomly picked among byte positions that allow for the full *clone_len* to be copied from within the *clone_range*. Here, the range we are cloning from starts at position 5, and since we are cloning 2 bytes, we are left with 2 positions to pick from; 5 or 6. The RNG function shown would evaluate to 3 minus 2 plus 1, or *rand(2)* which would return either 1 or 0. This illustration assumes 0 is returned, meaning 2 bytes at position 5 will be cloned; 'ub'. *clone_to* is simply a random position within the range we are mutating (*range_choice*) without regard for space; the buffer will be lengthened if needed.

Point three shows the result after the mutation has been applied. Since we are exploiting and mutating a zero-length range, padding must be added to the mutable range before the cloned bytes to preserve syntax validity. Which bytes to add are provided by Libpackmute, a syntax companion library we wrote as part of AFLNet-Packmute (discussed in Section 3.3.3). The two cloned bytes can then be inserted. Three bytes have been added to the second mutable range and so the *out_ranges* must be updated. The second mutable range needs to be shifted to account for the padding and its length increased by the number of bytes inserted (start goes from 14 to 15 and length goes from 0 to 2). The padding is necessary to conform to the syntax of the *LIST* command when it has an argument.

This second mutation illustration stacks a prepend-region mutation atop the cloning illustration, hence, Figure 3.3 point one matches the values we ended the previous illustration with. Point two shows that a random region from a random seed in the corpus is selected. For the prepend-region mutation the selected region may or may not have mutable ranges regardless of whether we are exploring or exploiting. In this case, the region chosen is '*CWD tmp\r \n*'. Point three shows the selected region prepended to the original *out_buf*. When exploiting, the mutable ranges of the newly added region must be prepended to *out_ranges* which causes the start positions of the existing

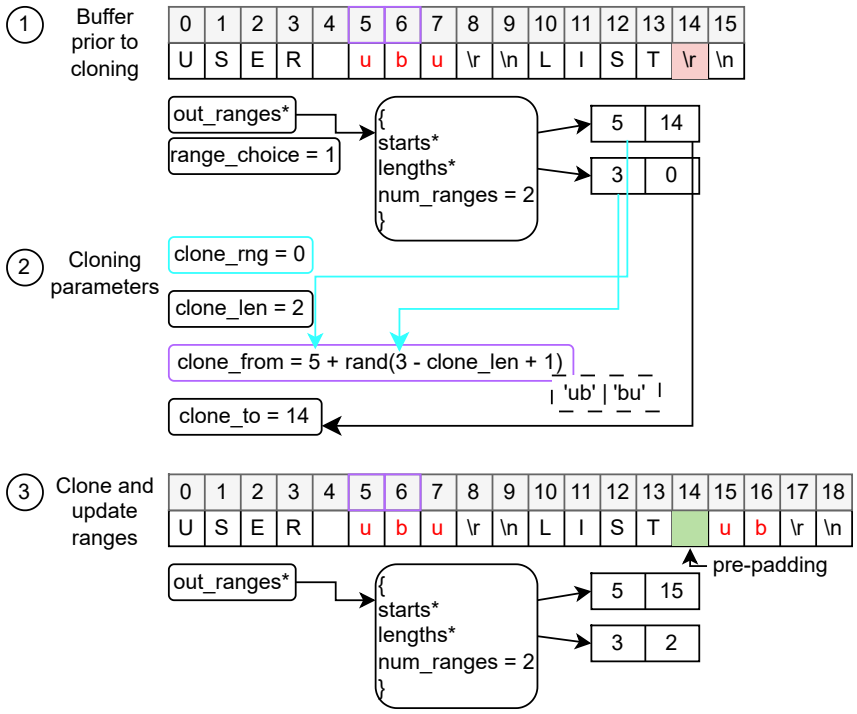

Figure 3.2: Cloning insert mutation illustration (exploit)

mutable ranges to be pushed to the right.

Lines 25 and 26 of Algorithm 3 in Section 3.3.1 are where the splicing operation occurs. A target is picked then the splicing takes place. When exploiting, the splice target is ensured to have at least one mutable range of length two or more. When exploring, any target that is not the current queue entry and is not entirely less than two bytes is acceptable. Once a target is selected it is read from disk into a byte array and the splicing takes place.

Syntax-aware splicing will splice mutable ranges in the *in_buf* and the target, while not effecting the remainder of the *in_buf*. Regular splicing will create a new buffer with a front-half of the old buffer and the back-half of the new target buffer.

We've discussed how mutations and the AFLNet fuzzing loop are made syntax-aware. These exist within the *afl-fuzz* program itself. We now need to understand how AFLNet-Packmute gets the ability to identify mutable

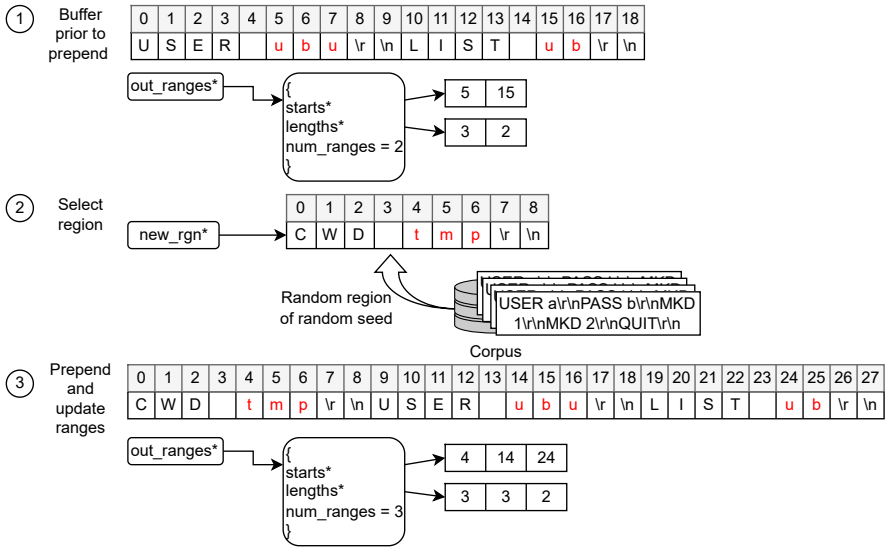

Figure 3.3: Prepend-region mutation illustration (exploit)

ranges from arbitrary buffers and how that information is efficiently stored in the region metadata.

3.3.3 Structural Design

To perform syntax-aware fuzzing, AFLNet-Packmute needs three things:

1. The ability to verify the syntax validity of a sequence of messages;
2. The ability to find the mutable byte ranges within a sequence of messages; and
3. The ability to pad zero-length ranges when mutating to preserve syntax validity.

Figure 3.4 shows the overall project design. There are three components; PcapPlusPlus, Libpackmute and AFLNet-Packmute (noteably the *afl-fuzz* program). This section describes how these are structurally designed, with some reference to their interaction, while the next section 3.3.4 depicts the interaction between the project's components using sequence diagrams.

PcapPlusPlus is an existing network packet library which is known for its excellent performance and active development [35]. It handles the management of raw data comprising protocol layers, can parse out application layer data fields from that raw data, and exposes some protocol specification data, such as an enumeration of FTP commands. This library is selected due to

its ability to parse application layer protocols and its language compatibility with AFL, being written in C++. The library does not however contain all the functionality needed by AFLNet-Packmute. For example, it does not know which FTP Commands must or may have arguments, it does not expose a C-compatible interface, and it does not operate on message sequences.

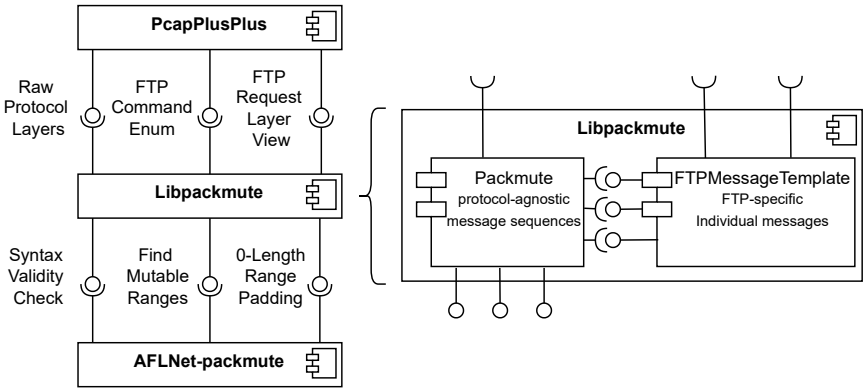

Figure 3.4: AFLNet-Packmute component diagram (left) and Libpackmute modules (right)

Libpackmute is a C++ library developed for this research which implements that missing functionality. Via the *Packmute* module, it exposes a C-compatible interface that is protocol-agnostic and operates on sequences of messages. AFLNet-Packmute can thus provide it entire message sequences and ask Libpackmute to determine whether the messages in the sequence have valid syntax, where the mutable ranges are found, and how to pad zero-length ranges to retain syntax-validity. The message sequences are passed from AFLNet-Packmute to Libpackmute using AFLNet's replay format, which encodes the length of individual messages within a single continous byte array. Libpackmute then can decode these to individual messages.

Packmute conducts protocol-agnostic portions of the process and offloads protocol-specific functions to a separate module, which in the case of FTP is named *FtpMessageTemplate*. This module uses and enriches the protocol knowledge offered by PcapPlusPlus to offer the same three interfaces offered by *Packmute*, but for individual application layer protocol messages. Adding support for a new protocol would simply require a new protocol-specific module to be written, the new protocol added to *Packmute*'s enumeration of supported protocols and the dispatch functions within *Packmute* to have one case added. Let's look in greater detail how each part of the project is structured,

starting with PcapPlusPlus.

Figure 3.5 shows the hierarchy of PcapPlusPlus classes relevant to parsing FTP requests. At the top you have a *Layer*, which is simply some raw data with a length. The destructor handles freeing the data. Its child class, *SingleCommandTextProtocol*, is used by protocols which comprise a single command with some text arguments (ex. FTP, SMTP). This class provides a static method, *isDataValid()*, which verifies if the layer's data buffer is terminated by the carriage-return and line-feed characters. That method is later used by Libpackmute to help verify syntax validity. The *SingleCommandTextProtocol* class contains methods which parse out the bytes associated with the command and argument fields to be used by child classes. Finally there is the *FtpRequestLayer* class which provides public methods to access the command and option fields within an FTP request buffer. The command can either be obtained as a string or as an enumeration value, which it also defines. All public methods shown are used by Libpackmute's *FTPMessageTemplate* module, and the *Layer* class methods are also used by the *Packmute* module. When instantiating these layers from an existing data buffer, the FTP class simply calls the parent's constructor using its initialization list. Now onto Libpackmute.

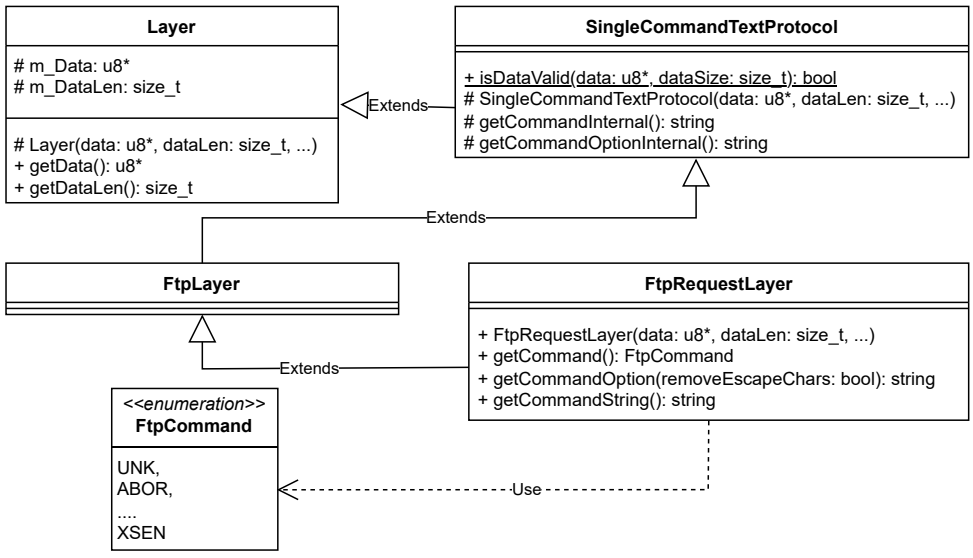

Figure 3.5: PcapPlusPlus select classes.

Libpackmute comprises a protocol-specific class (*FtpMessageTemplate*), and a source file offering the C-compatible interface and protocol-agnostic message sequence parsing (*Packmute*). Figure 3.6 shows the class diagram. The *FtpMessageTemplate* class comprises private members needed to define a template; a command, argument and whether that argument is optional. The command is from the PcapPlusPlus defined enumeration and the argument is defined using a regex pattern. Why then is a boolean *optional* member needed? There is no way to extract the regex string from an instantiated *std::regex* object. *Boost::basic_regex* could have been used, but would have introduced an additional dependency. The class initializes a static private vector of *FtpMessageTemplates* defining each FTP command, and it's corresponding argument. Again, this is due to PcapPlusPlus not having knowledge of which commands may or must have arguments. This portion of *FtpMessageTemplate* could potentially be upstreamed to PcapPlusPlus. Next we look at the methods in *FtpMessageTemplate*.

The private static *match_template()* method uses the *FtpMessageTemplate* vector to determine whether a given *FtpRequestLayer* object first has a valid command then has a valid argument for that command. The public static method *is_syntax_valid()* first uses *SingleCommandTextProtocol::isDataValid()* to verify the message is properly terminated and them *match_template()* to verify if the message matches a FTP template.

The *find_mutable()* method takes a valid *FtpRequestLayer* and two lists by reference. It will then append to the lists the start of the mutable range in this FTP request and the length if any. The template obtained from a call to *match_template()* indicates if there is optionally an argument, in which case it creates a zero-length mutable range. PcapPlusPlus' *FtpRequestLayer* methods that provide string representations of each the command and option fields are used to obtain the lengths of each of those fields. Note this method is very simple in the case of FTP, where each message has at most one mutable range. More complex protocols would need to append a mutable range start and length for each variable field in the protocol message.

In the case of FTP, if AFLNet proceeds to insert bytes in a zero-length mutable range, we need padding to be added to keep it syntactically valid. The padding required is offered by the public static method *pre_padding()*. Though in the case of FTP it is a single space character, the method is written to return a pointer to a character array so that AFLNet-Packmute could support any size of pre-padding needed.

As the name implies, *FtpMessageTemplate* is FTP-specific and it only handles individual messages. *Packmute* is a source file and header which bridges *afl-fuzz* and the protocol-specific knowledge of *FtpMessageTemplate*.

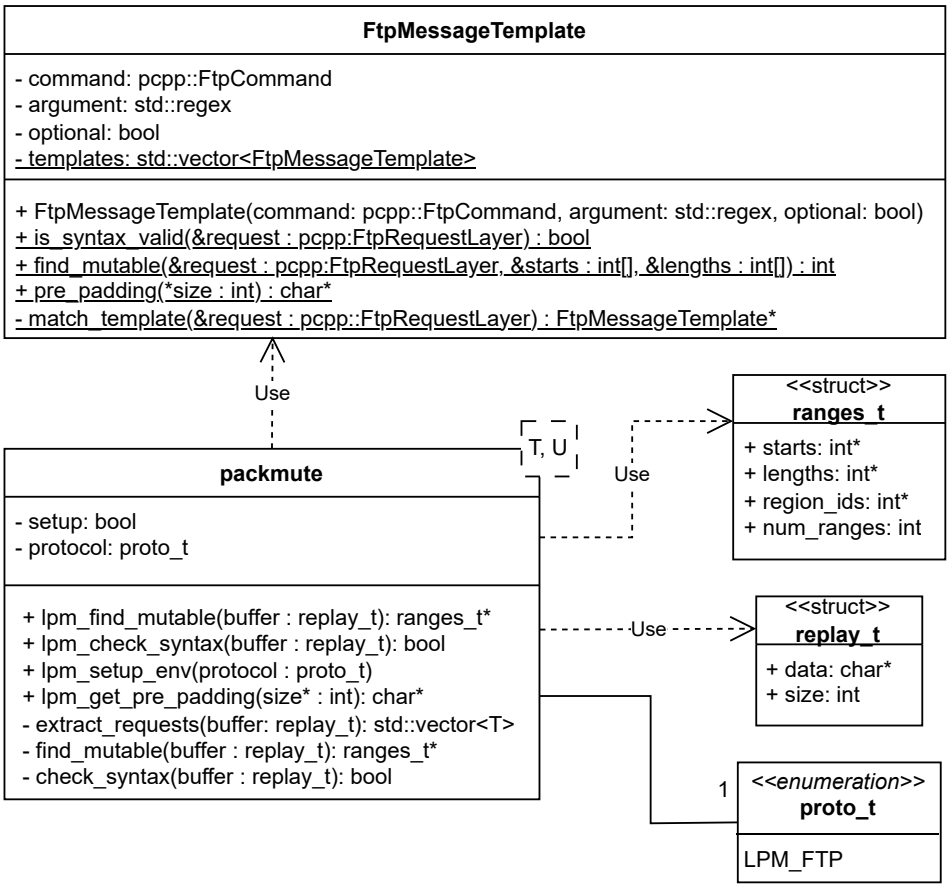

Figure 3.6: Libpackmute class diagram

It comprises public setup and dispatch functions alongside private template functions which are instantiated with the proper types and then called by the public dispatch functions.

First, *afl-fuzz* must call the public function *lpm_setup_env()* and provide a value found in the *proto_t* enumeration. The private *setup* member is then set. This protocol setting is used by the public dispatch functions *lpm_find_mutable()* and *lpm_check_syntax()* to instantiate their private function equivalents using the type-specific class (ex. *FtpMessageTemplate* and

FtpRequestLayer). These templated functions are where protocol-agnostic logic occurs as well as the conversion from a message sequence to individual messages.

The templated function *check_syntax()* first splits the replay format buffer passed as an argument into a vector of PcapPlusPlus layer objects through the *extract_requests()* function and then passes those individual layer objects to the *is_syntax_valid()* method of the corresponding protocol-template class. This function returns true if all messages in the sequence have valid syntax, otherwise it returns false.

The *find_mutable()* templated function similarly starts by obtaining a vector of individual layer objects, iterating through each one and verifying that the syntax is valid. In the case the syntax is valid, it calls the protocol-template class' *find_mutable()* method to find the mutable range(s) in the message. The start position, length and associated region of the mutables ranges are stored in a *ranges_t* struct. *afl-fuzz* needs to know to which region in the message sequence each mutable range belongs and where the mutable ranges are relative to the start of the overall message sequence. These are tracked by *Packmute*'s *find_mutable()* function and assisted by PcapPlusPlus' layer features to obtain the raw data length of the entire message. Now we have a protocol-agnostic, message-sequence based interface to support syntax-aware fuzzing. Next comes the changes and additions made to AFLNet to create AFLNet-Packmute.

Shown in Figure 3.7 are **additions** or *changes* made to *afl-fuzz* functions and variables. A call to *lpm_setup_env()* is added in *main()* passing it the protocol indicated when *afl-fuzz* is executed. The **app_protocol** variable is added for that purpose. The existing AFLNet region struct is expanded to include two additional arrays (**mut_starts** and **mut_lengths**) and an array length variable (**mut_counts**). These store the start positions and lengths of mutable ranges within the region. The mutable range start position is relative to the start of the entire seed the region finds itself in. The change to *destroy_queue()* is to delete these new arrays.

The decision to store mutable range information alongside each queue entry in its region information differs from ChatAFL's approach. ChatAFL reparses the mutable ranges in a given buffer each time the fuzzing loop is entered. A fuzzing loop is always seeded with a queue entry, so AFLNet-Packmute annotates the region structs with the mutable range information when an entry is added to the queue. When a queue entry is selected to seed a round of fuzzing, its mutable ranges can simply be extracted from its existing region structures, reducing the overhead of matching the buffer to a message template and finding the mutable ranges. There are three occasions where

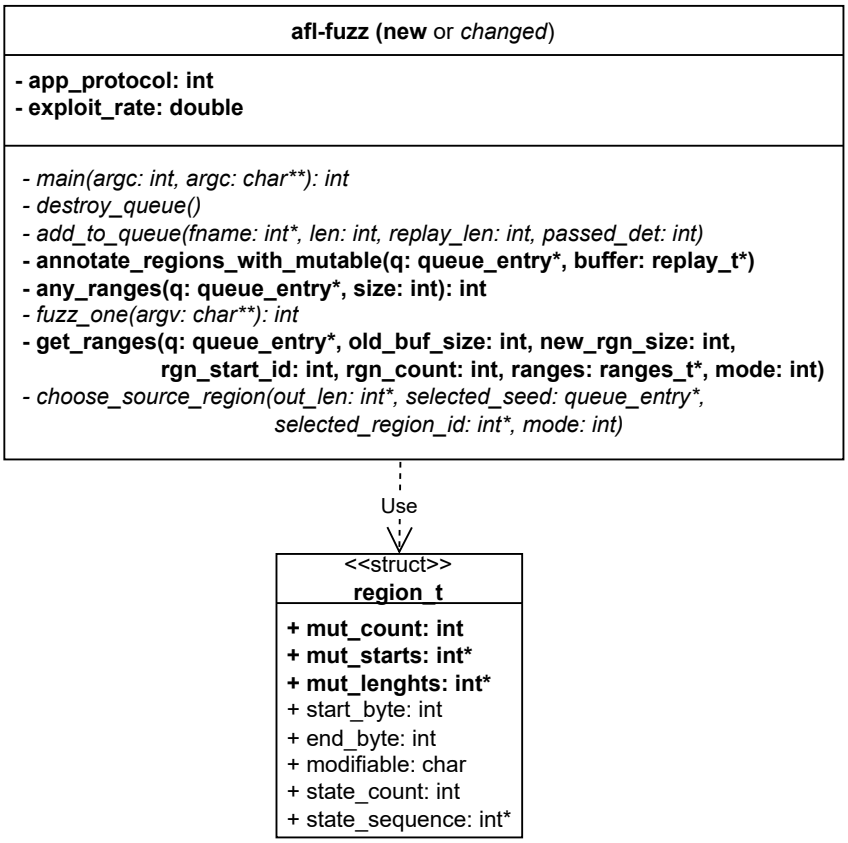

Figure 3.7: AFLNet-Packmute class diagram

entries are added to the queue; during fuzzing, parallel fuzzer sync, and initial testcases. In the case of fuzzer sync or during fuzzing, the entry has already been recorded to disk in replay-format, so the *add_to_queue()* function reads it from disk and passes it to **annotate_regions_with_mutable()**. For initial testcases, it must first construct the replay-format buffer from the queue entry then call the aforementioned function. **annotate_regions_with_mutable()** simply calls *lpm_find_mutable()* to get the mutable ranges and writes them to the appropriate region's **mut_** struct elements.

any_ranges() is a helper function that looks through a queue entry and counts the number of mutable ranges of a certain size. It is used during the

splice stage to ensure a splice target with at least one mutable range of size two or greater is selected. It is also used during region mutations when an entry with at least one mutable range of zero-length or greater is needed.

fuzz_one() is the function where the fuzzing loop algorithm takes place (Algorithm 3). How ranges are added was discussed in Section 3.3.1 and how syntax-awareness was added to mutations discussed in Section 3.3.2. Here we discuss how range management occurs through the **get_ranges()** function and some more technical details on how the syntax-aware mutations occur. This is abstracted as a call to *mutate()* in Algorithm 3 though in practice it involves loops and case statements inside *fuzz_one()*.

On line 7, Algorithm 3 calls a function named **get_ranges()** and passes it the buffer that will be mutated in this fuzzing loop. This function is responsible to update the range struct in the case where entire new regions are being added or removed. Provided to **get_ranges()** is a pointer to the queue entry from which regions are being added, the existing size of the byte buffer, the size in bytes of the new region being added, the ID of the first region in the queue entry being added and the number of regions added. Also passed to **get_ranges()** is a pointer to the range struct being updated and a mode argument. The mode argument determines what type of update should be performed on the range struct:

1. Mode 0: replace all existing ranges with ranges from new region(s);
2. Mode 1: prepend ranges from new region(s) to existing ranges; and
3. Mode 2: append ranges from new region(s) to existing ranges.

Mode 0 is used when a new fuzzing round begins to get the *orig_ranges* and when the replace buffer region mutation occurs on *out_buf* while exploiting. Mode 1 and 2 are used for the region mutations where a region is added to the start or end of *out_buf*.

On line 17, Algorithm 3 calls a function named *mutate()* which applies a single mutation to the *out_buf*. This fuction does three things. First, the function randomly selects a mutable range to mutate. When the decision is to explore, or the original buffer had no mutable ranges, there is only one range to pick from; the entire buffer. Second, the function generates a random number to pick which mutation to apply. Third, it attempts to apply that mutation. Most mutations have a length requirement. For example, adding to a dword requires that the mutable range be at least 4 bytes long. If the mutable range does not meet that requirement, the loop continues to the next stacked mutation without making any change to the buffer. If the requirement is met, the mutation is constrained to occurring within the selected mutable range. In the case where the length of the buffer is changed, the length of the effected

mutable range is updated, as well as the start positions of every mutable range after the currently effected range. To make these changes convenient, the mutable ranges are sorted based on their order in the buffer; first to last. The inserting of new bytes is the only non-region mutation that applies to zero-length ranges. In that case, a call to *lpm_pre_padding()* is needed and the padding must be taken into consideration when adjusting ranges.

For region mutations, AFLNet relies on the *choose_source_region()* function to pick a random region from a random seed. AFLNet-Packmute modifies the function to have two modes; mode 0 to pick any region or mode 1 to pick a region with at least one mutable range. Mode 1 is used when replacing the buffer and exploiting, while mode 0 is used for prepend/append ranges or when we are replacing the entire buffer and exploring. AFLNet-Packmute ensures new regions that are added to the buffer have their mutable ranges added to the range struct as well, unless we are exploring in which case the single mutable range is updated. When the buffer is being entirely replaced with a new region, and we are currently exploiting, AFLNet-Packmute will ensure a region with at least one mutable range is selected so exploitation can continue.

To facilitate testing, the **exploit_rate** variable is set through a command line switch *-z*. This is a new switch to AFLNet-Packmute which sets the exploitation rate. That is, what proportion of fuzzing rounds should apply syntax awareness.

3.3.4 Behavioural Design

This section provides sequence diagrams for each Libpackmute public function with a focus on showing the interaction between *afl-fuzz* with Libpackmute modules, and those modules' interaction with PcapPlusPlus. As PcapPlusPlus is an existing library not developed by this research, and for readability, it is not broken down into its individual classes.

Figure 3.8 depicts the sequence diagram for *lpm_get_pre_padding()*. First, *afl-fuzz* must tell the *Packmute* module that the FTP protocol is being fuzzed. Once *lpm_get_pre_padding()* is called, *Packmute* simply obtains a pointer to a buffer containing the padding from *FtpMessageTemplate* and returns it to *afl-fuzz*. A pointer to an integer variable is passed along to record the size of the padding.

Figure 3.9 depicts the sequence diagram for *lpm_check_syntax()*. The sequence of application-layer requests are sent to *Packmute* in replay-format. *Packmute* then instantiates a function using the FTP-specific PcapPlusPlus layer class and its own FTP template class. This function extracts

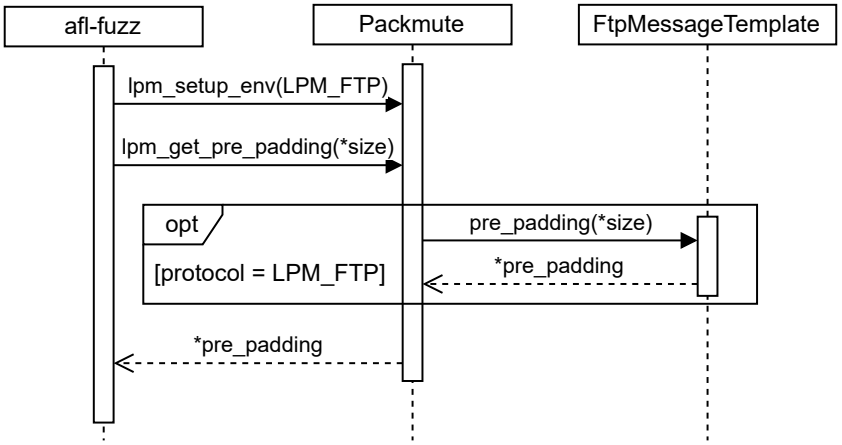

Figure 3.8: Pre-padding sequence diagram

the individual requests from the replay-format buffer and builds a vector of layer objects. The syntax validity of each request is then verified by *FtpMessageTemplate*. To do this, it verifies the buffer is properly terminated using PcapPlusPlus' *SingleCommandTextProtocol::isDataValid()* static function and then verifies that it matches a FTP message template. Matching a template involves an exact match of the command, using PcapPlusPlus' FTP command enumeration, and a regex search of the option field using the message template's regex pattern. If any one request does not have valid syntax, the function returns FALSE immediately, else it verifies each request before returning TRUE. This message sequence-checking function is not used in the final AFLNet-Packmute implementation, rather *is_syntax_valid()* is used directly in *lpm_find_mutable()* to determine syntax validity on a per-message level. It was used, however, during the development of AFLNet-Packmute for verification purposes as will be discussed in Section 4.4. This sequence diagram is also useful to show how the *extract_requests()* and *match_template()* functions work, details that are ommitted in the next sequence diagram for readability.

Figure 3.10 depicts the sequence diagram for *lpm_find_mutable()*. The sequence of application-layer requests are sent to *Packmute* in replay-format. *Packmute* then instantiates a function using the FTP-specific PcapPlusPlus layer class and its own FTP template class. That function extracts the individual requests, iterates through each one, and if it has valid syntax it will

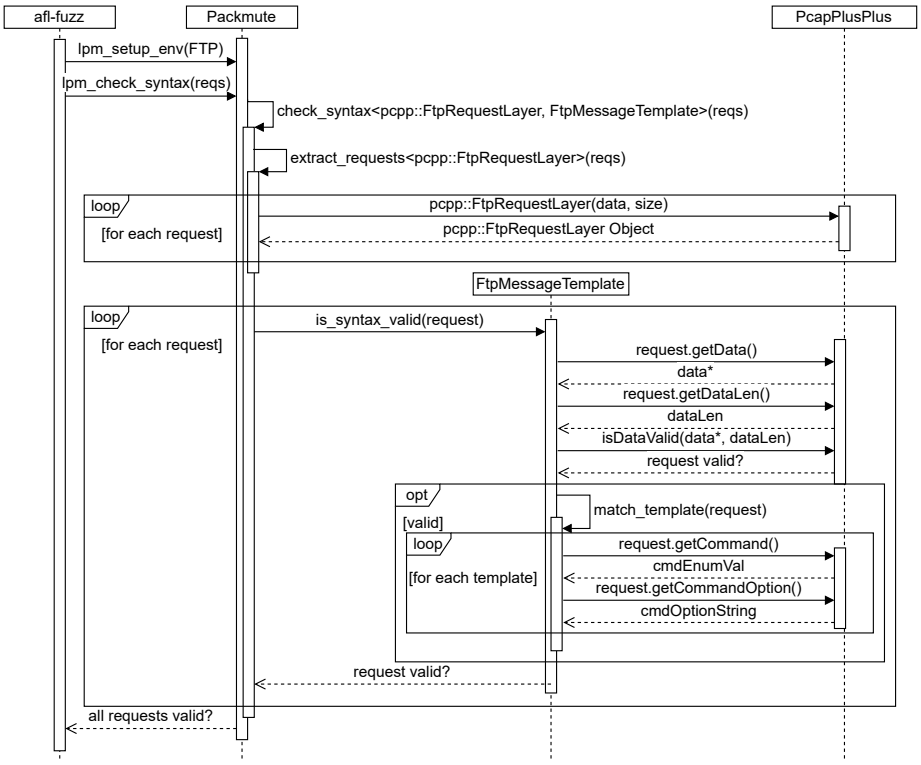

Figure 3.9: Check syntax sequence diagram

find the mutable ranges in it by calling *FtpMessageTemplate*'s *find_mutable()* function. That function accepts references to a start and length vector which it will append mutable ranges to, and it returns the number of new ranges added. To find the mutable ranges, in the case of FTP, it obtains the protocol field sizes from PcapPlusPlus and uses those sizes to append a mutable range according to the size and position of the option field. If there is no option, *FtpMessageTemplate* will verify if an option could go there, in which case it adds a zero-length mutable range. This applies to FTP commands that may or may not have an option, which is captured in the vector of valid message templates. AFLNet-Packmute could, for examples, insert bytes there to give it an option.

Packmute's *find_mutable()* function handles the request sequence-level logic. It will populate which region in the request sequence the mutable range be-

longs to and shift range start positions according to where they exist in the overall message sequence. Recall mutable range positions are relative to the entire sequence they exist in, and not the individual region. Lastly, it takes the three parts; start, length, and region IDs, to build the range structs that are returned to *afl-fuzz*.

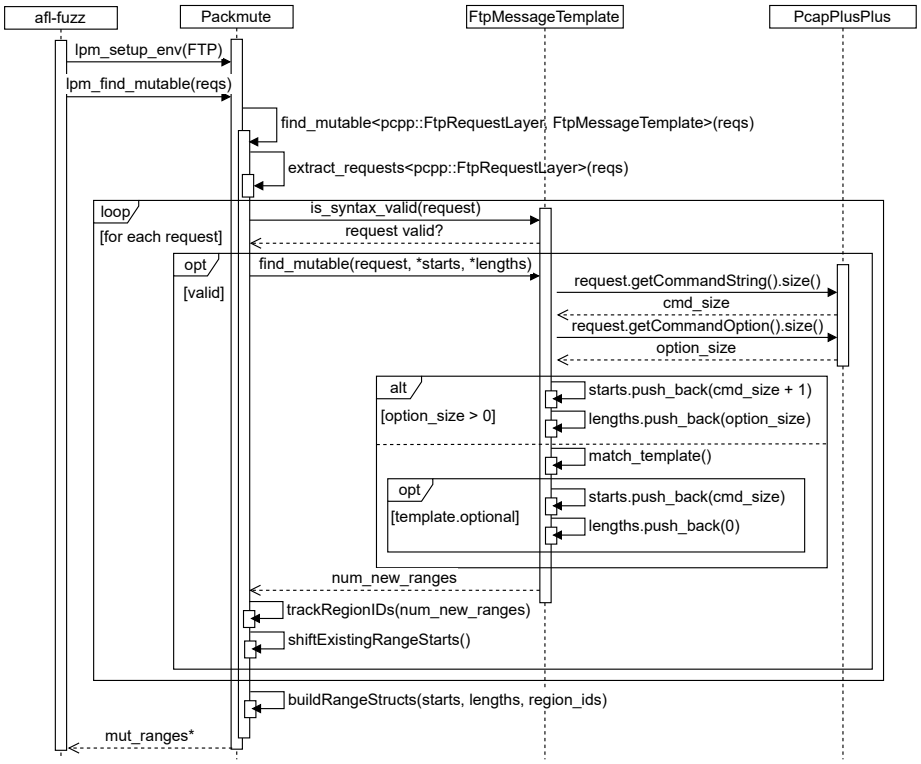

Figure 3.10: Find mutable range sequence diagram

With this design in mind, we will next discuss how the output of AFLNet-Packmute is verified.

3.4 Verification

The verification phase is designed to confirm that AFLNet-Packmute behaves as intended and that analysis scripts are correctly implemented before moving onto the validation phase. We expect AFLNet-Packmute to cre-

ate syntactically valid fuzz when exploiting, while retaining AFLNet-like behaviour when exploring.

3.4.1 Analysis Script Verification

We assume ProFuzzBench's coverage time series output is accurate. This is our starting point for analysis. From the coverage time series, we produce two independent outputs; plots of median coverage over time overlaid with min/max coverage, and overall performance statistics. These independent outputs should agree on the final median, min, and max coverage as well as the coverage speed-up factors. This helps verify that the analysis scripts are correctly implemented. To further verify the statistics, values for one fuzzer configuration and target combination is verified manually.

3.4.2 AFLNet-Packmute Verification

The verification of AFLNet-Packmute involves:

1. **Test-driven development of Libpackmute**: for each public function providing features to AFLNet-Packmute, tests are written based on expected behaviour and then the function is developed to satisfy the test. This is done using CMake's testing program, CTest;
2. **AFLNet-Packmute mutation output verification using Libpackmute**: code is added to AFLNet-Packmute to call *lpm_check_syntax()* on every fuzz generated and break if the syntax is invalid. The mutation applied and the content of *out_buf* on each iteration is logged so that root cause can be found. There is a form of self-fuzzing that occurs at this verification step as well. Writing code that directly works with fuzz being generated inherently triggers edge-cases and uncovers bugs; and
3. **Final AFLNet-Packmute verification using a real FTP server**: LightFTP is fuzzed for exactly ten minutes while network traffic is being captured. Analysis of server response codes then allows us to verify the quantity of commands that are rejected due to bad command syntax. This is conducted using AFLNet as a baseline, AFLNet-Packmute with an exploit rate of 100% and AFLNet-Packmute with an exploit rate of 0%. Results vary over multiple runs and so it is repeated three times and the average taken.

3.5 Validation

This research phase is designed to fulfill the aim of the research. The effectiveness of state feedback features in a syntax-aware AFLNet derivative, AFLNet-Packmute, is measured. Then the change in effectiveness between a syntax-aware (AFLNet-Packmute) and syntax-unaware (AFLNet) mutation process is measured. The change in effectiveness achieves the aim of the research. The state feedback features of interest are:

1. **Basic state feedback.** Seeds are parsed into a prefix, candidate sequence, and suffix. Mutations are only applied to the candidate sequence. States are derived from target response codes and selected for fuzzing. Seeds are chosen from among those that traverse the selected state;
2. **Favoured state selection.** States are selected for fuzzing based on heuristics which score a state's likelihood to find new coverage as described in Section 2.2.1; and
3. **Favoured seed selection.** Seeds with a higher likelihood for the discovery of new coverage are selected as described in Section 2.2.1.

Fuzzing experiments are conducted using each fuzzer configuration previously listed in the ablation study design (Table 3.1), though instead of AFLNet, AFLNet-Packmute is used. The experimental parameters in this phase match the ablation study described in Section 3.2. Ten fuzzing sessions of each configuration are run against each FTP target in ProFuzzBench. Experiments last 24 hours. Each ten fuzzer experiments are conducted in a separate VM to mitigate kernel bottlenecks. Effectiveness is measured by the percent change in performance when each of the above listed state feedback features is enabled relative to a parent configuration lacking that feature. Performance is based on branch coverage achieved during fuzzing as measured by *gcov*. Like the ablation study, the set of configurations used allows for multiple ways to measure each feature, mitigating risk posed by statistical significance. The impact of favoured seed and state selection can be measured four ways:

1. Measuring the feature impact when using the STFL-FVR configuration enabling each favoured state or seed selection relative to STFL-RND;
2. Same configurations as 1. but with region mutations enabled;
3. Measuring the feature impact when using a STFL-FULL configuration relative to the STFL-FVR configuration enabling each favoured state or seed selection; and
4. Same configurations as 3. but with region mutations enabled.

Basic state feedback's feature impact can be measured two ways;

1. Measuring the feature impact when using the STFL-FVR-SD relative to a BASELINE configuration. The impact the different seed selection approaches between these configuration has been measured by previous work [15]; and
2. Same configurations as 1. but with region mutations enabled.

For conciseness the notation "BASELINE[-RGNS] v. STFL-FVR-SD[-RNGS]" is used in table titles. This title describes the two measurements for basic state feedback's feature impact; one between BASELINE and STFL-FVR-SD and the other between BASELINE-RGNS and STFL-FVR-SD-RGNS.

The change in effectiveness is measured between results obtained using AFLNet in the ablation study and the results obtained using AFLNet-Packmute in the validation phase. Referring back to Figure 3.1, we measure how arrows 1, 2 and 3 change after adding syntax-awareness. Using this change in effectiveness, a determination on the impact to adding syntax-awareness can be made and the aim of this research achieved.

To draw additional insights beyond the exact aim of the research or help explain findings, this validation also looks at secondary measurements to draw additional insights; speed-up (how fast fuzzer B reached the coverage of fuzzer A), average executions per second, and fuzzer min/max run code coverage. Lastly, this research provides insights into the code coverage value of adding protocol-syntax awareness into the mutation process of greybox coverage-guided protocol fuzzing which is discussed in Appendix A.

4 Results

This chapter presents the results of the research and achieves the aim in six parts:

- Experimental Design - The infrastructure on which experiments are conducted, and the data pipeline used to analyze results;
- Ablation Study - The performance results and state feedback feature impacts for the AFLNet fuzzer against all ProFuzzBench targets;
- AFLNet-Packmute Development - The development and configuration of the AFLNet-Packmute fuzzer;
- Verification - The outcome of each verification step outline in the design;
- Validation - The performance results and state feedback feature impact for the AFLNet-Packmute fuzzer against ProFuzzBench FTP targets. Measurement of state feedback feature impact change as a result of adding syntax-awareness; and
- Discussion - To sumamrize, a discussion highlights the main research findings throughout each research activity.

4.1 Experimental Design

Experiments were conducted using ProFuzzBench and its included target versions. Containers were launched in Ubuntu 22.04LTS guest virtual machines (VMs) using the VMWare Workstation hypervisor running on a Windows Host OS. Engineering workstations were used to run the experiments. Each workstation runs two VMs, each VM capable of running one 10-container experiment. 12 cores were assigned to each VM, such that each container gets one core equivalent and allows two additional cores for the guest OS. Though AFL is designed to consume an entire core's processing, AFLNet is IO bound, meaning much less than a full core is used. In this research the most CPU intensive target (Forked-DAAPd) saw all 12 cores above 60% usage, though not exceeding 90%, while most targets saw much lower usage. Tests using LightFTP and OpenSSH targets found half a GB of RAM per fuzzer was

sufficient to meet demand, with most usage being caching, thus 8GB of RAM was provided to each VM.

To speed up experimentation, two sets of computers were used while ensuring that a given target's experiments were only conducted on one set. The first set comprised four Dell workstations accessible over WiFi. The second set comprised three HP Z8 workstations on RMC's network. In total, this meant 14, 10-container experiments could run concurrently. Targets were fuzzed for 24 hours followed by coverage analysis. The coverage analysis step of ProFuzzBench is CPU intensive and takes from a few minutes to several hours depending on the target. This fact, paired with power outages, surprise reboots of RMC networked machines, and target stability issues led to experimentation taking several weeks. Other research-related work was done concurrently. Figure 4.1 shows the experimental setup, the software layers involved, as well as the network configuration of the WiFi accessible fuzzing computers. In the case of failed runs, additional runs were performed until 10 fuzzing experiments lasting the full 24 hours were successful. Prior to experimentation all commands used to run each target + configuration combo were created and reviewed for accuracy. A single VM was prepared with all the target containers built and dependencies installed prior to being cloned across all the experimentation workstations.

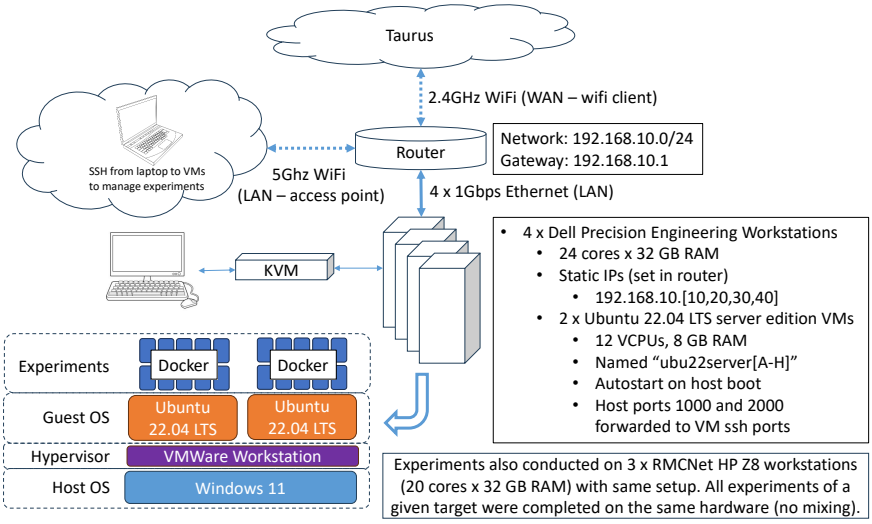

Figure 4.1: Experimental setup

The latest version of ProFuzzBench (commit 8573ec8) and AFLNet (commit 6d86ca0) at the time of this research was used. ProFuzzBench by

default compiles an old version of AFLNet, so all Dockerfiles were updated to pull the specific newer AFLNet from AFLNet's code repository and install new dependencies.

ProFuzzBench outputs an archive per container, containing the fuzzer output directory and the coverage analysis results. ProFuzzBench provides a shell script that extracts the coverage time series from the archives and another that plots the average coverage at each time step. The plotting script outlined in Section 2.2.2 was modified to plot the median coverage at each timestep overlaid with min/max coverage. Shell scripts were written to extract data from the fuzzer output directory and create aggregated logs. Statistics were calculated using R since it has a standard library for calculating the Vargha-Delaney effect size. The plotting script already builds a dataframe of the coverage time series, so it is used to enrich the statistics with a coverage speed-up factor; that is, how much faster did fuzzer B reach the max coverage of fuzzer A, in the case that it is faster. All this data is grouped to form two results CSV files that could be analyzed using pandas [36]. Table 4.1 lists the result data fields and information by file. The shared fields are found in both files and together act as the primary key for the data.

4.2 Ablation Study

The ablation study in total comprised 10 instances each of 10 AFLNet configurations fuzzing 13 targets over 24 hours, totalling 1300 days worth of fuzzing. Theoretically, the seven workstations could complete the fuzzing portion in just over nine days. In practice, the time to start experiments, confirm they are running correctly, collect results, verify they ran the full 24-hours, overcome target-specific challenges, host computer issues and power outages meant it took around five weeks.

This section is broken down in three subsections:

- Coverage results - The performance results of the fuzzing experiments including coverage, executions speeds and crash statistics;
- State feedback feature impact - Measuring the branch coverage contribution of enabling basic state feedback, favoured seed selection and favoured state selection. For completeness results of a fully featured state feedback configuration is included;
- Discussion - comments, thoughts, notes around the conduct of the ablation study not related to the numerical results themselves.

Table 4.1: Results data fields

Name	Type	Description
		Shared fields
Fuzzer	String	Name of the fuzzer used in experiment
Target	String	Name of target fuzzed
Config	Enum	Configuration name; ref Table 3.1
		Performance results
Crash_Median	Float	Median of crashing testcases found by the fuzzer
Execs_second	Float	Average target executions per second
BCov_Min	Integer	Least final branch coverage among set of runs
BCov_Max	Integer	Most final branch coverage among set of runs
BCov_Median	Float	Median final branch coverage among set of runs
		Comparative results
RelativeTo	Enum	Comparison configuration for statistical significance calculation; ref Table 3.1
BCov_MannWhitneyP	Float	p-value between Config and RelativeTo
BCov_VDA.12.val	Float	Effect size between Config and RelativeTo
BCov_med_delta_abs	Float	Difference in median branch coverage achieved between Config and RelativeTo as a count
BCov_med_delta_per	Float	Difference in median branch coverage achieved between Config and RelativeTo as a percentage
Speed Up	Float	Ratio of the total fuzzing time over seconds taken by Config to reach the final coverage of RelativeTo

4.2.1 Coverage Results

Table 4.2 lists the fuzzing performance results of the ablation study. Results are grouped by target, listing the fuzzing metrics for each configuration. Metrics include the minimum, maximum and median branch coverage achieved among the 10 runs. As there are 10 runs, the median is the average of the two middle values in a sorted list of final branch coverages. The percent change in median final branch coverage between two configurations is used to measure the impact of fuzzer features. The same process is used to measure feature impact in AFLNet-Packmute, our syntax-aware AFLNet variant. The difference in these measured impacts allowed us to reach our aim.

To the right of branch coverage results, the target executions per second are listed. This is the number of times the fuzzer spawns the target process, sends a fuzz, observes the response and kills the process each second on average throughout the fuzzing session. This metric helps compare results across research efforts. Lastly, the median number of unique crashing testcases found

by the 10 runs is listed. Unique crashes from a coverage perspective can be related to the same bug. No bug deduplication is performed.

The median final branch coverage bolded for each target indicates the most performant configuration for that target. Observe that the best performing configuration is seldom STFL-FULL-RGNS, the configuration with all of AFLNet's features enabled. This points to two things: 1) potential future work involving the automated discovery of ideal fuzzer configuration for a given target. In preliminary experiments, we found that coverage results are significantly impacted by changes to timeouts, such as how long the server is allowed to process each message. 2) using a single fuzzer configuration to assess the impact of changes to AFLNet can lead to incomplete observations. Appendix A discusses this further in regards to the impact of adding syntax-awareness.

Table 4.2: Ablation study performance results

Config	Branch Coverage			Execs/s	Crash
	Min	Max	Med		Med
	Live555 - RTSP				
BASELINE	2797	2826	**2814.5**	10.00	48.0
BASELINE-RGNS	2787	2837	2810.5	9.59	69.5
STFL-RND	2742	2794	2762.0	9.19	16.5
STFL-RND-RGNS	2708	2774	2739.0	8.29	50.0
STFL-FVR-SD	2734	2809	2766.0	9.23	24.5
STFL-FVR-SD-RGNS	2708	2786	2764.5	8.53	27.5
STFL-FVR-ST	2754	2811	2773.0	9.34	17.5
STFL-FVR-ST-RGNS	2734	2763	2751.0	9.01	31.0
STFL-FULL	2750	2802	2778.5	9.66	25.5
STFL-FULL-RGNS	2707	2788	2770.5	9.38	34.0
	Exim - SMTP				
BASELINE	2783	2962	2859.0	2.91	0
BASELINE-RGNS	2841	2946	2886.5	3.06	0
STFL-RND	2758	2929	2845.5	2.90	0
STFL-RND-RGNS	2824	2923	2896.0	3.08	0
STFL-FVR-SD	2755	2910	2844.0	2.95	0
STFL-FVR-SD-RGNS	2864	2957	2914.5	3.10	0
STFL-FVR-ST	2760	2952	2873.5	2.83	0
STFL-FVR-ST-RGNS	2817	2934	2861.5	3.00	0
STFL-FULL	2844	2929	2873.5	2.61	0
STFL-FULL-RGNS	2878	2966	**2931.0**	2.67	0

Continuation of Table 4.2

Config	Branch Coverage			Execs/s	Crash Med
	Min	Max	Med		
	ProFTPd - FTP				
BASELINE	5208	5544	**5347.5**	2.93	0
BASELINE-RGNS	5008	5310	5226.5	2.15	0
STFL-RND	4381	5356	5036.0	2.08	0
STFL-RND-RGNS	4624	5252	5042.5	1.96	0
STFL-FVR-SD	4862	5325	5096.0	2.18	0
STFL-FVR-SD-RGNS	4858	5218	4950.5	2.13	0
STFL-FVR-ST	4459	5188	4944.5	2.08	0
STFL-FVR-ST-RGNS	4465	5270	4939.5	1.96	0
STFL-FULL	4922	5221	4976.0	2.50	0
STFL-FULL-RGNS	4693	5381	4948.5	2.26	0
	LightFTP - FTP				
BASELINE	345	376	**360.0**	5.59	0
BASELINE-RGNS	345	362	347.0	5.74	0
STFL-RND	342	363	346.5	5.35	0
STFL-RND-RGNS	331	364	346.0	5.29	0
STFL-FVR-SD	342	363	347.5	5.56	0
STFL-FVR-SD-RGNS	341	365	346.0	5.57	0
STFL-FVR-ST	332	358	343.5	5.58	0
STFL-FVR-ST-RGNS	324	362	344.0	5.37	0
STFL-FULL	340	359	348.5	5.88	0
STFL-FULL-RGNS	341	365	344.5	5.64	0
	PureFTPd - FTP				
BASELINE	1191	1288	**1244.5**	5.00	0
BASELINE-RGNS	1171	1277	1214.0	4.21	0
STFL-RND	1010	1187	1132.5	4.55	0
STFL-RND-RGNS	779	1144	989.5	3.48	0
STFL-FVR-SD	891	1230	1150.0	4.37	0
STFL-FVR-SD-RGNS	886	1109	1046.0	3.85	0
STFL-FVR-ST	1062	1174	1133.0	4.40	0
STFL-FVR-ST-RGNS	863	1127	1038.5	4.22	0
STFL-FULL	882	1275	1139.5	4.40	0
STFL-FULL-RGNS	916	1108	1054.0	4.09	0

Continuation of Table 4.2

Config	Branch Coverage			Execs/s	Crash Med
	Min	Max	Med		
	BFTPd - FTP				
BASELINE	463	496	484.5	3.06	31.0
BASELINE-RGNS	467	489	481.0	3.34	28.0
STFL-RND	455	491	484.5	3.61	19.0
STFL-RND-RGNS	471	496	476.5	3.11	19.0
STFL-FVR-SD	476	495	485.0	3.71	22.0
STFL-FVR-SD-RGNS	480	489	482.0	3.15	17.5
STFL-FVR-ST	468	490	483.5	3.51	20.0
STFL-FVR-ST-RGNS	455	486	476.5	2.75	24.0
STFL-FULL	481	498	**485.5**	3.98	19.5
STFL-FULL-RGNS	477	489	485.0	3.13	21.5
	OpenSSH - SSH				
BASELINE	3254	3384	3328.5	8.86	0
BASELINE-RGNS	3238	3360	3292.5	6.01	0
STFL-RND	3294	3369	3327.0	20.71	0
STFL-RND-RGNS	3311	3355	3329.0	17.00	0
STFL-FVR-SD	3315	3372	3339.0	18.96	0
STFL-FVR-SD-RGNS	3315	3354	3335.0	16.38	0
STFL-FVR-ST	3321	3388	**3360.5**	20.92	0
STFL-FVR-ST-RGNS	3323	3359	3340.5	17.82	0
STFL-FULL	3326	3360	3345.5	18.70	0
STFL-FULL-RGNS	3310	3378	3351.5	17.18	0
	OpenSSL - SSL				
BASELINE	9987	10190	10051.0	4.03	0
BASELINE-RGNS	10014	10129	10049.0	4.06	0
STFL-RND	10011	10322	10057.0	4.35	0
STFL-RND-RGNS	9699	10104	9949.5	4.55	0
STFL-FVR-SD	9975	10347	10069.5	4.33	0
STFL-FVR-SD-RGNS	9829	10117	10024.0	4.71	0
STFL-FVR-ST	10072	10142	10113.5	4.09	0
STFL-FVR-ST-RGNS	10040	10190	**10121.5**	4.36	0
STFL-FULL	10015	10556	10117.5	3.94	0
STFL-FULL-RGNS	9862	10157	10089.5	4.45	0

Continuation of Table 4.2

Config	Branch Coverage			Execs/s	Crash Med
	Min	Max	Med		
	TinyDTLS - DTLS				
BASELINE	561	624	**586.5**	2.23	47.5
BASELINE-RGNS	569	596	580.0	2.36	51.5
STFL-RND	408	590	570.5	2.33	41.0
STFL-RND-RGNS	495	595	574.0	2.37	42.5
STFL-FVR-SD	384	630	559.0	2.39	42.0
STFL-FVR-SD-RGNS	474	584	563.0	2.41	47.0
STFL-FVR-ST	382	587	561.5	2.41	41.5
STFL-FVR-ST-RGNS	482	644	570.0	2.39	43.5
STFL-FULL	503	638	**586.5**	2.46	43.5
STFL-FULL-RGNS	485	585	514.5	2.47	43.0
	DNSmasq - DNS				
BASELINE	1116	1128	**1118.0**	7.29	50.0
BASELINE-RGNS	1115	1128	1116.0	7.40	37.5
STFL-RND	1111	1117	1113.5	5.50	45.5
STFL-RND-RGNS	1114	1128	1116.0	6.03	44.5
STFL-FVR-SD	1111	1126	1115.0	5.72	47.5
STFL-FVR-SD-RGNS	1114	1128	1115.0	6.02	46.5
STFL-FVR-ST	1112	1152	1115.0	6.17	47.0
STFL-FVR-ST-RGNS	1114	1127	1115.0	6.38	48.5
STFL-FULL	1112	1128	1115.0	6.16	51.5
STFL-FULL-RGNS	1115	1127	1115.0	6.27	50.5
	Kamailio - SIP				
BASELINE	8558	9105	8858.5	4.54	0
BASELINE-RGNS	8937	9306	9168.0	4.45	0
STFL-RND	8312	8875	8650.5	4.22	0
STFL-RND-RGNS	8836	9265	8964.0	3.93	0
STFL-FVR-SD	8700	9141	8932.0	4.38	0
STFL-FVR-SD-RGNS	8639	9212	8854.5	4.14	0
STFL-FVR-ST	8344	8683	8516.5	4.26	0
STFL-FVR-ST-RGNS	9409	9792	9515.0	3.97	0
STFL-FULL	8715	8939	8783.5	4.50	0
STFL-FULL-RGNS	9501	10057	**9788.5**	4.28	0

Continuation of Table 4.2

Config	Branch Coverage			Execs/s	Crash Med
	Min	Max	Med		
	Forked-DAAPd - DAAP				
BASELINE	2211	2412	2353.0	0.83	0
BASELINE-RGNS	1908	2555	2472.5	0.84	0
STFL-RND	2203	2360	2275.0	0.88	0
STFL-RND-RGNS	2169	2486	2466.5	0.86	0
STFL-FVR-SD	2122	2347	2275.5	0.88	0
STFL-FVR-SD-RGNS	2429	2514	**2474.0**	0.84	0
STFL-FVR-ST	2223	2382	2335.5	0.86	0
STFL-FVR-ST-RGNS	2380	2517	2468.0	0.85	0
STFL-FULL	2104	2399	2330.0	0.87	0
STFL-FULL-RGNS	2449	2505	2469.0	0.85	0
	Dcmtk - DICOM				
BASELINE	2979	3067	3024.0	12.65	15.0
BASELINE-RGNS	2966	3065	3016.0	10.86	28.5
STFL-RND	3018	3087	3044.0	9.99	2.0
STFL-RND-RGNS	2997	3072	3021.0	9.83	3.0
STFL-FVR-SD	2999	3065	3031.0	13.32	3.0
STFL-FVR-SD-RGNS	2994	3108	**3083.5**	10.02	1.5
STFL-FVR-ST	2989	3049	3011.5	10.03	2.5
STFL-FVR-ST-RGNS	2988	3082	3019.0	12.29	3.0
STFL-FULL	3019	3104	3049.0	13.07	3.0
STFL-FULL-RGNS	2997	3090	3053.5	13.19	5.0

End of Table 4.2

Using these results we can now measure the code coverage impact of specific features.

4.2.2 State feedback feature impact

The discussion presented here will focus on aspects directly related to the research aim; state feedback features. The impact of region mutations are left to Appendix B. This section answers four research questions, one for each state feedback feature and one looking at the impact of all features combined. Impact is reported as a percent change between a child configuration's performance relative to its parent configuration. Coverage speed up is reported where possible, alongisde statistical significance metrics. For brevity, the cov-

erage values themselves are not duplicated here. Interested readers can refer to Table 4.2 for exact coverage values.

RQ 1 - What is the code coverage impact of basic state feedback? This is measured as the median branch coverage difference between one set of configurations, with and without region mutations: BASELINE[-RNGS] (parent) v. STFL-FVR-SD[-RNGS] (child). Table 4.3 shows the impact of the feature as a percent, the coverage speed-up, the p-value from the Mann-Whitney-U test and lastly the Vargha-Delaney effect size. On average, enabling basic state feedback changes coverage performance by -1.77 (RGNS) to -1.89 (no RGNS)%. Results for the region-mutation enabled configurations were mostly statistically significant or nearly so, and the average is skewed by one outlier (PureFTPd), though removing it still yields a negative average coverage change. In both cases, five targets saw coverage improvements, albeit not statisticallly signficant for the configurations without region-mutations.

Recall this measurement comprises two effects; a difference in seed selection approaches and the addition of state modeling and state selection. Previous work has measured the impact of using FAVOR seed selection v. queue order as BASELINE does [15]. Hence, by measuring basic state feedback using STFL-FVR-SD, we can break down the contribution of each effect. Meng *et al.* found FAVOR seed selection, isolated from state-feedback influences, led to a -2.04% coverage change across all targets. In other words, STFL-FVR-SD has a handicap of -2.04% in coverage performance due to the state-heuristics it takes into account for seed selection. Considering this, the impact of the state modeling and state selection aspects in basic state feedback contributed a coverge increase of +0.15% and +0.27%. Fuzzing a singled message targetting a random state improved coverage slightly, though using a seed selection approach informed by target state led to an overall coverage reduction.

RQ 2 - What is the code coverage impact of favoured seed selection? This is measured as the median branch coverage difference between two sets of configurations, each with and without region mutations: STFL-RND[-RGNS] v. STFL-FVR-SD[-RNGS] and STFL-FVR-ST[-RGNS] v. STFL-FULL[-RNGS]. Table 4.4 shows the per-target results. On average, enabling favoured seed selection alone increased branch coverage by 0.08 to 0.88%. Most results across all measurements are not statistically significant, with a p-value above 0.05. Thus we conclude favoured seed selection alone contributes a small and statistically insignificant improvement to fuzzing performance.

RQ 3 - What is the code coverage impact of favoured state selection? This is measured as the median branch coverage difference between two sets of configurations, each with and without region mutations: STFL-RND[-

Table 4.3: Code coverage impact of basic state feedback

	BASELINE[-RGNS] v. STFL-FVR-SD[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
Live555	-1.72	-	0.00	0.97	-1.64	-	0.00	1.00
Exim	-0.52	-	0.60	0.58	0.97	1.25×	0.09	0.27
ProFTPd	-4.70	-	0.00	0.92	-5.28	-	0.00	0.89
LightFTP	-3.47	-	0.07	0.74	-0.29	-	0.47	0.60
PureFTPd	-7.59	-	0.00	0.95	-13.84	-	0.00	1.00
BFTPd	0.10	1.13×	0.91	0.48	0.21	2.29×	0.42	0.39
OpenSSH	0.32	1.28×	0.21	0.33	1.29	2.34×	0.08	0.27
OpenSSL	0.18	1.41×	0.25	0.34	-0.25	-	0.06	0.76
TinyDTLS	-4.69	-	0.15	0.70	-2.93	-	0.02	0.83
DNSmasq	-0.27	-	0.01	0.86	-0.09	-	0.10	0.72
Kamailio	0.83	1.28×	0.43	0.39	-3.42	-	0.00	0.91
Forked-DAAPd	-3.29	-	0.03	0.80	0.06	1.01×	0.65	0.44
Dcmtk	0.23	1.65×	0.47	0.40	2.24	6.13×	0.00	0.11
Avg	-1.89	1.35×		**Avg**	-1.77	2.60×		
	σ =2.62				σ =4.18			

RGNS] v. STFL-FVR-ST[-RNGS] and STFL-FVR-SD[-RGNS] v. STFL-FULL[-RNGS]. Table 4.5 shows the per-target results. On average, enabling favoured state selection alone changes branch coverage by -0.10 to 0.69%. Most results across all measurements are not statistically significant, with a p-value above 0.05. There is one statistically significant outlier; Kamailio, when region-mutations are enabled, saw a 6.15% and 10.55% increase in performance through adding favoured state selection. Thus we conclude favoured state selection, on average, does not effect fuzzing performance.

This aspect is measured by Liu *et al.*, where they found that all AFLNet's state selection algorithms led to similar coverage results [11]. They test six targets; LightFTP, ProFTPd, Exim, OpenSSH, OpenSSL and Live555. Our expanded study agrees with theirs, but does find it may help depending on the target and configuration.

RQ 4 - What is the code coverage impact of full state feedback? This is measured as the median branch coverage difference between one set of configurations, with and without region mutations: BASELINE[-RNGS] (parent) v. STFL-FULL[-RNGS] (child). Table 4.6 shows the results. On average state feedback, with all features enabled, changed code coverage achieved by -1.48 to -1.51%. This is a mild improvement from basic state feedback, brought on from favoured seed and state selection combined. The results for the ma-

Table 4.4: Code coverage impact of favoured seed selection

	STFL-RND[-RGNS] v. STFL-FVR-SD[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
Live555	0.14	1.07×	0.79	0.46	0.93	1.41×	0.06	0.25
Exim	-0.05	-	0.12	0.71	0.56	1.11×	0.07	0.26
ProFTPd	1.19	1.87×	0.58	0.42	-1.82	-	0.85	0.53
LightFTP	0.29	1.31×	0.85	0.47	0.00	-	0.91	0.48
PureFTPd	1.55	1.37×	0.73	0.45	5.71	1.40×	0.35	0.37
BFTPd	0.10	1.13×	0.62	0.43	1.15	3.39×	0.06	0.25
OpenSSH	0.36	1.40×	0.04	0.23	0.18	1.19×	0.73	0.45
OpenSSL	0.12	1.22×	0.60	0.43	0.75	2.09×	0.05	0.24
TinyDTLS	-2.02	-	0.91	0.52	-1.92	-	0.15	0.70
DNSmasq	0.13	1.48×	0.25	0.35	-0.09	-	0.21	0.67
Kamailio	3.25	2.65×	0.00	0.07	-1.22	-	0.08	0.74
Forked-DAAPd	0.02	1.00×	0.63	0.57	0.30	1.07×	0.19	0.32
Dcmtk	-0.43	-	0.12	0.71	2.07	6.13×	0.01	0.15
Avg	0.36	1.45×		**Avg**	0.51	2.22×		
	σ =1.20				σ =1.94			

	STFL-FVR-ST[-RGNS] v. STFL-FULL[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
Live555	0.20	1.33×	0.73	0.45	0.71	1.73×	0.03	0.21
Exim	0.00	-	0.97	0.49	2.43	3.28×	0.00	0.07
ProFTPd	0.64	1.82×	0.35	0.37	0.18	1.46×	0.65	0.44
LightFTP	1.46	1.23×	0.09	0.27	0.15	1.30×	0.79	0.54
PureFTPd	0.57	1.05×	0.57	0.42	1.49	1.16×	0.44	0.39
BFTPd	0.41	1.93×	0.22	0.34	1.78	3.36×	0.02	0.20
OpenSSH	-0.45	-	0.17	0.69	0.33	1.20×	0.38	0.38
OpenSSL	0.04	1.10×	0.38	0.38	-0.32	-	0.11	0.72
TinyDTLS	4.45	2.11×	0.02	0.19	-9.74	-	0.05	0.77
DNSmasq	0.00	-	0.88	0.48	0.00	-	0.84	0.47
Kamailio	3.14	1.95×	0.00	0.00	2.87	1.79×	0.00	0.12
Forked-DAAPd	-0.24	-	0.74	0.45	0.04	1.02×	0.71	0.45
Dcmtk	1.25	6.67×	0.00	0.08	1.14	2.09×	0.21	0.33
Avg	0.88	2.13×		**Avg**	0.08	1.84×		
	σ =1.43				σ =3.11			

jority are statistically significant, with a p value less than or equal to 0.05. Accounting for Meng *et al.*'s finding that state heuristics in seed selection reduces performance by -2.04%, we find other state feedback aspects contribute a small coverage increase of +0.53 to +0.56%.

Table 4.5: Code coverage impact of favoured state selection

	STFL-RND[-RGNS] v. STFL-FVR-ST[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
Live555	0.40	1.69×	0.17	0.32	0.44	1.24×	0.50	0.41
Exim	0.98	1.37×	0.41	0.39	-1.19	-	0.47	0.60
ProFTPd	-1.82	-	0.39	0.62	-2.04	-	0.58	0.58
LightFTP	-0.87	-	0.10	0.72	-0.58	-	0.70	0.56
PureFTPd	0.04	1.01×	0.94	0.52	4.95	1.17×	0.76	0.46
BFTPd	-0.21	-	0.82	0.54	0.00	-	0.57	0.58
OpenSSH	1.01	2.05×	0.01	0.16	0.35	1.48×	0.12	0.29
OpenSSL	0.56	1.61×	0.00	0.13	1.73	4.98×	0.00	0.05
TinyDTLS	-1.58	-	0.47	0.60	-0.70	-	0.71	0.56
DNSmasq	0.13	1.44×	0.20	0.33	-0.09	-	0.73	0.55
Kamailio	-1.55	-	0.09	0.73	6.15	4.02×	0.00	0.00
Forked-DAAPd	2.66	1.35×	0.15	0.31	0.06	1.01×	0.55	0.42
Dcmtk	-1.07	-	0.00	0.90	-0.07	-	0.97	0.49
Avg	-0.10	1.50×		**Avg**	0.69	2.32×		
	σ =1.28				σ =2.34			

	STFL-FVR-SD[-RGNS] v. STFL-FULL[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
Live555	0.45	1.72×	0.23	0.34	0.22	1.24×	0.65	0.44
Exim	1.04	1.55×	0.19	0.32	0.57	1.25×	0.33	0.37
ProFTPd	-2.35	-	0.25	0.66	-0.04	-	1.00	0.50
LightFTP	0.29	1.08×	0.97	0.51	-0.43	-	0.82	0.54
PureFTPd	-0.91	-	0.91	0.48	0.76	1.16×	0.58	0.42
BFTPd	0.10	1.00×	0.62	0.43	0.62	1.36×	0.79	0.46
OpenSSH	0.19	1.17×	0.15	0.31	0.49	1.24×	0.14	0.30
OpenSSL	0.48	1.95×	0.08	0.26	0.65	3.16×	0.02	0.19
TinyDTLS	4.92	2.11×	0.16	0.31	-8.61	-	0.33	0.64
DNSmasq	0.00	-	0.62	0.43	0.00	-	0.45	0.40
Kamailio	-1.66	-	0.08	0.74	10.55	5.83×	0.00	0.00
Forked-DAAPd	2.40	1.61×	0.06	0.25	-0.20	-	0.88	0.53
Dcmtk	0.59	2.38×	0.02	0.19	-0.97	-	0.08	0.74
Avg	0.43	1.62×		**Avg**	0.28	2.18×		
	σ =1.80				σ =3.95			

It remains that a few targets see a large coverage reduction from state feedback; ProFTPd, PureFTPd, LightFTP, TinyDTLS (when region mutations are enalbed). Kamailio, the SIP target, sees a significant coverage benefit from enabling state feedback and region mutations. The fact three

of four FTP targets are not favourable to state feedback could mean the final results of this research are muted. If the results demonstrate increased impact through the additon of syntax-awareness, even for FTP targets, that could mean even greater benefits to targets benefitting from state feedback. At a minimum, one target, BFTPD, does benefit from the addition of state feedback here.

Table 4.6: Code coverage impact of full state feedback

	BASELINE[-RGNS] v. STFL-FULL[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
Live555	-1.28	-	0.00	0.97	-1.42	-	0.00	0.99
Exim	0.51	1.31×	0.48	0.40	1.54	2.05×	0.01	0.16
ProFTPd	-6.95	-	0.00	0.99	-5.32	-	0.01	0.84
LightFTP	-3.19	-	0.04	0.78	-0.72	-	0.34	0.63
PureFTPd	-8.44	-	0.00	0.91	-13.18	-	0.00	1.00
BFTPd	0.21	1.19×	0.73	0.45	0.83	1.46×	0.27	0.35
OpenSSH	0.51	1.61×	0.01	0.17	1.79	4.68×	0.01	0.16
OpenSSL	0.66	2.12×	0.01	0.15	0.40	1.56×	0.25	0.34
TinyDTLS	0.00	-	0.91	0.48	-11.29	-	0.01	0.88
DNSmasq	-0.27	-	0.06	0.76	-0.09	-	0.20	0.67
Kamailio	-0.85	-	0.38	0.62	6.77	3.31×	0.00	0.00
Forked-DAAPd	-0.98	-	0.91	0.52	-0.14	-	0.68	0.44
Dcmtk	0.83	3.21×	0.03	0.20	1.24	2.09×	0.11	0.29
Avg	-1.48	1.89×		**Avg**	-1.51	2.53×		
	σ =2.98				σ =5.45			

These results are less favourable towards the code coverage value of state feedback than those from Meng *et al.*[15], which found a negligible performance decrease of 0.01%. The difference is explained by our inclusion of state-heuristic based seed selection as part of state feedback. The specific configuration of their fuzzer for each target and timeouts used would also influence these results.

If we look at FTP targets only, which will be relevant for our validation, we see full state feedback has an average impact of -4.59% (no RGNS) and -4.60% (RGNS). This negative results is explained by the impact of seed selection informed by state heuristics (FAVOR-mode relative to queue order only). Meng *et al.* found that, on average for FTP targets, this difference caused a coverage decrease of -5.07%.

4.2.3 Discussion

This ablation study concludes that AFLNet state feedback features reduce coverage relative to a BASELINE configuration which mutates a random number of sequential messages in a sequence. This coverage reduction is fully explained by the negative impact of state-heuristic based seed selection relative to queue-order based selection. Accounting for this, remaining feedback features do add a postive coverage effect, though small and often not statistically significant.

Authors have hypothesized the limited impact of state feedback could be due to response codes not being a good state representation [15], fuzz being invalid (syntax) or irrelevant (semantics), and lastly that maybe state feedback will prove more impactful at greater number of executions [11]. This research investigates how syntactically valid fuzz changes the impact of state feedback, under the hypothesis that valid fuzz are needed to reach and engage with program state. Those results are discussed in Section 4.5. For now, we present target-specific challenges and observations while conducting the ablation study:

1. Forked-DAAPd: the coverage analysis portion of some runs took 36+ hours. This is due to long timeouts paired with seeds containing 200+ messages. There are two versions of this target, a kernel-thread version with low stability (likely contributor to large seeds, paired with region mutations) and a user-thread version which has issues completing the code coverage portion. Based on execution speeds, it appears NSFuzz[3] used the kernel-thread version and so we proceeded with that one but had to let coverage analysis run a long time.
2. OpenSSL: commit 6d86ca0 of AFLNet fails to fuzz this target when state feedback is enabled (-E), due to *state_ids_count* equalling zero. Attempts were made to resolve this by studying a pull request to add an additional SSL target (WolfSSL) but without success. For this target only, we reverted to using the ProFuzzBench included AFLNet version (Jan 2021 - commit f2e71ea).
3. BFTPd: disk would fill up causing fuzzing to fail. This is due to log files accumulating. Target *clean.sh* was updated to clear the log files between target executions. % coverage change among configurations was near zero; why? Though BFTPd has 1328 branches, only 378 of them are in the source file which handles command parsing (and AFLNet is only fuzzing the command channel), and so a good branch improvement in the command source results in an overall small branch percent improvement. When fuzz only target portions of the code but coverage is assessed

relative to the entire source then percentages can be misleading. For example, between the worst run of BFTPd (STFL-RND run 8) and the best run (STFL-FULL run 3) there is a 13% coverage difference in the command parsing file.

4. TinyDTLS: regardless of AFLNet version, after a long fuzz time, some fuzzing sessions will crash/quit. Experiments were repeated to get 10 successful runs. Reasons are unknown.
5. ProFTPd, Exim, PureFTPd, BFTPd: initial seeds sometimes timeout during dry-run which causes fuzzing to fail starting. Longer timeouts and/or permitting the target to exceed timeouts during the dry-run (adding + to the *-t* parameter value) resolved this. In an effort to stick to ProFuzzBench included commands, experiments were simply restarted to get 10 successful starts.
6. DNSmasq: very similar median coverage results across all configurations. DNS has very few response codes; only 1 for 'success' and 9 for different failures. The DNSmasq config used by ProFuzzBench does not contain any dns records, so most of its functionality would not be reached. Results were very skewed across runs, with the min run being very close the median, but most configs have a max run much greater than the median. Similar to BFTPd, with this target we see the fuzzing only execute a small subset of the overall program banches. The branch different between the min and max BASELINE runs were all in one specific file (rfc1035.c) while all other source file coverage is identical. rfc1035.c represents 9.2% of the branches in DNSmasq's source.

Verifying that experiments have succesfully started simply requires running the *top* utility and confirming there are 10 *afl-fuzz* processes running. Afterwards, the data processing scripts verify the fuzzing duration in seconds based on the fuzzer output directory logs.

4.3 AFLNet-Packmute

This section presents the results of developing our syntax-aware AFLNet variant, AFLNet-Packmute. First, details on how the tool is developed are discussed followed by a look at how we arrived at the final design presented in Chapter 3.

4.3.1 Development

PcapPlusPlus and Libpackmute are written in C++, compiled statically and installed on the local system using the CMake build system generator.

To build *afl-fuzz*, we use the AFLNet included Makefile with modifications. The object file must be compiled with a C compiler and then linked against PcapPlusPlus and Libpackmute using a C++ linker. The modified Makefile splits the compilation and linking steps for the *afl-fuzz* target. Listing 5 shows the object file target being compiled with a C compiler then the binary target being linked with a C++ compiler.

Listing 5 Afl-fuzz makefile target

```
afl-fuzz.o: afl-fuzz.c $(COMM_HDR)
	$(CC) $(CFLAGS) -c afl-fuzz.c -o $@

afl-fuzz: afl-fuzz.c $(COMM_HDR) afl-fuzz.o aflnet.o aflnet.h
	$(CXX) aflnet.o afl-fuzz.o -o $@ $(LDFLAGS) \\
	-lpackmute -lPacket++ -lCommon++
```

Early in development, we found an error in PcapPlusPlus' handling of command-only FTP messages. When accessing the option field, instead of returning an empty string it crashes the application. This is due to it comparing a zero-indexed postion field against a length field. We patched the issue, added a testcase to the project test suite, and our pull-request was merged into the project, resolving the issue.

Stability is important since the application needs to run for 24 hours. Valgrind's Memcheck [37] tool is used, alongside AFL's built-in head/tail allocation canaries to catch memory management errors. An integer overflow bug proved most problematic and was only found through manual print statements. To conduct fuzzing with AFLNet-Packmute, a new ProFuzzBench Dockerfile is created which compiles and installs PcapPlusPlus and Libpackmute before the fuzzer is finally compiled.

4.3.2 Syntax-awareness

How syntax-awareness is applied to the mutation process in general was clear prior to development. Specific opportunities to reduce the likelihood of breaking syntax were discovered and addressed one-by-one. Two questions arose regarding exploitation v exploration that had less defined answers; where in the process? and how often?

Where? ChatAFL chose to situate the explore-exploit decision at the start of the havoc stage; a decision is made on the first havoc stage and after each splice cycle. We placed the exploit-explore decision at the start of a

fuzzing round, so that it persists through the splice cycle as well. We do not believe that this decision had a particular advantage; it was necessary due to our design. Mutable range information is stored alongside the queue entry struct when a seed is added to the corpus; we did not parse ranges from arbitrary buffers while fuzzing. Thus, after splicing the only knowledge of mutable ranges of that buffer are found in the range structs. If we go from exploring a buffer, which is spliced with another, to exploiting, the mutable ranges within it are unknown, since to this point we've treated it as one big mutable range. This is why we could not place the exploit-explore decision at the start of the havoc stage. Though Libpackmute does have the ability to parse mutable ranges from buffer while fuzzing once converted to replay format, a strength of our design is saving that effort and placing the exploit-explore decision on a fuzzer round-basis made it possible.

How often? Though their real explore-exploit ratio is unclear due to design decisions, ChatAFL's authors chose a 50-50 exploit-exploit ratio, meaning half the havoc stages initially used syntax awareness while half did not. The authors of the paper were contacted to know if this ratio had been experimentally determined or simply selected as a good starting point. We have not yet received a response.

Literature on the exploitation-exploration tradeoff suggests the focus on one or the other should vary across the system's lifespan depending on the nature of its environment and its learning [38]. Berger-Tal *et al.* found that generally, to maximize the production of energy, a system should first focus on building knowledge early in its life (exploring) and later focusing on just converting that knowledge to energy (exploiting). This is similar to the approach used in *Directed Greybox Fuzzing* by Böhme *et al.* which used a new power schedule to progressively focus more on seeds which where closer to target code branches [39]. Exploring early on enabled them to avoid getting stuck in a local minimum. In their case however, there was a clear 'knowledge' the fuzzer was gaining; seeds close to the target code branches. In AFLNet-Packmute, there is no such knowledge gained; the fuzzer has a fixed syntax knowledge. While fuzzing, seeds that reach new coverage are found (a form of knowledge) but it is really always trying to produce energy (coverage).

To investigate if this approach has value for syntax-aware fuzzing we conducted a simple experiement. Fuzz using two different exploit-explore configurations where each configuration had an average exploitation rate of 75% over the lifespan of the fuzzing. One configuration used a fixed 75% exploit rate, named *75ee*, and the other a linearly increasing exploit rate from 0-100% over the first half of the lifespan and 100% thereafter, named *linEE*. Both use a BASELINE configuration of AFLNet-Packmute fuzzing the LightFTP

target and was repeated 10 times each, lasting 24 hours. Figure 4.2 shows the median branch coverage over time for each fuzzer. The fuzzer with a linearly increasing exploit rate from 0-100% for the first 12 hours is in orange (named *linEE*), whereas the fixed 75% exploit rate is shown in blue (named *75ee*). The final performance difference is small (*linEE* achieving 0.41% fewer branches) and statistically insignificant. The coverage progress however is very interesting. Once *linEE* enters 100% exploitation at 720 minutes (12 hrs) we see the coverage essentially plateaus, while it was making good progress during exploration. *75ee* meanwhile makes progress throughout. There appears to be coverage value in exploration throughout the fuzzer lifespan. This may be due to some of the reasons other literature has opted to keep some invalid fuzz; targets diverging from protocol definitions, fuzzing edge cases or protocol parsing logic [4],[6],[15].

We leave further investigation as future work and proceed with tuning the exploit-explore ratio as a fixed value applied throughout the fuzzer lifespan. Using AFLNet-Packmute's *-z* switch, we conducted experiments using an exploit rate of 25, 50, 75 and 100%, respectively named *%ee*. The BASELINE configuration of AFLNet from the ablation study is included to compare against exploration-only. Figure 4.3 shows the median branch coverage over time for each exploit rate, overlaid with the min/max at each timestep in a shaded region of the same colour as the median line. The plot omits the first hour so that the axis can be better scaled to see detail.

The configuration with an exploit rate of 75% performed best, though not statistically significant relative to the base-case of exploration only. *75ee* had a median performance equal to *50ee* but superior minimum and maximum performance. Table 4.7 shows to final metrics of the experiments; min run, max run, median, execs/second. We see the difference between exploiting fuzzers and the base-case of AFLNet is not statistically significant, though greatest for *75ee*. Why the performance drop at a 100% exploit rate? This question led to an interesting discovery. Exploration in this specific context (AFLNet + LightFTP) has an ability exploitation does not have; to introduce entirely new commands not found in the initial seeds. This is because it can overwrite existing FTP commands with dictionary words that are valid FTP commands. In the case of FTP targets, ProFuzzBench provides the fuzzer a dictionary which contains the command keywords for several other message types not found in the initial seeds. When exploring, these are inserted or overwritten into the buffer to form new commands and reach significant portions of code. We expect this advantage of exploration is reduced for targets with more feature complete initial seeds or more complex protocols where a simple keyword insertion does not create an entirely new message type. For exploitation to

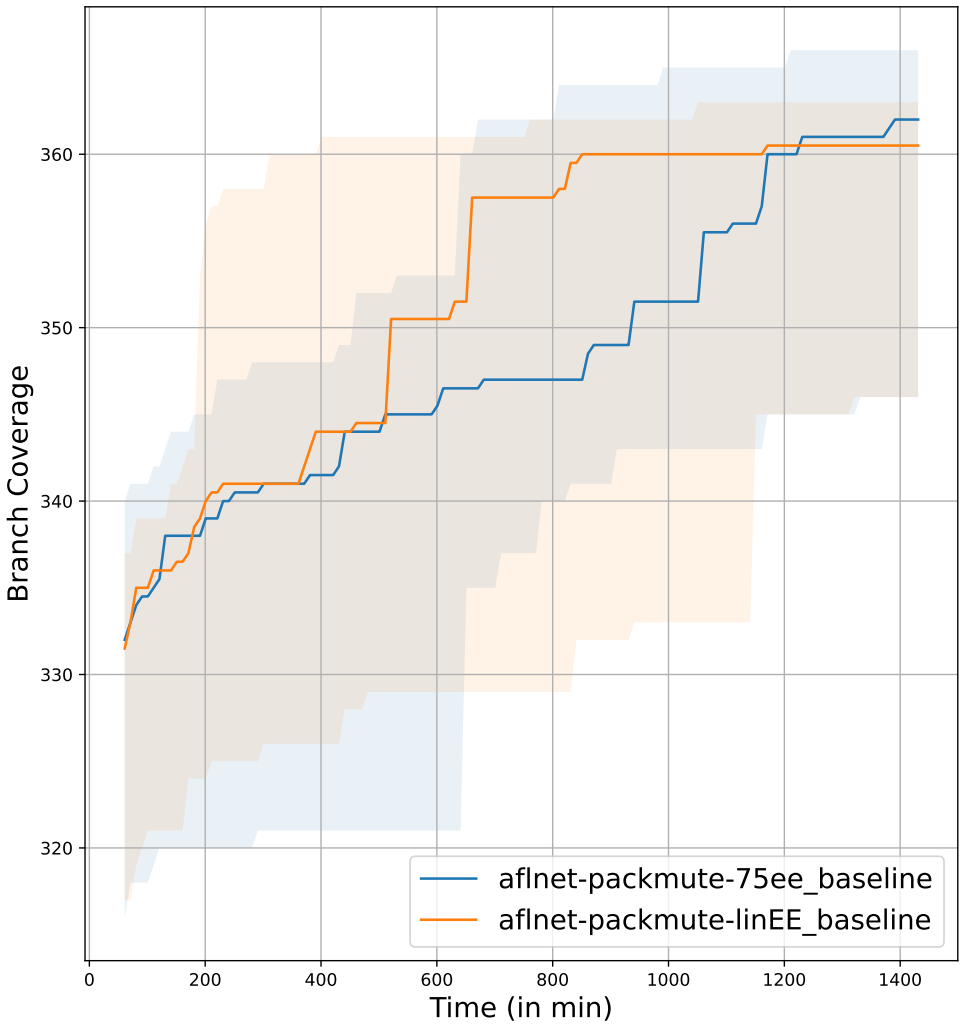

Figure 4.2: Constant v linearly increasing exploit rate in AFLNet-Packmute

achieve this, the fuzzer would need to add the FTP command to a mutable range,create a message terminator previous to it, and then add a space after it; a very unlikely occurence.

Why then is 100% exploitation not much worse than 0% if it can't reach the code for several entire FTP commands? The cause is syntactically invalid fuzz being introduced from two sources, even when AFLNet-Packmute tries to apply syntax all the time: 1) The grouping of many messages into one if the

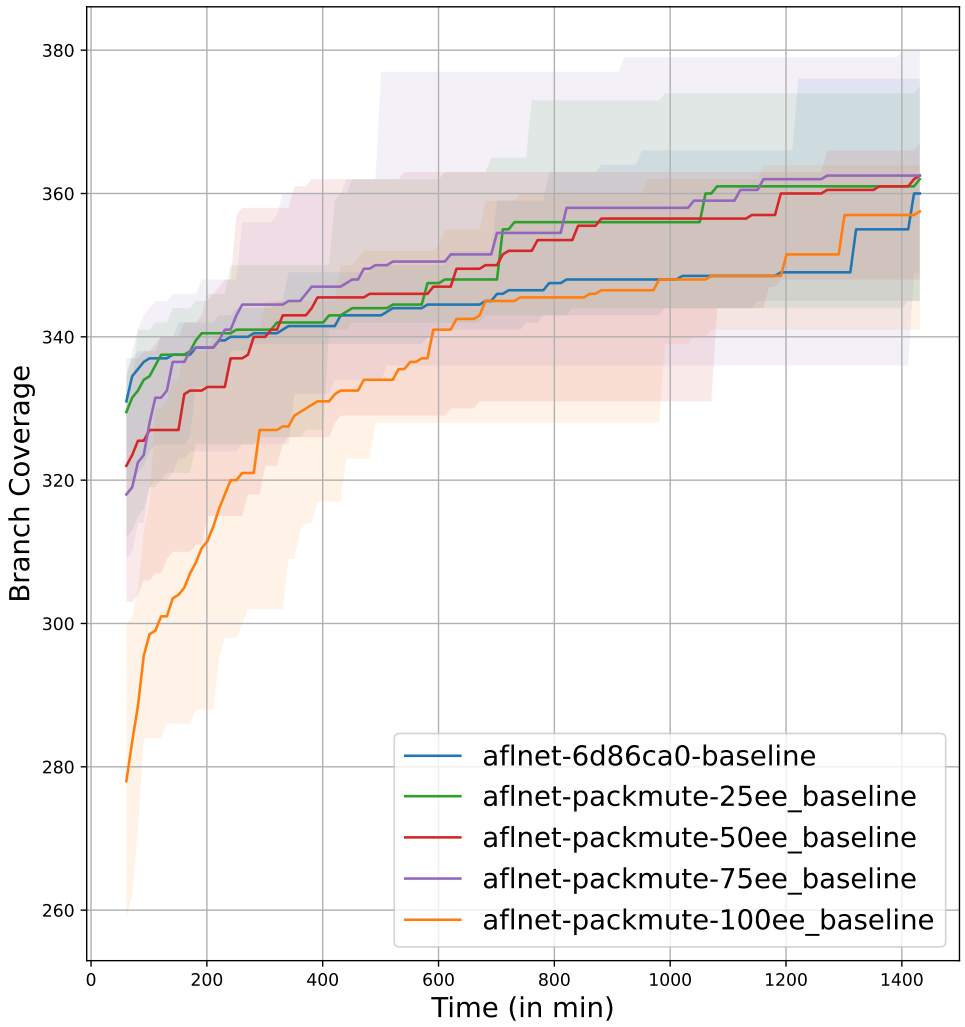

Figure 4.3: Tuning a fixed exploit rate in AFLNet-Packmute against the LightFTP target

mutation buffer has introduced too many messages. These messages are saved into a single region which may or may not be syntactically valid and have one big mutable range with a bunch of messages. LightFTP rejects these as having bad command syntax. On later fuzzing rounds these multi-message regions are added to the mutation buffer through region mutations. 2) Random mutations creating message terminators. FTP messages are terminated by the byte sequence *0x0d0x0a* which can be introduced into a mutation buffer

Table 4.7: AFLNet-Packmute performance across exploit rates

Config	Min	Max	Median	%Δ v. 0ee	Execs/s	p	$\hat{A}_{12}$
afl	345	376	360	-	5.59	-	-
25ee	345	375	362	0.56	5.68	0.43	0.39
50ee	348	367	362.5	0.69	5.76	0.20	0.33
75ee	349	380	362.5	0.69	5.69	0.17	0.32
100ee	341	364	357.5	-0.69	6.81	0.32	0.64

through, for example, the insertion of *0x0a* bytes followed by an incrementation. AFLNet-Packmute parses these into individual messages and sends syntactically invalid fuzz to the target, which then get saved to the corpus. These syntactically invalid regions can later be added to fuzz from region mutations, splice into mutation buffers or seed entire fuzzing rounds. If there is no mutable range, AFLNet-Packmute would default to mutating the entire buffer; just like exploration.

The first cause of invalid fuzz can be addressed by removing the max new message constraint when parsing and inserting the mutation buffer into the message linked list, however we deem this is a change to AFLNet beyond adding syntax-awareness so it was left out of scope. Also, removing this constraint would further explode the size of seeds when region mutations are enabled. The second cause could be addressed by sanitizing the mutation buffer at the end of the mutation loop before the buffer is parsed; removing any *0x0d0x0a* found within any mutable ranges. As will be discussed in 4.4, we find invalid fuzz caused by this represents a very small proportion of the overall fuzz and would be even less likely in protocols with more complex message terminators (e.g. those defined by header fields). Thus, we pursued verification and validation using a fixed 75% exploit rate.

4.4 Verification

Recall from Chapter 3 the verification of this research involved three steps. First, the scripts that produce the final metrics and coverage plots are verified. Second, AFLNet-Packmute's code is verified for correctness through test driven development and built-in syntax checks. Third, a final verification of AFLNet-Packmute is conducted against a real FTP server to confirm fuzz produced are syntactically valid when exploiting and resemble AFLNet when exploring.

The analysis scripts comprised coverage summary statistics scripts (final

min, max, median branch coverage, statistical significance) and coverage plotting scripts (min, max, median branch coverage over time). These are independently derived from the ProFuzzBench coverage time series output, thus they should agree with each other. During development, comparing these outputs proved useful in finding errors in each other. The plots pointed to errors in speed-up calculations. For example if the speed-up between configuration A and B is 2×, then we expect a horizontal line drawn from the end value of B to intersect A's curve at the 12 hour mark. At first, the speed-up script was not preserving the sequence of values correctly before appending them to the statistics CSV. The summary statistics pointed to errors in how the plot legend was populated, assigning incorrect labels to each line.

Verification of AFLNet-Packmute started with the test-driven development of Libpackmute. No tests were directly written for *lpm_setup_env()* as it simply sets a variable, and all other functions fail immediately if it has not first been called. 15 tests where written to verify Libpackmute's three other public functions:

1. *lpm_check_syntax()*: Four tests of good syntax; ProFuzzBench intial seed, single byte FTP command, FTP arguments including null bytes, FTP arguments containing newline characters, and valid message sequences thousands of bytes long. Seven tests of bad syntax; invalid FTP command, the presence of an option where there shouldn't be, the absence of an option where there should be, invalid replay format, invalid FTP message terminator, random bytes, and random bytes in replay format.
2. *lpm_find_mutable()*: Three tests comprising valid messages with no mutable ranges, valid messages with mutable ranges, and a sequence with a mix of invalid and valid message containing mutable ranges.
3. *lpm_get_pre_padding()*: One test verifying the correct byte is provided for FTP.

Using CMake's CTest program, these tests were executed as development progressed to ensure new changes did not break existing functionality. Now that we had a verified function that checks the syntax validity of a buffer we could use it to verify our addition of syntax-awareness to AFLNet. As described in Section 3.4 a call to *lpm_check_syntax()* was made with each fuzz generated. This found errors in the FTP regex patterns and an incorrect parameter setting in a call to PcapPlusPlus' *getCommandOption()* method. These checks also showed the combining of messages that occurs where multiple regions are grouped into a single message if the total buffer's region count exceed the maximum.

The nature of writing code which handles randomly changing buffers is that it fuzzes itself. Simply running AFLNet-Packmute revealed logic errors in conditional statements, limitations to *std::regex_search*'s length capacity, and an integer overflow error. These improved the stabilty and correctness of AFLNet-Packmute.

The last step was to verify AFLNet-Packmute against a FTP server. Each AFLNet, AFLNet-Packmute (expoit = 0%) and AFLNet-Packmute (exploit = 100%) fuzz the LightFTP target for 10 minutes while network traffic is recorded. A STFL-FULL-RGNS configuration is used. This is repeated three times and averaged. This verifies the number of commands rejected by the server due to bad command syntax. For an exploit rate of 100% we expect this number to be near zero, while for an exploit rate of 0% we expect it to be comparable to AFLNet. As is discussed in Section 4.3 bad commands can be introduced from AFLNet's maximum new message constraint as well as the default behaviour to mutate the entire buffer if no mutable ranges were found. For the purposes of verification, the fuzzer with a 100% exploit rate has the maximum new message constraint removed and the default behaviour is to skip selected seeds that have no mutable ranges.

The following FTP response codes were counted:

- 500 - Syntax error, command unrecognized
- 501 - Syntax error in parameters or arguments
- 503 - Bad sequence of commands
- 504 - Command not implemented for that parameter
- 530 - Not logged in
- 550 - Requested action not taken

Table 4.8 shows the average number of requests sent by each fuzzer and the average number of response codes of each type as a percent of the requests sent. We expect the proportion of each to change over time, but 10 minutes is sufficient to verify AFLNet-Packmute is working as intended.

With an exploit rate of 0%, Packmute-XplR should resemble AFLNet since it is always exploring (XplR). In terms of response codes as a percent of requests sent, it mostly is in line with AFLNet, although there is a small reduction in requests being rejected for bad command syntax and an increase in requests sent. This may be due to one run for Packmute-XplR having a high request count and one run of AFLNet having a low request count creating that gap. A given run might favour seeds with more or less messages due to coverage discoveries. In the case of both fuzzers, they complete 1 queue cycle over the 10 minutes of fuzzing; meaning each initial seed is used once to start the fuzzing round.

With an exploit rate of 100%, Packmute-XplT should see nearly no bad commands. The only source of bad commands we expect is the random creation of new message terminators as described in Section 4.3. On average, 0.3% of commands sent are rejected for bad command syntax. This varies across runs, with one run seeing 0.7% and another seeing 0%. This makes sense if a random message terminator is introduced into a buffer early in the fuzzing session, creating invalid fuzz which is then regularly introduced into the fuzz through region mutations.

Table 4.8: AFLNet-Packmute real FTP server verification

Fuzzer	Requests Sent	Response Code %					
		500	501	503	504	530	550
AFLNet	80 942	18.8	5.3	0.1	0.01	18.5	1.1
Packmute-XplR	85 121	17.9	5.7	0.01	0.01	17.7	1.5
Packmute-XplT	100 591	0.3	8.1	0	0	30.8	6.2

Beyond verification, these results reveal two important things. First, that exploration can create behaviour exploitation cannot such as the 503 and 504 response codes, resulting from it introducing new message types via dictionary insertion/overwriting. Second, that while exploiting, many syntactically valid fuzz are now rejected due to being semantically irelevant. Though AFLNet-Packmute sees a response code 500 reduction of 18.5%, the occurrence of 530 response codes increases by 12.3%. Considering commands rejected for syntax or the lack of a valid login session (500 + 530) adding syntax-awareness has shifted the total from 37.3% to 31.1% of commands being rejected. This suggests to us adding some semantic awareness has great potential to increase the quantity of effective fuzz. The low throughput of protocol fuzzing paired with more than a third of fuzz being rejected due to syntax or semantics makes for limited coverage progress. We must keep in mind these observations are from a short 10 minutes of fuzzing. Our intuition is that at longer fuzzing durations the proprotion of fuzz rejected due to response code 530 (please login) would reduce. Over time, the fuzzer would exhaust easily reached code coverage outside of valid login sessions, giving more time to seeds and states that have valid logins.

This verification confirms AFLNet-Packmute mimics AFLNet when exploring and creates a negligible number of bad commands when exploiting. We can continue to the validation phase knowing AFLNet-Packmute works as expected.

4.5 Validation

This section is organized in three parts. First, the coverage results of AFLNet-Packmute with a 75% exploit rate fuzzing the four FTP targets are presented. Second, the code coverage impact of state feedback features in AFLNet-Packmute is measured. Lastly, the change in coverage impact between AFLNet and AFLNet-Packmute can be calculated to allow us to fulfill the aim of this research.

4.5.1 Coverage Results

The fuzzing performance results of AFLNet-Packmute with a 75% exploit rate are presented in Table 4.9. The minimum, maximum and median final branch coverage value across the ten runs of each configuration are listed, alongisde the executions per second and median number of crashing test-cases found. In bold are the best performing configurations for each target. Appendix A compares these values directly to the ablation study results to derive the coverage impact of adding syntax awareness to AFLNet's mutation process and also does a comparison to ChatAFL-CL1, the only other syntax-aware AFLNet variant.

Table 4.9: Validation performance results

Config	Branch Coverage			Execs/s	Crash Med
	Min	Max	Med		
	ProFTPd - FTP				
BASELINE	5248	5485	**5324.5**	3.23	0
BASELINE-RGNS	4333	5115	4788.0	2.49	0
STFL-RND	5167	5526	5297.5	3.21	0
STFL-RND-RGNS	4937	5150	4971.0	1.80	0
STFL-FVR-SD	4921	5290	5094.5	2.73	0
STFL-FVR-SD-RGNS	4990	5304	5088.0	2.13	0
STFL-FVR-ST	4334	5290	4745.5	3.28	0
STFL-FVR-ST-RGNS	4840	5194	5021.5	1.99	0
STFL-FULL	4801	5116	4909.0	3.01	0
STFL-FULL-RGNS	4966	5280	5115.0	2.30	0

Continuation of Table 4.9

Config	Branch Coverage			Execs/s	Crash Med
	Min	Max	Med		
	LightFTP - FTP				
BASELINE	349	380	**362.5**	5.69	0
BASELINE-RGNS	348	374	354.0	5.35	0
STFL-RND	327	360	348.5	5.76	0
STFL-RND-RGNS	349	364	350.5	4.95	0
STFL-FVR-SD	346	363	358.0	6.24	0
STFL-FVR-SD-RGNS	335	363	348.5	5.04	0
STFL-FVR-ST	326	365	345.5	6.00	0
STFL-FVR-ST-RGNS	335	366	348.5	4.87	0
STFL-FULL	330	363	358.0	5.96	0
STFL-FULL-RGNS	347	372	354.5	5.18	0
	PureFTPd - FTP				
BASELINE	1272	1344	1290.0	5.18	0
BASELINE-RGNS	1285	1327	**1304.0**	3.67	0
STFL-RND	1023	1203	1075.5	4.40	0
STFL-RND-RGNS	952	1149	1041.5	2.52	0
STFL-FVR-SD	857	1135	1080.0	3.93	0
STFL-FVR-SD-RGNS	1052	1199	1108.5	2.62	0
STFL-FVR-ST	1038	1163	1101.5	3.35	0
STFL-FVR-ST-RGNS	956	1153	1098.5	2.62	0
STFL-FULL	1045	1296	1121.5	3.85	0
STFL-FULL-RGNS	1059	1195	1130.5	2.60	0
	BFTPd - FTP				
BASELINE	471	498	481.0	3.06	26
BASELINE-RGNS	459	496	485.0	3.15	29.5
STFL-RND	439	503	482.5	4.10	16
STFL-RND-RGNS	460	495	484.0	2.54	22
STFL-FVR-SD	476	511	486.5	4.02	12.5
STFL-FVR-SD-RGNS	478	491	486.5	2.96	17
STFL-FVR-ST	454	492	**487.0**	4.18	11
STFL-FVR-ST-RGNS	473	488	482.5	2.41	18.5
STFL-FULL	476	499	482.5	4.05	13.5
STFL-FULL-RGNS	473	489	481.5	2.83	24

End of Table 4.2

4.5.2 State Feedback Impact

Using the coverage performance results we calculate the median branch coverage change between two configurations to determine the impact of that feature. This section answers four research questions, one for each state feedback feature and one looking a the impact of all features combined. Impact is reported as a percent change between a child configuration's performance relative to it's parent configuration. Coverage speed up is reported where possible, alongisde statistical significance metrics. For brevity, the coverage values themselves are not duplicated here. Interested readers can refer to Table 4.9 for exact coverage values. These feature impacts will be later compared to AFLNet's feature impact.

RQ 1 - What is the code coverage impact of syntax-aware basic state feedback? This is measured as the median branch coverage difference between one set of configurations, with and without region mutations: BASELINE[-RNGS] (parent) v. STFL-FVR-SD[-RNGS] (child). Table 4.10 shows the impact of the feature as a percent, the coverage speed-up when possible, the p-value from the Mann-Whitney-U test and lastly the Vargha-Delaney effect size.

For three targets, enabling state feedback reduced coverage, all statistically significant with a p value below 0.05. BFTPd saw a small but statistically insignificant coverage increase. On average, coverage changed by -5.05 to -5.76% as a result of enabling basic state feedback in our syntax-aware fuzzer AFLNet-Packmute.

Table 4.10: Code coverage impact of syntax-aware basic state feedback

	BASELINE[-RGNS] v. STFL-FVR-SD[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
ProFTPd	-6.64	-	0.00	1.00	-3.95	-	0.00	0.91
LightFTP	-1.24	-	0.02	0.80	-1.55	-	0.04	0.77
PureFTPd	-16.28	-	0.00	1.00	-14.99	-	0.00	1.00
BFTPd	1.14	2.55×	0.10	0.28	0.31	1.25×	0.97	0.51
Avg	-5.76	2.55×		**Avg**	-5.05	1.25×		

RQ 2 - What is the code coverage impact of syntax-aware favoured seed selection? This is measured as the median branch coverage difference between two sets of configurations, each with and without region mutations: STFL-RND[-RGNS] v. STFL-FVR-SD[-RNGS] and STFL-FVR-ST[-RGNS] v. STFL-FULL[-RNGS]. Table 4.11 shows the per-target results.

Nearly half the results are or are almost statistically significant, and on average the impact ranges from +1.56% to +1.99%. Since favoured seed selection incorporates aspects of AFL's proven seed selection, we do expect this to improve performance relative to random selection. Only PureFTPd consistently sees coverage improvement across all measures. BFTPd, ProFTPd, and LightFTP see a coverage decrease in at least one measurement.

Table 4.11: Code coverage impact of syntax-aware favoured seed selection

	STFL-RND[-RGNS] v. STFL-FVR-SD[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
ProFTPd	3.82	3.30×	0.02	0.19	-0.13	-	0.85	0.47
LightFTP	2.73	1.17×	0.06	0.25	-0.57	-	0.04	0.78
PureFTPd	0.42	1.15×	0.71	0.56	6.43	3.25×	0.06	0.25
BFTPd	0.83	2.43×	0.45	0.40	0.52	1.5×	0.60	0.43
Avg	+1.95	2.01×		**Avg**	+1.56	2.38×		
	STFL-FVR-ST[-RGNS] v. STFL-FULL[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
ProFTPd	3.45	3.30×	0.22	0.33	1.86	2.86×	0.06	0.25
LightFTP	3.62	2.06×	0.08	0.27	1.72	1.93×	0.04	0.23
PureFTPd	1.82	1.10×	0.35	0.37	2.91	1.28×	0.19	0.32
BFTPd	-0.92	-	0.97	0.51	-0.21	-	0.91	0.52
Avg	+1.99	2.15×		**Avg**	+1.57	2.02×		

RQ 3 - What is the code coverage impact of syntax-aware favoured state selection? This is measured as the median branch coverage difference between two sets of configurations, each with and without region mutations: STFL-RND[-RGNS] v. STFL-FVR-ST[-RNGS] and STFL-FVR-SD[-RGNS] v. STFL-FULL[-RNGS]. Table 4.12 shows the per-target results. Most results are not statistically significant, though overall favoured state selection has increased coverage achieved by 0.4 to 0.80 %. As with favoured seed selection, the four measurements for this feature impact show different targets benefitting under different fuzzer configurations. Each target benefits when favoured state selection is added to certain configurations, while only PureFTPd benefits in all cases.

RQ 4 - What is the code coverage impact of syntax-aware full state feedback? This is measured as the median branch coverage difference between one set of configurations, with and without region mutations:

Table 4.12: Code coverage impact of syntax-aware favoured state selection

	STFL-RND[-RGNS] v. STFL-FVR-ST[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
ProFTPd	-0.89	-	1.00	0.50	-1.43	-	0.09	0.73
LightFTP	-0.86	-	0.34	0.63	-0.57	-	0.02	0.82
PureFTPd	2.42	1.44	0.34	0.37	5.47	2.27	0.44	0.39
BFTPd	0.93	1.57	0.94	0.52	-0.31	-	0.40	0.62
Avg	0.40	1.51×		**Avg**	0.79	2.27×		
	STFL-FVR-SD[-RGNS] v. STFL-FULL[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
ProFTPd	-1.25	-	0.01	0.85	0.53	1.50	0.74	0.45
LightFTP	0.00	-	0.70	0.56	1.72	1.93	0.07	0.26
PureFTPd	3.84	1.30	0.09	0.27	1.98	1.18	0.62	0.43
BFTPd	-0.82	-	0.40	0.62	-1.03	-	0.14	0.70
Avg	0.44	1.30×		**Avg**	0.80	1.54×		

BASELINE[-RNGS] (parent) v. STFL-FULL[-RNGS] (child). Table 4.13 shows the per-target results. On average enabling all state feedback features led to a coverage decrease of -4.43 to -5.45%, most results being statistically significant. ProFTPd and PureFTPd see a large coverage decrease, while LightFTP and BFTPd either see a small increase or decrease depending on whether region mutations are enabled.

Table 4.13: Code coverage impact of syntax-aware full state feedback

	BASELINE[-RGNS] v. STFL-FULL[-RNGS]							
	No RGNS				RGNS			
Target	Impact %	Speed-up	p	$\hat{A}_{12}$	Impact %	Speed-up	p	$\hat{A}_{12}$
ProFTPd	-7.80	-	0.00	1.00	-3.45	-	0.00	0.89
LightFTP	-1.24	-	0.02	0.82	0.14	1.15×	0.85	0.53
PureFTPd	-13.06	-	0.00	0.93	-13.31	-	0.00	1.00
BFTPd	0.31	1.08×	0.38	0.38	-0.72	-	0.21	0.67
Avg	-5.45	1.08×		**Avg**	-4.34	1.15×		

4.5.3 Change in state feedback feature impact

In this section we present the change in state feedback feature impact resulting from the addition of syntax-awareness to AFLNet'mutation process. Though many of the results are not statistically significant, the fact that the separate measurements generally agree about feature impact gives us additional confidence in the results. We consider the average feature impact across all measurements, but also separate it by configurations with and without region mutations enabled. These results achieve the aim of the research.

Table 4.14 lists the results by feature as an overall impact change, alongside the impact change grouped by configurations with and without region mutations. We find that syntax-awareness decreases the impact of basic state feedback (impact change of -1.04%) while it increases the impact of favoured seed and favoured state selection (impact changes of +0.84% and +0.77% respectively). Region mutation enabled configurations saw a lessened impact change, while configurations without region mutations saw a stronger impact change (more negative and more positive).

Regarding basic state feedback, the impact decrease is largely due to PureFTPd which saw a substantial decrease in performance between BASELINE to STFL-FVR-SD when syntax-awareness is added; -7.59% without syntax down to -16.28% with syntax. Other targets saw a blend of small impact increases and decreases. The root cause of this for PureFTPd we believe is the large difference in target executions achieved over the 24 hours of fuzzing. Using AFLNet, BASELINE to STFL-FVR-SD saw a decrease of 0.63 executions per second, whereas AFLNet-Packmute saw a decrease of 1.25 executions per second (double the slowdown). Syntax-aware STFL-FVR-SD against PureFTPd better succeeds at interacting with the program's logic, meaning each execution is slower. If we look at the run of each BASELINE and STFL-FVR-SD which most closely reaches the median performance and consider the slowest target execution seen, we see in AFLNet BASELINE has a slowest execution of 765ms while STFL-FVR-SD has a slowest execution of 871ms (14% slower). For AFLNet-Packmute BASELINE has a slowest execution of 601ms while STFL-FVR-SD has a slowest execution of 930ms (55% slower). Looking at the specific code branches reached by AFLNet-Packmute, but not AFLNet, we see many of them are loops which would cause a decrease in fuzzing speeds. Interestingly, when region mutations are enabled, this slowdown in fuzzing speeds is also present, though syntax-awareness makes region mutations so much more positively impactful to coverage that it overcomes the effect of slower fuzzing.

Though syntax-awareness has worsened the impact of basic state feedback,

Table 4.14: Change in code coverage impact of state feedback features from adding syntax-awareness (FTP Targets only)

Feature	AFLNet	AFLNet-Packmute	Impact Δ
Basic State Feedback	-4.36	-5.40	**-1.04**
RGNS	-4.80	-5.05	-0.25
No RGNS	-3.92	-5.76	-1.84
Favoured Seed	+0.93	+1.77	**+0.84**
RGNS	+1.08	+1.57	+0.49
No RGNS	+0.78	+1.97	+1.19
Favoured State	-0.16	+0.61	**+0.77**
RGNS	+0.41	+0.79	+0.38
No RGNS	-0.72	+0.42	+1.14
All Features	-4.60	-4.89	**-0.29**
RGNS	-4.60	-4.34	+0.26
No RGNS	-4.59	-5.45	-0.86

it has improved the impact of favoured seed and favoured state selection. Regarding favoured seed selection, the impact increase is mostly from ProFTPd and LightFTP which usually saw an important impact increase from adding syntax-awareness. BFTPd and PureFTPd meanwhile had a mix of impact changes. The improvement to favoured state selection feature impact is consistent across all four targets, with some variation across measurments. Importantly, favoured state selection goes from have a negative impact to code coverage to a positive impact. This supports our hypothesis that syntax-aware fuzzing would better engage with the target's statefullness.

Overall, enabling all state feedback features resulted in a decreased impact of -0.29% after adding syntax-awareness. The individual positive impact changes of each favoured state and seed selection did not jointly overcome the decrease in impact resulting from adding basic state feedback. The region mutation enabled configurations disagree with this conclusion, seeing instead a small impact increase of +0.26% from adding syntax-awareness, while the configuration without region mutations saw a larger impact decrease of -0.86%.

4.6 Discussion

The motivation for this research was to understand why improvements to AFLNet's state feedback features have had limited impact to code coverage achieved. The hypothesis we tested was that state feedback features have been ineffective due to syntactically invalid fuzz, unable to reach and engage with program state. We did this by measuring the impact of state feedback features in an ablation study, developing a syntax-aware protocol fuzzer based on AFLNet called AFLNet-Packmute, verifying our implementation and lastly validating the research.

The ablation study revealed that state feedback overall, on average, reduces the coverage achieved by AFLNet. This is driven by the selection of seeds based on state heuristics vice queue order, which achieves better coverage. State modelling aspects of basic state feedback has a small postivie coverage effect. Relative to random selection, the state-informed FAVOR mode of each state and seed selection have a positive code coverage impact. During development of our syntax-aware fuzzer AFLNet-Packmute, we found that exploration has value throughout the entire fuzzing session and that for the LightFTP target a 3:1 ratio of syntactically valid to random fuzz is optimal. This was implemented as a fixed exploit rate of 75%, a decision made at the start of each fuzzing round, after a seed is selected. Unlike the existing syntax-aware AFLNet-based fuzzer, we enforced syntax validity through all mutations and including the splicing mutation stage.

The verification of AFLNet-Packmute confirmed it produces syntactically valid fuzz when exploiting and resembles AFLNet when exploring. The verification phase additionally showed that although adding syntax awareness does reduce the overall fuzz rejected by the server at initial stages, most, though not all of the now syntactically valid fuzz are rejected due to semantics. This points to valuable future work to improve the quantity of effective fuzz through the addition of semantic knowledge. In the validation phase we found, for FTP targets, that though syntax-awareness improves the effectiveness of favoured state and seed selection, it was unable to overcome the coverage decrease seen from basic state feedback. For targets where state feedback enabled configs reach the greatest coverage (Exim, BFTPd, OpenSSH, OpenSSL, Dcmtk, Kamailio, Forked-DAAPd) this research suggests adding syntax-awareness may allow them to reach even greater maximum coverage - as seen with BFTPd. This research also suggests syntax-awareness could amplify the code coverage impact of previously researched alternative AFLNet features (see 2.4.1), especially improved state selection approaches, which became positively impactful once syntax-aware.

5 Conclusion

Network applications are attractive targets for attackers, making the timely discovery of vulnerabilities important. One method to uncover potential vulnerabilities in applications is fuzzing. Protocol fuzzers are designed to send fuzz over TCP/IP sockets and manage the structured packets and stateful connections expected by network targets. AFLNet is the most common tool recently used as a baseline to extend and evaluate against. Research extending AFLNet has had difficulty finding important code coverage increases through alternative state feedback features. Our hypothesis was that syntactically invalid fuzz created by AFLNet hinders the effectiveness of state feedback features. From this, our research aim was to determine how adding syntax-awareness in AFLNet's mutation process changes the code coverage impact of state feedback features in AFLNet. We achieved this by measuring the change in the effectiveness of state feedback features after adding a syntax-aware mutation process relative to the current AFLNet process.

Our results demonstrate favoured seed and state selection, as state feedback features, become more impactful to code coverage achieved when a syntax-aware mutation process is used, seeing increases in their impact of +0.84% and +0.77% respectively. Basic state feedback meanwhile sees a reduction in effectiveness, possibly due to fuzz better engaging with the target program as found in slower target executions speeds. This supports our initial hypothesis that syntax-aware fuzz would better engage with target statefulness, with favoured state selection now becoming positively impactful for FTP targets, and favoured seed selection nearly doubling in impact. This also suggests existing works that have proposed alternative state feedback features in AFLNet, which resulted in modest coverage increases, could see their effectiveness improved, and that for targets which reach their greatest code coverage using state feedback could see even further positive effects.

We will list research contributions made by this work, followed by limitations and lastly, future work.

5.1 Contributions

The main contribution of this research is demonstrating that syntax-awareness, when added to the state-of-the-art protocol fuzzer, AFLNet, improves the effectiveness of two of its state feedback features while worsening the performance of basic state feedback. Reaching this contribution has allowed us to make several smaller contributions along the way:

- An ablation study of AFLNet, complimenting the recently published ablation study [15], through the addition of the OpenSSL target, consideration for region mutations and breaking state feedback down into three more granular features. The three state feedback features researched are basic state feedback, favoured seed selection and favoured state selection;
- A syntax-aware AFLNet variant named AFLNet-Packmute, complimenting the existing ChatAFL syntax-aware fuzzer, with a different design and greater syntax constraints to ensure validity throughout splicing, cloning and insert mutations;
- Demonstrated that the challenges posed by region mutations as currently implemented in AFLNet are partially overcome through the addition of syntax-awareness, making them positively impactful to fuzzing performance (for FTP targets);
- Demonstrated that comparing performance changes using a single fuzzer configuration can lead to incomplete conclusions. Using multiple configurations gives a better sense of under what conditions a certain change will improve fuzzing; and
- Demonstrated, through short verification experiments, that 37% of fuzz sent by AFLNet to FTP targets are rejected due to syntax or semantics and that adding syntax-awareness only reduces the rejection rate to just over 31%, most syntactically valid commands now being rejected due to semantics.

5.2 Limitations

The most significant limitation of this work was the decision to only implement syntax-awareness for FTP, which means the results observed may not apply to other protocols. This decision was made to constrain the time this research would take to complete. This was partially mitigated by using multiple FTP targets for validation, which we found behaved differently from one another.

The statistical significance of results was a limitation we did not initially anticipate. Several of the configurations had similar results, including several identical branch coverages, meaning the Mann-Whitney U-test did not result

in a p-value low enough to reject the null hypothesis that the two sets of samples came from the same distribution. We mitigated this by modifying the design to measure feature impact in several different ways. Each of these measurements had varying levels of statistical significance, though in general agreed with each other giving us additional confidence there was a performance difference.

Achieving the aim of this research required implementing syntax-awareness. During design and development we made decisions which introduced limitations. We conducted a limited study of the exploration-exploitation tradeoff in the context of syntax-awareness. This means there are almost certainly better ways to implement syntax-awareness in AFLNet. Our intuition is that further improved syntax-awareness would amplify our results (more positive or more negative feature impacts). If fuzz can even better interact with program state, fuzzing may become slower but the state-aware selection of seeds and states to fuzz may become more impactful. Another way syntax-awareness in AFLNet could be further improved is through changes to its region-mutations. Currently those mutations cause the mutation buffer to quickly exceed the maximum number of new message AFLNet will accept from a single fuzz, creating multi-region messages that cause syntactically invalid fuzz to be introduced.

The real goal of fuzzing research is to improve bug finding ability of fuzzers which is best done by considering bugs found (the ground truth). Due to time, we did not de-duplicate and investigate crashes. This was a limitation of the ablation study and of the validation. Instead, our aim was achieved through code coverage, a proxy for bug finding ability. Increased code coverage moderately agrees with increase in bug finding ability, though may not always agree [14]. Though we used 24 hour experiment durations which exceeds the recommended minimum 12 hours for best coverage to bug finding agreement, due to protocol fuzzing's much slower speed than binary fuzzing 24 hours may still not be enough. Of the FTP targets, our fuzzing experiments, on median, only found crashing test cases of BFTPd. A few runs of each PureFTPd and ProFTPd found crashing test cases.

5.3 Future Work

There is future work related to our research limitations as well as our contributions:

- Expand AFLNet-Packmute to support other protocols;

- Further investigate the exploitation-exploration tradeoff as it relates to applying syntax-awareness to AFLNet's mutation process. This could include using the power shedule to distribute fuzzing effort between exploitation and exploration similar to AFLGo [39];
- Evaluate and improve on AFLNet's region mutations. We believe they currently result in wasted stacked mutations due to the replace buffer mutation occuring too frequently and too large of seeds due to the duplicate buffer mutation. Using syntax-knowledge there is potential for new mutations as well;
- Add semantic knowledge to AFLNet. Even after adding syntax-awareness, over 30% of messages sent in the first 10 minutes of fuzzing LightFTP were rejected due to bad semantics ("Please login"). That represents a large portion of fuzzing effort and, if addressed, would make speed improvements already made to AFLNet be that much more effective;
- Automate the discovery of optimal AFLNet configurations for a given target. Our research found the best configuration for a target varies. In preliminary experiments we found that small details like timeouts can also be very impactful towards code coverage achieved. One method might involve many parallel fuzzers of different configurations, which recursively test different parameters to arrive at an approximate optimal. As each fuzzer is stopped and restarted, the existing corpus can be kept and even synced across to other fuzzers to continue progress; and
- Re-evalutate alternate state feedback features made by previous research using a syntax-aware AFLNet variant.

Bibliography

[1] M. Zalewski, "The afl++ fuzzing framework trophies." AFLplusplus. Accessed: 24-10-2024. [Online.] Available: `https://aflplus.plus/#trophies`.

[2] Google, "Honggfuzz: Security oriented software fuzzer." Github. Accessed: 24-10-2024. [Online.] Available: `https://github.com/google/honggfuzz?tab=readme-ov-file#trophies`, 2015.

[3] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, pp. 1–26, Nov. 2023.

[4] J. Ba, M. Bohme, Z. Mirzamomen, and A. Roychoudhury, "SGFuzz," in *Proceedings of the 31st USENIX Security Symposium*, (Boston MA USA), USENIX Security Symposium, Aug. 2022.

[5] V.-T. Pham, M. Bohme, and A. Roychoudhury, "AFLNET: A Greybox Fuzzer for Network Protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, (Porto, Portugal), pp. 460–465, IEEE, Oct. 2020.

[6] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large Language Model guided Protocol Fuzzing," in *Proceedings 2024 Network and Distributed System Security Symposium*, (San Diego, CA, USA), Internet Society, 2024.

[7] J. Robben and M. Vanhoef, "Netfuzzlib: Adding First-Class Fuzzing Support to Network Protocol Implementations," in *Computer Security – ESORICS 2024* (J. Garcia-Alfaro, R. Kozik, M. Choraś, and S. Katsikas, eds.), (Cham), pp. 65–84, Springer Nature Switzerland, 2024.

[8] A. Andronidis and C. Cadar, "SnapFuzz: high-throughput fuzzing of network applications," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, (Virtual South Korea), pp. 340–351, ACM, July 2022.

[9] J. Li, S. Li, G. Sun, T. Chen, and H. Yu, "SNPSFuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols Using Snapshots," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 2673–2687, 2022.

[10] R. Natella, "StateAFL: Greybox Fuzzing for Stateful Network Servers," *Empirical Software Engineering*, vol. 27, p. 191, Dec. 2022. arXiv:2110.06253 [cs].

[11] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. Rubinstein, " State Selection Algorithms and Their Impact on The Performance of Stateful Network Protocol Fuzzing ," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, (Los Alamitos, CA, USA), pp. 720–730, IEEE Computer Society, Mar. 2022.

[12] L. Yu, S. Yanlong, and Z. Ying, "Stateful protocol fuzzing with statemap-based reverse state selection," Aug. 2024. arXiv:2408.06844 [cs].

[13] 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*, (Virtual Denmark), pp. 662–665, ACM, July 2021.

[14] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, (Pittsburgh Pennsylvania), pp. 1621–1633, ACM, May 2022.

[15] R. Meng, V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet Five Years Later: On Coverage-Guided Protocol Fuzzing," *IEEE Transactions on Software Engineering*, vol. 51, pp. 960–974, Apr. 2025.

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

[17] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "SoK: Prudent Evaluation Practices for Fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*, (San Francisco, CA, USA), pp. 1974–1993, IEEE, May 2024.

[18] M. Zalewski, "American Fuzzy Lop." lcamtuf.coredump.cx. Accessed: 24-10-2024. [Online.] Available: `https://lcamtuf.coredump.cx/afl/technical_details.txt`.

[19] GNU, "Instrumentation Options (Using the GNU Compiler Collection (GCC))." gcc.gnu.org. Accessed: 25-10-2024. [Online.] Available: `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`.

[20] LLVM, "Clang Compiler User Manual; Clang 20.0.0." clang.llvm.org. Accessed: 25-10-2024. [Online.] Available: `https://clang.llvm.org/docs/UsersManual.html`.

[21] Gitlab, "protocol-fuzzer-ce." gitlab.com. Accessed: 25-10-2024. [Online.] Available: `https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce`.

[22] J. Pereyda, "Boofuzz: A fork and successor of the Sulley Fuzzing Framework." github.com. Accessed: 25-10-2024. [Online.] Available: `https://github.com/jtpereyda/boofuzz`.

[23] S. Jiang, Y. Zhang, J. Li, H. Yu, L. Luo, and G. Sun, "A Survey of Network Protocol Fuzzing: Model, Techniques and Directions," Feb. 2024. arXiv:2402.17394 [cs].

[24] Y. Yu, Z. Chen, S. Gan, and X. Wang, "SGPFuzzer: A State-Driven Smart Graybox Protocol Fuzzer for Network Protocol Implementations," *IEEE Access*, vol. 8, pp. 198668–198678, 2020.

[25] C. Song, B. Yu, X. Zhou, and Q. Yang, "SPFuzz: A Hierarchical Scheduling Framework for Stateful Network Protocol Fuzzing," *IEEE Access*, vol. 7, pp. 18490–18499, 2019.

[26] Z. Luo, J. Yu, F. Zuo, J. Liu, and Y. Jiang, "Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations," *USENIX Security Symposium*, Aug. 2023.

[27] V.-T. Pham, "AFLNwe." github.com. Accessed: 28-10-2024. [Online.] Available: `https://github.com/thuanpv/aflnwe`.

[28] R. Natella and V.-T. Pham, "ProFuzzBench - A Benchmark for Stateful Protocol Fuzzing." github.com. Accessed: 28-10-2024. [Online.] Available: `https://github.com/profuzzbench/profuzzbench/tree/master`.

[29] 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*, CCS '18, (New York, NY, USA), pp. 2123–2138, Association for Computing Machinery, Oct. 2018.

[30] GNU, "gcovr Frequently Asked Questions." gcovr.com. Accessed: 8-07-2025. [Online.] Available: `https://gcovr.com/en/stable/faq.html`.

[31] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. Publisher: Institute of Mathematical Statistics.

[32] A. Vargha and H. D. Delaney, "A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000. Publisher: [American Educational Research Association, Sage Publications, Inc., American Statistical Association].

[33] Y. Wei, B. Meyer, and M. Oriol, "Is Branch Coverage a Good Measure of Testing Effectiveness?," in *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures* (B. Meyer and M. Nordio, eds.), pp. 194–212, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[34] M. Böhme and B. Falk, "Fuzzing: on the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (Virtual Event USA), pp. 713–724, ACM, Nov. 2020.

[35] Seladb, "PcapPlusPlus: a multiplatform C++ library for capturing, parsing and crafting of network packets." github.com. Accessed: 18-11-2024. [Online.] Available: `https://github.com/seladb/PcapPlusPlus`.

[36] T. pandas development team, "pandas: Python data analysis library." pandas.pydata.org. Accessed: 15-06-2025. [Online.] Available: `https://pandas.pydata.org/`.

[37] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd international conference on Virtual execution environments*, (San Diego California USA), pp. 65–74, ACM, June 2007.

[38] O. Berger-Tal, J. Nathan, E. Meron, and D. Saltz, "The Exploration-Exploitation Dilemma: A Multidisciplinary Framework," *PLoS ONE*, vol. 9, p. e95693, Apr. 2014.

[39] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, (Dallas Texas USA), pp. 2329–2344, ACM, Oct. 2017.

Appendices

A Syntax-Awareness Coverage Value

This appendix investigates two things; what is the coverage value of adding syntax awareness to AFLNet and how AFLNet-Packmute compares to ChatAFL. ChatAFL's authors find syntax-aware mutations consistently increase coverage achieved over all targets tested; however this research makes a more nuanced conclusion.

A.1 Coverage value of syntax-awareness in AFLNet

By how much does adding syntax-awareness to AFLNet's mutation process increase the median branch coverage of a run? That is the question we consider here, limiting ourselves to FTP targets due to the limitation of AFLNet-Packmute. Across all configurations a fixed exploit rate of 75% is used. Table A.1 shows the percent coverage improvement for each of the four FTP targets between AFLNet (NET) and AFLNet-Packmute (PM) for each fuzzer configuration. Bolded coverage values show the largest median coverage value for that target. For three targets, the addition of syntax-awareness increases the largest median branch coverage achieved by any configuration; 2.5 branches for LightFTP (+0.70%), 1.5 branches for BFTPd (+0.31%), and 66 branches for PureFTPd (+4.78%). ProFTPd meanwhile saw a decrease of -0.43% after adding syntax-awareness. This decrease is not a result of fewer executions; in fact, AFLNet-Packmute had 0.29 executions per second more than AFLNet. It may be that ProFTPd has a different optimal exploit rate, benefitting more from exploration than LightFTP. Although we performed ten runs, there is the chance a few lucky runs in the AFLNet BASELINE resulted in a higher median value. In preliminary experimentation repeating experiments once resulted in two sets of 10 runs with feature impacts

doubling due to two very good runs. This is unlikely to be the case here though because ProFTPd sees a coverage decrease across many configurations.

If we limit ourselves to comparing a single fuzzer configuration we might make very different conclusions. STFL-FULL led to decreases in 3/4 targets once syntax-awareness was added, while STFL-FVR-SD-RGNS led to increases across all 4 targets. Averaged across all configurations, three targets saw a coverage increase, while ProFTPd saw a small decrease from adding syntax awareness.

Table A.1: AFLNet (NET) v AFLNet-Packmute (PM)

Config	NET	PM	%Δ	NET	PM	%Δ
	LightFTP			BFTPd		
BASELINE	**360**	**362.5**	0.69	484.5	481.0	-0.72
BASELINE-RGNS	347	354	2.02	481	485.0	0.83
STFL-RND	346.5	348.5	0.58	484.5	482.5	-0.41
STFL-RND-RGNS	346	350.5	1.30	476.5	484.0	1.57
STFL-FVR-SD	347.5	358	3.02	485	486.5	0.31
STFL-FVR-SD-RGNS	346	348.5	0.72	482	486.5	0.93
STFL-FVR-ST	343.5	345.5	0.58	483.5	**487.0**	0.72
STFL-FVR-ST-RGNS	344	348.5	1.31	476.5	482.5	1.26
STFL-FULL	348.5	358	2.73	**485.5**	482.5	-0.62
STFL-FULL-RGNS	344.5	354.5	2.90	485	481.5	-0.72
			Avg 1.59			Avg 0.32
	ProFTPd			PureFTPd		
BASELINE	**5347.5**	**5324.5**	-0.43	**1244.5**	1290.0	3.66
BASELINE-RGNS	5226.5	4788.0	-8.39	1214	**1304.0**	7.41
STFL-RND	5036	5297.5	5.19	1132.5	1075.5	-5.03
STFL-RND-RGNS	5042.5	4971.0	-1.42	989.5	1041.5	5.26
STFL-FVR-SD	5096	5094.5	-0.03	1150	1080.0	-6.09
STFL-FVR-SD-RGNS	4950.5	5088.0	2.78	1046	1108.5	5.98
STFL-FVR-ST	4944.5	4745.5	-4.02	1133	1101.5	-2.78
STFL-FVR-ST-RGNS	4939.5	5021.5	1.66	1038.5	1098.5	5.78
STFL-FULL	4976	4909.0	-1.35	1139.5	1121.5	-1.58
STFL-FULL-RGNS	4948.5	5115.0	3.36	1054	1130.5	7.26
			Avg -0.26			Avg 1.99

A.2 Comparison to ChatAFL

The main thesis document has discussed at length about the differences between this work and the syntax-aware AFLNet variant, ChatAFL [6]. The name of their fuzzer which only adds syntax-aware mutations to AFLNet is named ChatAFL-CL1: `https://github.com/ChatAFLndss/ChatAFL.git`. They report average branch coverage achieved by 10x24 hour experiments against two FTP targets using ProFuzzBench; ProFTPd and PureFTPd. Consulting their repository, they use a STFL-FULL configuration, possibly with a timeout of 5000+ ms as that is the default of their script. Though ChatAFL made changes to the likelihood and implementation of certain mutations, these are all found in the region mutations which STFL-FULL omits. ChatAFL uses target versions 3-4 years newer than included in ProFuzzBench which we expect will impact results slightly.

Table A.2 shows the branch coverage results for each fuzzer against the two aforementioned FTP targets. The baseline for each is the coverage achieved by AFLNet. For AFLNet-Packmute we include both the STFL-FULL and STFL-FULL-RGNS configurations. Comparing STFL-FULL, AFLNet-Packmute sees a reduction in coverage, while ChatAFL-CL1 sees important coverage increases of +3.63% for ProFTPd and +6.67% for PureFTPd. These coverage increases are aligned with our results for the STFL-FULL-RNGS configuration, achieving +3.36% coverage increase for ProFTPd and +7.26% coverage increase for PureFTPd.

Table A.2: AFLNet-Packmute v ChatAFL

Target	AFLNet-Packmute (PM)			
	AFLNet	PM	%Δ	Speed-up
	STFL-FULL			
ProFTPd	4976.0	4909.0	-1.35	-
PureFTPd	1139.5	1121.5	-1.58	-
	STFL-FULL-RGNS			
ProFTPd	4948.5	5115.0	+3.36	3.57×
PureFTPd	1054.0	1130.5	+7.26	2.71×
Target	ChatAFL-CL1 (CHAT) STFL-FULL			
	AFLNet	CHAT	%Δ	Speed-up
ProFTPd	4763.00	4935.90	+3.63	2.45×
PureFTPd	1056.30	1126.76	+6.67	1.34×

These results show the differences in design decisions do effect coverage results. ChatAFL used a fixed exploit rate of 50% with the decision placed prior to each havoc stage, while AFLNet-Packmute makes a single exploit decision once per seed with an exploit rate of 75%. AFLNet-Packmute also modifies the splicing stage to be syntax-aware as well. This provides further support that perhaps ProFTPd benefits from more exploration, as mentioned in Section A.1 where we noted syntax-awareness on average reduced code coverage achieved for that target. Perhaps ChatAFL-CL1 could see even greater coverage increases using a STFL-FULL-RGNS configuration.

B Region Mutations

This appendix investigates the code coverage impact of region mutations and how syntax-awareness changes that impact. Though not a state feedback feature, the data is on hand from the ablation study and no previous research has looked at the contribution towards code coverage in AFLNet.

Region mutations make a lot of sense, being the only mechanism to rearrange and modify the message sequence overall. However, we believe as implemented they are partially counterproductive and lead to too large of seeds.

- Replace buffer (mutation case 17): This mutation replaces the entirety of the current mutation buffer with a random region from a random seed. On average this mutation is applied 3 times per fuzz created; 1/21 chance of occurring where there is on average 64 stacked mutations applied. This mutation essentially resets the stacked mutations back to zero using a new buffer. Stacked mutations have been wasted.
- Duplicate buffer (mutation case 21): This mutation doubles the entire mutation buffer, which can lead to massive fuzz sizes in terms of number of messages if applied multiple times to a single fuzz.

Relevant future work would be to re-design region mutations. Perhaps having a separate stacked mutation loop for region mutations, selecting them less frequently, or for only some proportion of total havoc stages.

RQ A - What is the code coverage impact of region mutations? Five fuzzer configurations per target added region mutations in the ablation study. These are shown in Figure 3.1 as the boxes titled "+RGNS". In total there are 65 experiments (13 targets times 5 region configs). Across all configurations, enabling regions mutations had an average impact of -0.001%, however with a standard deviation of 4.18, and 25 experiments together seeing a mean speed-up of 3.20×, there was significant variation. Table B.1 shows the mean percent change in branch coverage from adding region mutations (impact) along with the standard deviation and mean coverage speed-up fac-

tor. We see that region mutations can lead to important speed-up factors when they do help, though it is for a limited number of targets. When region mutations are added to the STFL-FVR-ST configuration, we see a positive mean impact, however this is skewed by two large positive results. Two targets consistently benefitted in a statistically significant way from region mutations; Kamailio and Forked-DAAPd. One target consistently dropped in performance in a statistically significant way when region mutations were enabled; PureFTPd.

Recall region mutations introduce new messages into the fuzz or replace the message being mutated. A positive consequence is a greater number of message orders and combinations are attempted. The traditional AFL mutations would not neatly add messages from other seeds. A negative result is much larger seeds, greatly increasing mutation space. AFL was designed on the premise of small changes to small seeds, which progressively reach deeper application logic. If seeds keep getting larger, it becomes more unlikely the specific bytes that we need to mutate in a specific way to get deeper will be found.

Table B.1: Code coverage impact of region mutations

Config+Regions	Impact		Speed-up (# targets)
	Mean	σ	
BASELINE	-0.18	2.34	2.83 (3)
STFL-RND	-0.17	4.58	2.59 (7)
STFL-FVR-SD	-0.06	3.84	3.74 (4)
STFL-FVR-ST	0.59	4.47	3.99 (6)
STFL-FULL	-0.18	5.62	2.91 (5)

RQ B - How has adding syntax-awareness changed the code coverage impact of region mutations? This question can only be considered for the 4 FTP targets, a limitation of AFLNet-Packmute. We expect syntax-awareness to improve the code coverage impact of region mutations, or at least lessen the coverage reduction it causes. Why? Syntax-awareness reduces the mutation space available to the fuzzer, mitigating some of the big-seed problem discussed in RQ A. These results use a 75% exploit rate as was determined optimal for a BASELINE configuration against the LightFTP target. Table B.2 shows the mean impact of adding region awareness to the listed configurations for each AFLNet and AFLNet-packmute. Lastly, it shows the change in impact. For FTP targets, the addition of region mutations to AFLNet configurations, on average, reduces the median code coverage

achieved with an overall average of -2.77% coverage change. For AFLNet-Packmute however, region mutations lead to positive coverage impact when added to STFL-RND, STFL-FVR-SD, STF-FVR-ST and STFL-FULL configurations with an overall average of +0.74%. Adding syntax-awareness increased the coverage impact of region mutations for every configuration, from +2.03% up to +4.60% median coverage improvement.

Table B.2: Change in code coverage impact of region mutations from adding syntax-awareness (FTP Targets only)

Config+ Regions	AFLNet			AFLNet-Packmute			Impact Δ
	Impact		Speed-up (# targets)	Impact		Speed-up (# targets)	
	Mean	σ		Mean	σ		
BASELINE	-2.26	1.19	-	-0.23	1.57	1.85 (2)	+2.03
STFL-RND	-3.57	6.09	1.08 (1)	1.03	3.96	3.16 (3)	+4.60
STFL-FVR-SD	-3.24	4.02	-	0.58	2.46	2.32 (2)	+3.82
STFL-FVR-ST	-2.43	4.00	1.23 (1)	1.38	3.05	2.74 (2)	+3.81
STFL-FULL	-2.33	3.48	-	0.95	2.28	2.81 (2)	+3.28