ADAPTIVE MUTATION OPERATOR SCHEDULING IN PROTOCOL FUZZING

A region mutation-focused approach using bandit algorithms

SÉLECTION ADAPTATIVE DES OPÉRATEURS DE MUTATION DANS LE FUZZING DE PROTOCOLE

Une approache centrée sur les opérateurs de mutation de région utilisant les algorithmes de bandit

A Thesis Submitted to the Royal Military College of Canada
by

Patricia Watkins, B.Eng, rmc
Captain

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

May 2026

© This project may be used within the Department of National Defence but copyright for open publication remains the property of the author.

Abstract

Fuzzing has emerged as a critical technique in software testing for identifying bugs and vulnerabilities. It works by generating test inputs, often using randomness to produce unusual cases, which are then executed against the target software to attempt to expose abnormal program behaviour. Protocol fuzzing is a specialized form which focuses on testing network protocol implementations. It presents an interesting challenge in that the fuzzer must manage exchanges of messages that influence the program's state. AFLNET is a popular protocol fuzzer which adapted greybox mutational fuzzing for protocol targets. In addition to standard byte mutations, AFLNET introduced region mutations which apply modifications to the message sequence rather than the message content. No prior work has analyzed how AFLNET applies its two mutation types, leaving potential inefficiencies that may limit the code coverage achieved. The current random-based approach may squander valuable execution time on ineffective mutations, as the effectiveness of mutation types and operators varies across targets. This research seeks to improve its coverage performance to more thoroughly test protocol implementations for security vulnerabilities and unintended behaviour.

The aim of this research is to evaluate the impact of a region mutation-focused target-adaptive mutation strategy on code coverage in AFLNET. This approach utilizes the multi-armed bandit algorithms, Thompson Sampling and Linear Thompson Sampling, to dynamically adjust the selection of mutation types and operators during the fuzzing process. This design, known as Target-Adaptive Mutation Strategy (TAMS), can learn which mutations are most effective for a given target using code coverage feedback.

Validation experiments were conducted across five diverse protocol targets: LightFTP (FTP), DNSmasq (DNS), Exim (SMTP), OpenSSH (SSH), and Kamailio (SIP). The performance of the TAMS design was compared to AFLNET with its original mutation strategy using median branch coverage results. The results showed that AFLNET-TAMS achieved improved coverage on OpenSSH with the highest performing configuration seeing a 0.641%

increase. The four other targets resulted in comparable coverage to baseline, with Kamailio experiencing lower coverage in some configurations, particularly when the *delete* region mutation operator is included. Additionally, a 52.2% reduction in average seed length was observed across targets in TAMS implementations.

Résumé

Le fuzzing est une technique essentielle dans les tests logiciels pour identifier les bogues et les vulnérabilités. Il consiste à générer des entrées, souvent en recourant au hasard pour produire des cas inhabituels, qui sont ensuite exécutées sur le logiciel cible afin de détecter des comportements anormaux. Le fuzzing de protocole est une forme spécialisée qui se concentre sur le test des implémentations de protocoles réseau. Il présente un défi intéressant dans la mesure où le fuzzer doit gérer les échanges de messages qui influencent l'état du programme. AFLNET est un fuzzer de protocole populaire qui a adapté le fuzzing mutationnel greybox pour les cibles de protocole. En plus des mutations d'octets standard, AFLNET a introduit des mutations de région qui appliquent des modifications à la séquence de messages. Aucun travail antérieur n'a analysé la manière dont AFLNET applique ses deux types de mutation, laissant des inefficacités potentielles qui pourraient limiter la couverture de code atteinte. L'approche actuelle basée sur le hasard risque de gaspiller un temps d'exécution précieux sur des mutations inefficaces, car l'efficacité des types de mutation et des opérateurs varie selon les cibles. Ce travail cherche à améliorer la couverture de code de AFLNET pour tester plus en profondeur les implémentations de protocoles afin de découvrir des failles de sécurité et des comportements indésirables.

L'objectif de cette recherche est d'évaluer l'impact d'une stratégie de mutation adaptative ciblée, axée sur la mutation de région, sur la couverture de code dans AFLNET. Cette approche utilise les algorithmes de bandit à plusieurs bras, l'échantillonnage de Thompson et l'échantillonnage linéaire de Thompson, pour ajuster de manière dynamique la sélection des types de mutations et des opérateurs au cours du fuzzing. Cette conception, connue sous le nom de stratégie de mutation adaptative à la cible (TAMS), permet d'apprendre quelles mutations sont les plus efficaces pour une cible donnée en utilisant le retour d'information sur la couverture de code.

Des expériences de validation ont été menées sur cinq cibles de protocole différentes: LightFTP (FTP), DNSmasq (DNS), Exim (SMTP), OpenSSH

(SSH) et Kamailio (SIP). Les performances de l'approche TAMS ont été comparées à celles de AFLnet avec sa stratégie de mutation d'origine, en utilisant les résultats de couverture de branche médiane. Les résultats ont montré que le fuzzer AFLnet-TAMS a atteint une couverture améliorée sur OpenSSH, la meilleure configuration atteignant une augmentation de 0,641 %. Les quatre autres cibles ont fourni une couverture comparable à celle de la référence, Kamailio présentant une couverture inférieure dans certaines configurations, en particulier lorsque l'opérateur de mutation de région *delete* est inclus. De plus, une réduction de 52,2 % de la longueur moyenne des entrées de base a été observée sur l'ensemble des cibles dans les implémentations TAMS.

Contents

Abstract **ii**

Résumé **iv**

List of Tables **ix**

List of Figures **xi**

List of Acronyms **xii**

1 Introduction **1**
1.1 Motivation 2
1.2 Statement of Deficiency 2
1.3 Aim 3
1.4 Research Activities 4
1.5 Results 4
1.6 Organization 5

2 Background **6**
2.1 Fuzzing 6
2.1.1 Code Coverage 7
2.1.2 Fuzzer Evaluation 8
2.1.3 American Fuzzy Lop (AFL) 8
2.2 Protocol Fuzzing 12
2.2.1 AFLNET 12
2.2.2 *ProFuzzBench* 14
2.3 Multi-Armed Bandit 14
2.3.1 Thompson Sampling 15
2.3.2 Linear Thompson Sampling 16
2.4 Related Works 17

2.4.1 Mutation Operator Scheduling in Binary Fuzzing 18
2.4.2 Protocol Fuzzing Mutation Strategies and Other Related Research 19

3 Methodology and Design 21
3.1 Preliminary Mutation Experiments 22
3.1.1 Mutation Operator Performance 23
3.1.2 Probability of Region Mutation 23
3.1.3 Stack Size for Region Mutation 23
3.1.4 Addition of *Delete* Operator 24
3.2 Development of AFLNET-TAMS 24
3.2.1 Mutation Type Selection 25
3.2.2 Region Mutation Operator Selection 27
3.2.3 Byte Mutation Operator Selection 28
3.3 Verification 28
3.4 Validation 29
3.5 Summary 30

4 Results 32
4.1 Experimental Design 32
4.2 Preliminary Experiments on AFLNET 33
4.2.1 Development of *Delete* Operator 33
4.2.2 Mutation Operator Performance 36
4.2.3 Probability of Region Mutation 38
4.2.4 Stack Size for Region Mutation 39
4.2.5 Addition of *Delete* Operator 40
4.3 AFLNET-TAMS 40
4.3.1 Development 40
Random Sampling 40
Thompson Sampling Mutation Type Selector 41
Linear Thompson Sampling Region Mutation Selector . 42
4.3.2 Tuning Experiments 43
Reward Signal: “Interesting” Test Case Types 44
Exploration Parameter for Linear Thompson Sampling . 47
Stack Size for Region Mutations 48
4.4 Verification 49
4.4.1 Thompson Sampling Mutation Type Selector 49
4.4.2 Linear Thompson Sampling Operator Selector 50
4.4.3 *Delete* Operator 51
4.5 Validation 51

4.5.1 TAMS Type Selector 54
4.5.2 TAMS Region Operator Selector 54
4.5.3 TAMS Combined Design 54
4.6 Discussion 55
4.6.1 Mutation Type Selector 55
4.6.2 Region Operator Selector 57
4.6.3 TAMS Combined Design 59
4.6.4 Impact of TAMS on Seed Length and Execution Speed 61
4.7 Summary 64

5 Conclusion 67
5.1 Contributions 67
5.2 Limitations 68
5.3 Future Work 69

List of References 71

Appendices 77

A Success Rates for Mutation Operator Performance Preliminary Experiments A-1

B Linear Thompson Sampling (LinTS) Region Operator Selection Based on Seed Length B-1

List of Tables

2.1 Mutation Operators Used in AFL Havoc Stage. 11

3.1 Preliminary Experiment Design. 22
3.2 Comparison of Mutation Strategies between AFLnet and AFLnet-TAMS. 26
3.3 Comparison of Selected Protocol Targets. 29
3.4 Validation Test Configurations. 30

4.1 Branch Coverage Results for Preliminary Experiments. 34
4.2 TAMS-Type Branch Coverage Results using both "Interesting" Types as Reward. 44
4.3 Average Percentage of "Interesting" Test Cases Types across Targets. 45
4.4 Median Coverage Results for Targets using Different v Values in TAMS-RegOp. 47
4.5 Median Coverage Results for Targets across Different Stack Sizes in TAMS Configurations. 48
4.6 Branch Coverage Results for TAMS Validation on the LightFTP Target. 52
4.7 Branch Coverage Results for TAMS Validation on the DNSmasq Target. 52
4.8 Branch Coverage Results for TAMS Validation on the Exim Target. 53
4.9 Branch Coverage Results for TAMS Validation on the OpenSSH Target. 53
4.10 Branch Coverage Results for TAMS Validation on the Kamailio Target. 53
4.11 Mutation Type Statistics for TAMS-Type (*delete* disabled). 55
4.12 Average Region Mutation Operator Usage Percentages for TAMS-RegOp (*delete* enabled). 58
4.13 Average Region Mutation Operators Success Rates for TAMS-RegOp (*delete* enabled). 58

4.14 Mutation Type Statistics for TAMS-Combined (*delete* disabled). . 60
4.15 Average Region Mutation Operator Usage Percentages for TAMS-Comb (*delete* enabled). 60
4.16 Average Region Mutation Operator Success Rates for TAMS-Comb (*delete* enabled). 61
4.17 Average Seed Length (bytes), Average Execution Speed, and Median Branch Coverage across Targets for Baseline and TAMS Configurations. 62
4.18 TAMS Percent Change in Average Seed Length Relative to Baseline. 63

A.1 Mutation Operator Success Rate Statistics for LightFTP. A-2
A.2 Mutation Operator Success Rate Statistics for DNSmasq. A-3
A.3 Mutation Operator Success Rate Statistics for Exim. A-4
A.4 Mutation Operator Success Rate Statistics for OpenSSH. A-5
A.5 Mutation Operator Success Rate Statistics for Kamailio. A-6

List of Figures

2.1 Code Block Edges Example. 7
2.2 AFL Evolutionary Loop. 9
2.3 Stacked Mutation Example. 10

3.1 TAMS Process 25

4.1 Number of "Interesting" Test Cases Discovered over Time. 38
4.2 Success Rates of Each Mutation Operator at Triggering New Hit Counts (normalized to the targets). 45
4.3 Success Rates of Each Mutation Operator at Triggering New Edges (normalized to the targets). 46

B.1 TAMS-RegOp Region Operator Selection across Seed Lengths for LightFTP. B-1
B.2 TAMS-RegOp Region Operator Selection across Seed Lengths for DNSmasq. B-2
B.3 TAMS-RegOp Region Operator Selection across Seed Lengths for Exim. B-2
B.4 TAMS-RegOp Region Operator Selection across Seed Lengths for OpenSSH. B-3
B.5 TAMS-RegOp Region Operator Selection across Seed Lengths for Kamailio. B-3

List of Acronyms

AFL	American Fuzzy Lop
CDF	Cumulative Distribution Function
CSV	Comma-Separated Value
DAAP	Digital Audio Access Protocol
DNS	Domain Name Service
Exp3	Exponential-Weight Algorithm for Exploration and Exploitation
FA	Firefly Algorithm
FTP	File Transfer Protocol
GSL	GNU Scientific Library
IID	Independent and Identically Distributed
IPSM	Implemented Protocol State Machine
LinTS	Linear Thompson Sampling
LinUCB	Linear Upper Confidence Bound
MAB	Multi-Armed Bandit
PIT	Probability Integral Transform
PSO	Particle Swarm Optimization
RNG	Random Number Generator
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
SSH	Secure Shell
TS	Thompson Sampling
UCB	Upper Confidence Bound
VM	Virtual Machine

1 Introduction

Network protocols and their software implementations form the foundation for computer networks and the Internet, enabling communication between machines. This interconnected nature of protocol communications makes them valuable targets for adversaries attempting to gain unauthorized remote access to machines or networks. With the widespread use of network protocols, including in organizations such as financial and government, vulnerabilities can be far-reaching and have devastating impacts. Therefore, it is prudent to thoroughly test their software implementations to identify vulnerabilities for remediation to ensure their security and robustness.

Fuzz testing, or fuzzing, has emerged as an essential technique in software testing, being widely used and researched by companies such as Google and Microsoft [1–5]. Binary fuzzers, such as American Fuzzy Lop (AFL) [1], are very effective for programs that expect a single input but are not suitable for fuzzing protocols that expect sequences of messages as input. Protocols are stateful, meaning a server will react differently depending on what messages it has already received. In the case of File Transfer Protocol (FTP), the server restricts interactions, such as file system access, until the client completes a login. To work within these restrictions, a protocol fuzzer must effectively manage message sequencing and tracking of the target state. Fuzzing network protocols therefore presents added complexity compared to many other fuzzing targets such as program binaries. To address these challenges, stateful protocol fuzzers were developed to work with message sequences and protocol states.

AFLNET [6] introduced stateful, coverage-guided, mutational, protocol fuzzing. This fuzzer builds a state machine model of the target during the fuzzing process and tracks which message sequences lead to which states. This enables AFLNET to effectively fuzz network protocols and explore different server states. In addition to the standard byte manipulation mutations, AFLNET also applies mutations to the message sequence. These sequence-level mutations, known as region mutations, modify the structure of the message sequence, while the byte mutations modify the message content. Since

its creation in 2020, AFLNET has been further researched and built upon, with many extensions focusing on improving its state modelling, throughput, or seed selection [7].

1.1 Motivation

In the modern world, many critical services depend on network systems, such as finance, healthcare, and government. Network protocol implementations are large and complex software programs which often interact with external untrusted machines. It follows that protocol servers are at a high risk of cyber attack and a successful exploit could have severe ramifications including compromise of confidential information or loss of essential services.

Fuzzing has become an important tool in software testing. Its automated nature allows thousands of inputs to be executed against a program with limited human interaction, and enables the discovery of unexpected edge cases that manual testing might overlook. This scalability is important for complex and large protocol implementations. There have been advances in binary fuzzing, particularly with the use of lightweight program feedback. However, protocol fuzzing as a subcategory has received less attention compared to binary fuzzing, presenting opportunities for further improvement.

The motivation behind this research is to improve the effectiveness of stateful protocol fuzzing to strengthen the security of network protocol implementations. By enhancing the mutation process, protocol fuzzing has the potential to more effectively explore execution paths, increasing the likelihood of identifying bugs and vulnerabilities for remediation.

1.2 Statement of Deficiency

Despite many researchers' improvements to AFLNET, its mutation strategy remains underexplored. Existing research on greybox protocol fuzzing mutations has focused on format-aware approaches [8–11], gradient-guided byte selection [12], and an adaptive scheduling strategy applied indiscriminately to all mutation operators [13]. However to our knowledge, no work has attempted to improve AFLNET's unique region mutations which is speculated to contribute to undesirable behaviour where the message sequence can grow excessively in latter fuzzing stages and result in slower execution speeds [14, 15]. We hypothesize that the random method that AFLNET applies its region mutations contributes to this issue. More specifically, its duplication and insertion operations likely play a part in this problem.

AFLNET's mutation strategy randomly selects from 20 mutation operators with static weights. AFLNET does not differentiate between region and byte mutations when choosing an operator, even though these two categories have different purposes and effects. While region mutations can be important for exploring new state transitions, their overuse might hinder the mutation process by slowing down fuzzing with long sequences that seldom increase coverage. This observation forms the central hypothesis for our research: that region mutations are not being applied effectively in AFLNET's default mutation strategy, hindering the overall fuzzing process.

Recent research [9] has shown that many protocol targets achieve better fuzzing results with region mutations disabled, compared to their performance with this mutation type enabled. Based on these results, the fuzzing of simpler protocols, such as FTP, may benefit from a reduced number of region mutations, while the fuzzing of more complex stateful protocols, such as Session Initiation Protocol (SIP) and Digital Audio Access Protocol (DAAP), may perform better with more region mutations. This is reasonable given that different protocol implementations have varying sizes and complexity of their state space.

A more effective mutation strategy should adapt based on target feedback, achieving a balance of modifying the messages in the sequence and the content within them. Such a strategy should consider the total message sequence length and apply region duplication or insertion operations more sparingly when sequence length is already long.

Due to the lack of adaptability in its application of region mutations, AFLNET may fail to cover code that a target-adaptive mutation strategy would reach within the same timeframe. Untested code regions leave any contained bugs undiscovered, allowing potential vulnerabilities to persist. This research seeks to address this limitation by improving AFLNET's mutation strategy.

1.3 Aim

The aim of this research is to evaluate the impact of a region mutation-focused target-adaptive mutation strategy on code coverage in AFLNET. The mutation strategy developed first selects between performing a region mutation or a byte mutation based on their past code coverage performance. Next, if region mutation is selected, the associated mutation operators are selected based on the current sequence length as well as their historical code coverage performance.

Evaluation is performed by comparing code coverage metrics using an unmodified AFLNET against a version of AFLNET with the target-adaptive mutation strategy implemented. To assess how the new mutation strategy performs across different types of protocols, five diverse targets were used in experimentation: LightFTP, DNSmasq, Exim, OpenSSH, and Kamailio.

1.4 Research Activities

This research was conducted in four phases:

1. **Preliminary assessment of mutation operator performance.** The preliminary experiments investigated factors influencing mutation effectiveness to guide the final design of the new mutation strategy. These experiments analyzed individual operator performance, different values for region mutation stack size, different proportions of region mutations, and the impact of adding a new region mutation operator, *delete*.
2. **Develop AFLNET-Target-Adaptive Mutation Strategy (TAMS).** A variant of AFLNET, AFLNET-TAMS was developed which integrates a target-adaptive mutation strategy where the selection of the mutation operators is adapted based on code coverage feedback and sequence length as applicable.
3. **Verification.** Analysis was performed to confirm that AFLNET-TAMS acted as expected with mutation operator selection that adapted to target feedback according to the selected algorithms.
4. **Validation.** The change in code coverage between AFLNET-TAMS and AFLNET with its default mutation strategy was evaluated. Five protocol targets were included in the experiments which were conducted using a protocol fuzzing benchmark platform called *ProFuzzBench*.

1.5 Results

Our results reveal that the new mutation strategy improves median branch coverage for the OpenSSH target. The remaining targets achieved coverage comparable to the baseline, with Kamailio decreasing in coverage in some configurations. OpenSSH showed consistent performance differences among its region mutation operators, which likely contributed to the improvements observed with TAMS. Parameter tuning was found to be very target specific, suggesting that further target-adaptive behaviour could result in improvement across different protocol implementations.

1.6 Organization

Chapter 2 delves into the background information related to protocol fuzzing and mutation strategies which includes AFLNET's current mutation strategy, adaptive mutation scheduling in binary fuzzing, and Multi-Armed Bandit (MAB) algorithms which this research uses for mutation scheduling. Chapter 3 explains the details of the research including the design of the experiments and particulars of the new mutation strategy. Chapter 4 presents the experimental results and validates our aim. Chapter 5 summarizes our research contributions, limitations, and potential future work.

2 Background

This chapter provides an overview of previous research relating to our work. It begins with an introduction to fuzzing more broadly, before moving on to greybox mutational protocol fuzzing specifically. The next section shifts to MAB algorithms with particular emphasis on Thompson Sampling (TS) and Linear Thompson Sampling (LinTS), our selected methods for mutation scheduling. Finally, there is a review of related research on mutation operator scheduling and protocol fuzzing.

2.1 Fuzzing

Fuzzing is an automated software testing technique that generates and executes large numbers of test inputs to uncover bugs and vulnerabilities. New test inputs may be created by mutating existing inputs or generated using structural specifications. The first technique is known as mutation-based, or mutational fuzzing, while the second is classified as generation-based fuzzing. Mutation operations are typically random, aiming to produce unexpected inputs that might reveal abnormal program behaviour, which could expose vulnerabilities. In mutation-based fuzzing, the fuzzer has a list of normal existing inputs from which it pulls to then apply mutations to create new inputs. These existing inputs are known as seeds and a collection of them is referred to as the seed corpus.

Fuzzers are also categorized by their level of access to the target program. Blackbox fuzzers do not have any access to the internals of the target program, meaning they interact with the target's input and output only. On the other hand, whitebox fuzzers have full access to the target program and source code which allows them to create inputs specially designed to solve constraints to explore harder to reach areas of code. Blackbox fuzzers are very fast, but their random mutation style can suffer if the target program requires specific inputs like checksums to reach various regions. Whereas, whitebox fuzzers

can produce inputs which solve specific paths, however, they are a lot slower than blackbox fuzzing and can suffer from path explosion with large complex programs.

Greybox fuzzing lies somewhere between the whitebox and blackbox fuzzing categories. During fuzzing, it has some access to the internals of the target program. This usually takes the form of program instrumentation to receive runtime feedback from the target which allows this fuzzer type to strike a balance of speed and effectiveness. Using this method, they are able to improve test case quality by monitoring which ones trigger more code execution in the target, and adjusting fuzzing accordingly. Greybox feedback-guided mutational fuzzers have become very popular largely due to the success of AFL [1].

2.1.1 Code Coverage

Code coverage is a key metric in fuzzing used to approximate how thoroughly software has been tested. By tracking which code has been executed during fuzzing, we gain an estimate of which areas of the program have been ran and which remain untouched. Coverage can also be used to guide the fuzzer during the fuzzing process. For example, AFL [1] coarsely tracks edge coverage through program instrumentation which helps it select which seeds to mutate.

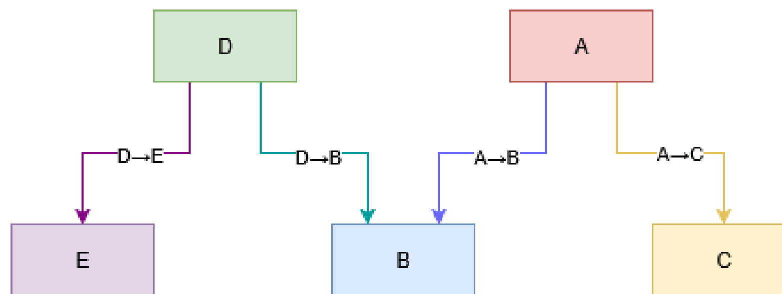

Figure 2.1: Code Block Edges Example.

Several types of code coverage exist. Line coverage measures the number of executed lines of code. Branch coverage measures whether each possible outcome has been executed for branching conditions in the code, such as the true and false paths of "if" statements and the different cases of switch statements. Edge coverage measures the transitions between blocks of code [1]. For example as seen in Figure 2.1, if block A leads to either B or C, and block D leads to B or E, the unique edges are A→B, A→C, D→B, and D→E.

This approach captures the order of execution rather than simply whether individual code blocks were executed, revealing additional program behaviour.

2.1.2 Fuzzer Evaluation

Based on Klees et al. [16], there are several best practices for evaluating fuzz testing to avoid misleading conclusions. Statistical tests, such as the Mann-Whitney U test [17], should be applied to determine the statistical significance of results. Performance can vary greatly depending on initial seeds, therefore, researchers should test a variety of seeds and clearly explain what initial seeds were used. Fuzzing trials must be sufficiently long (at least 24-hours) to get a more accurate estimate of fuzzer performance since it fluctuates over time. Evaluations should account for the inherent randomness in fuzzing by running multiple trials. Schloegel et al [18] recommend at least 10 trials be conducted. Using a sufficient number of trials is more likely to lead to a statistically significant Mann-Whitney U result since the test has more information to use. This is particularly important when changes are very small, more trials will be needed to show the improvement is consistent.

The Mann-Whitney U test helps determine whether two groups of samples come from the same underlying distribution. The p-value produced by the test represents the percentage of samples drawn from the same distribution that would show a difference this large or larger. A commonly chosen threshold is 0.05, where if the p-value is less than this value, the observed difference would be very unlikely ($\leq$5%), so the hypothesis that the groups come from the same distribution is rejected, and the groups are considered to be statistically different.

It is worth noting that protocol fuzzing is generally slower than binary fuzzing due to the overhead of the network stack, context switching, and server initialization and termination [19]. As a result, longer fuzzing durations may lead to a more stable assessment of coverage performance for protocol targets. However, a 24-hour fuzzing duration remains common practice in protocol fuzzing evaluations and comparisons [8, 15, 20–22].

2.1.3 AFL

Created in 2013, AFL [1] has become one of the most influential fuzzers. It is a greybox mutational coverage-guided fuzzer designed with simplicity and effectiveness in mind [1]. Since its release, many researchers have extended AFL's framework with new techniques.

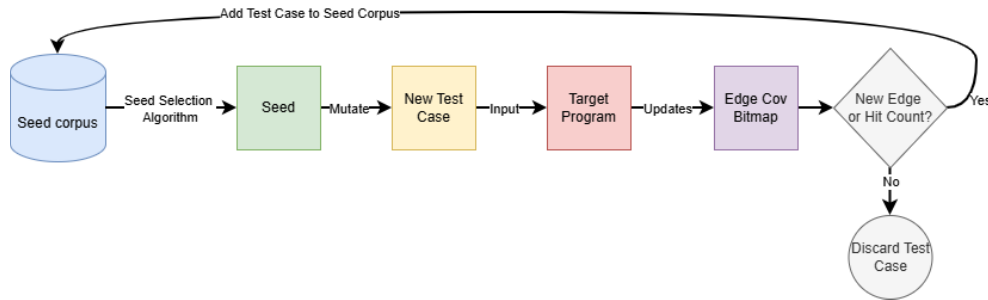

Figure 2.2: AFL Evolutionary Loop.

At the core of AFL is its fuzzing loop where seeds are selected from the corpus, mutated to create new test inputs, executed against the target, and any test case that triggers previously unexplored code or new program behaviour is added back into the corpus for further fuzzing as shown in Figure 2.2.

AFL uses a bitmap to approximately track edge coverage during fuzzing. It was designed to be efficient and to keep computational cost low. The bitmap is limited to 64kB where each byte stores a hit count for a computed edge value. The value for a particular edge is calculated using random identifiers inserted in basic blocks during compile time. Pseudocode for the computation and incrementation of the bitmap location is shown in Listing 2.1.

There may exist collisions of these computed edge identifiers where different edges in the code share the same edge identifier. Since the priority is a fast edge tracking method, this is an acceptable trade-off over using an expensive hash function and larger hash table. The small bitmap can fit inside the L2 cache, and the shift and XOR operations are inexpensive, making this method quick and lightweight.

Listing 2.1: AFL bitmap edge tracking pseudocode. [1]

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

The test cases which are chosen to be added to the seed corpus are referred to as “interesting”. There are two types of “interesting” test cases: new edge coverage and new hit count. New edge coverage test cases cause an edge tuple to be executed for the first time in the fuzzing session. New hit count test cases do not trigger a new edge execution but rather result in an existing edge being executed a different number of times than has been

observed previously. This captures variations in program behaviour, such as loops executing a different number of times, and it introduces redundancy since there may be edge collisions on the bitmap. To achieve this without triggering an "interesting" test case on every execution count, AFL groups hit counts into buckets (1, 2, 3, 4–7, 8–15, 16–31, 32–127, 128+), and only considers a test case an "interesting" new hit count if it falls into a new bucket.

An "energy" score is calculated for the selected seed to determine how many times AFL will mutate and test it. The energy score is based on a seed's edge coverage, execution speed, when it was added to the corpus, and how many mutations generations it is from its starting seed. After the energy score has been exhausted, a different seed is chosen from the corpus and the cycle continues.

AFL's mutation strategy consists of three stages: deterministic, havoc, and splicing [23]. The deterministic stage applies a predefined set of mutation operators, after which is the havoc stage that applies multiple operators chosen uniformly at random. The mutation operators used by AFL in its main havoc stage can be seen in Table 2.1 (most descriptions from Wu et al [24]). This application of multiple individual mutation operators is known as stacking. An example of a stacked mutation is shown in Figure 2.3 where four mutation operators are selected for a given mutation. In the havoc stage, AFL randomly selects powers of two for its stack size starting at 2, with a maximum of 128 stacked mutations. AFL spends most of its time in this stage, as randomness is effective at triggering new behaviours [25]. If no coverage progress is made for a given seed after the havoc stage, AFL enters the splicing stage, where a crossover operator combines two seeds into a new test case before applying havoc mutations to this hybrid seed [25].

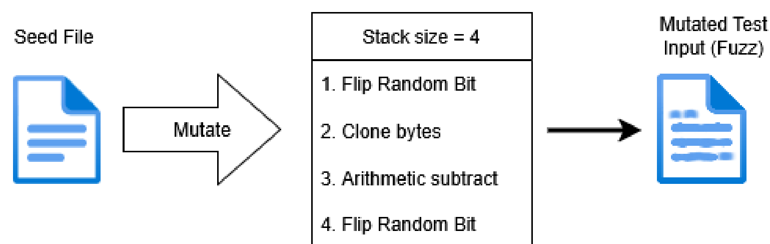

Figure 2.3: Stacked Mutation Example.

To adapt AFL's fuzzing to a target application, AFL can be given dictionary extras which are significant strings such as reserved keywords in the target program. AFL also has the ability to identify important keywords to some extent through analyzing the program behaviour when certain bytes are

Table 2.1: Mutation Operators Used in AFL Havoc Stage.

Type	**Description**	**Mutation Operator(s)**
bitflip	Flip a bit at a random position.	*bit_flip*
interesting values	Set bytes with hard-coded interesting values.	*byte_int_val* *word_int_val* *dword_int_val*
arithmetic increase	Perform addition operations.	*add_byte* *add_word* *add_dword*
arithmetic decrease	Perform subtraction operations.	*sub_byte* *sub_word* *sub_dword*
random value	Randomly set a byte to a random value.	*set_byte_val*
delete bytes	Randomly delete consecutive bytes.	*delete_bytes*
clone/insert bytes	Clone bytes in 75%, otherwise insert a block of constant bytes.	*insert_bytes*
overwrite bytes	Randomly overwrite the selected consecutive bytes.	*overwrite_bytes*
extras	Insert or overwrite bytes with extras which are meaningful strings to the target (can be user-provided or auto-detected). These are optional.	*overwrite_bytes_extra* *insert_extra*

modified during its deterministic phase. If modifying a certain section of bytes changes the program's execution path distinctly compared to modifying other bytes then AFL realizes that these bytes are important, and adds this token to the dictionary. These automatically detected dictionary extras, known as "auto-extras", can then be used in the same fashion as user-provided dictionary extras.

AFL detects and saves inputs that lead to server crashes. It attempts to determine whether crashes are unique by comparing the edge tuples hit. A crash is considered to be unique if it contains an edge tuple not hit by other crashes, or if an edge tuple that is present in other crashes is absent. Therefore, if the same edges are hit by two different crash-causing inputs, it is considered to be the same crash that is being triggered.

To prevent seeds from growing excessively during the mutation process, AFL uses seed trimming which cuts down the size of a seed input file. This seed trimming strategy uses program feedback to remove data from a seed while not impacting the code that it triggers [1]. Overall, AFL's design established a strong baseline for mutational fuzzing and influenced many subsequent fuzzers.

2.2 Protocol Fuzzing

With the great success of binary fuzzing, the technique has expanded beyond traditional binary targets to a variety of software. One important application is protocol fuzzing, where the target is an implementation of a network protocol. These targets are more challenging than single-input binaries because they involve stateful interactions, multiple message exchanges, and some level of syntactic and semantic correctness to reach deeper parts of the program. Early protocol fuzzers such as Peach [26] and Sulley [27] were primarily generation-based and required user-defined protocol models to create structured test inputs, making them more difficult and time-consuming to set up. With the rise of coverage-guided mutational fuzzers like AFL, researchers began exploring how these techniques could be adapted for stateful network protocols.

2.2.1 AFLNET

AFLNET [6] is a popular AFL-based fuzzer designed specifically for fuzzing stateful network protocols. It adapts AFL's evolutionary feedback loop to work with protocol message sequences and state transitions. This enables AFLNET to discover bugs deeper in the program that depend on message

order and server state. Since its release, AFLNET has been well received and many researchers have extended it with new techniques [7].

One of the key contributions of AFLNET is integrating protocol state tracking into the greybox fuzzing process. In its original implementation, the server response code is used to indicate the current protocol state. An Implemented Protocol State Machine (IPSM) is used to model internal states and a hashmap tracks the seeds that lead to a given state. Three state and seed selection options are available: random, round-robin, and favor, which prioritizes less explored states or seeds. AFLNET also supports an optional state feedback mode, where test cases that trigger a new state or state transition are deemed "interesting" and saved in the seed corpus. In follow-up experiments [7], however, the authors found that using both state and code feedback resulted in no significant improvement in code coverage over code feedback alone.

AFLNET uses a state-aware mutation strategy that selectively targets program states for exploration and helps discover new state transitions. Given a message sequence M that reaches the target state, it is split into three subsequences: M_1, the prefix sequence needed to reach the target state; M_2, the candidate subsequence containing messages that can be sent to the server while it remains in the target state; and M_3, the suffix containing the remaining messages. Only M_2 is mutated, preserving the required messages while allowing exploration of the selected state, and possibly state transitions.

In addition to AFL's mutation operators, AFLNET introduces protocol-aware mutation operators that modify the message sequence. These mutations that operate at the sequence-level are referred to as region mutations, and include the following operations: *duplicate*, *replace*, *append*, and *prepend*. Insertion (*append*/*prepend*) and *replace* operations pull from messages parsed from the seed corpus. Mutating the candidate subsequence in this way helps the fuzzer explore new states and transitions.

Unlike AFL, AFLNET does not perform seed trimming. Pham et al. [6] do not address this decision in their paper, but it is likely because trimming is harder to achieve in this context. It is more likely to invalidate protocol syntax or disrupt message formats leading to the fuzzer being unable to obtain the same coverage. We speculate that the omission of seed trimming can also lead to excessively long inputs that slow down fuzzing as seeds grow with little constraint.

2.2.2 *ProFuzzBench*

ProFuzzBench [28] was introduced in 2021 as an open-source standardized benchmark for evaluating stateful protocol fuzzers. It also helps to simplify the fuzzing workflow by providing scripts that automate the build, execution, and analysis phases. For reproducibility, *ProFuzzBench* uses Docker containers for each fuzzing session, ensuring identical environmental setup [29].

To improve comparability across runs, *ProFuzzBench* standardizes initial seeds and dictionary extras, and also applies patches to targets to remove randomness and to speed up execution. Since many trials are required due to fuzzing's inherent randomness, *ProFuzzBench* supports parallel experiments across multiple cores to facilitate larger scale testing.

ProFuzzBench uses GNU's code coverage tool *gcov* [30] to produce the final coverage metrics once the fuzzing session is complete. This is more accurate but slower than AFL's coarse but lightweight coverage tracking, making it suitable for final coverage reporting post fuzzing. The *gcov* tool computes branch and line coverage metrics.

ProFuzzBench disables the deterministic stage inherited by AFL. Although not stated directly by its authors, we infer that this is because AFL's deterministic stage has been shown to be very time-consuming and less effective in increasing code coverage compared to its havoc stage [25], prompting many AFL offshoots to remove this stage [31].

The benchmark currently includes 13 protocol targets, chosen for their maturity, popularity, and use in research. These include common protocols such as FTP, Simple Mail Transfer Protocol (SMTP), Secure Shell (SSH), and Domain Name Service (DNS). Overall, *ProFuzzBench* is a practical benchmark for protocol fuzzing which helps facilitate comparative experiments in research.

2.3 Multi-Armed Bandit

The Multi-Armed Bandit (MAB) problem involves choosing among several options with unknown chances of reward with the goal to maximize the rewards over time. The name originates from the scenario of a player deciding which slot machines (bandits) to play over repeated rounds. In MAB terminology, the possible actions are referred to as arms, and a decision must be made on which arm to pull every round. Each time an arm is selected, the MAB agent receives the reward for the selected arm only. Since initially each arm's probability of reward is unknown, reward feedback from previous rounds must be used to guide decisions. MAB algorithms have been widely applied in domains such as medicine, online advertising, and economics [32].

Fundamentally, the MAB problem centres on the trade-off between exploration and exploitation. In the context of mutation operator scheduling, this means balancing the exploration of less used operators with the exploitation of those that have performed well in the past. The classic stochastic version of the MAB problem assumes rewards are Independent and Identically Distributed (IID) [32]. Performance can be measured by *regret*, the difference between the reward obtained and the maximum possible reward. In fuzzing, maximum reward is unknown, making exact regret calculations not feasible. However, we can consider theoretical regret bounds, and empirical performance to inform algorithm selection.

Several algorithms have been proposed for use in MAB settings. There are simple approaches such as ϵ-greedy or explore-first, however, they do not have optimal regret or empirical performance. More effective and popular algorithms include those based on confidence bounds or Bayesian inference. The Upper Confidence Bound (UCB) algorithm [33], for instance, follows the principle of "optimism in the face of uncertainty" by giving an exploration bonus to lesser explored arms during the decision process. On the other hand, Bayesian bandits model the chance of reward for each arm as a probability distribution, and update it each round with the new reward information.

There also exists adversarial bandits where little to no assumptions can be made about the environment, unlike stochastic bandits. In this scenario, the Exponential-Weight Algorithm for Exploration and Exploitation (Exp3) algorithm is a popular approach which estimates the reward chance of arms using exponential weighting. Since adversarial algorithms tend to have higher regret compared to stochastic algorithms [34], it may be preferable to select stochastic algorithms even if the environment is not perfectly stochastic.

2.3.1 Thompson Sampling

Thompson Sampling (TS), proposed by W.R. Thompson in 1933 [35], was the first bandit algorithm created and is still considered to be one of the best thanks to more recent research [36–38] demonstrating both its theoretical and empirical advantages. Each arm is associated with a probability distribution representing its chance of reward. At each round, a sample is drawn from each arm's distribution and the arm with the highest sampled value is chosen. Once the reward is observed, the chosen arm's distribution is updated with this information.

This randomized sampling achieves a balance between exploration and exploitation where under-explored arms have wider distributions which increases their chance of being selected, while consistently well-performing arms have

higher mean values leading to their selection and exploitation. TS is asymptotically optimal and has been shown to perform better than UCB and its variants in longer trials [37]. Furthermore, it is simpler to implement compared to Kullback-Leibler(KL)-UCB and Bayes-UCB [37], and more robust to delayed feedback than UCB algorithms [38]. Algorithm 1 illustrates TS.

Algorithm 1 Thompson Sampling, modified from [32].

1: **for** each round $t = 1, 2, \dots$ **do**
2: Observe $H_{t-1} = H$, for some feasible $(t-1)$-history H.
3: **for all** arm a **do**
4: Sample mean reward $\mu_t(a)$ independently from distribution $\mathbb{P}_H^a$.
5: **end for**
6: Choose an arm with largest $\mu_t(a)$.
7: **end for**

When the only possible rewards are 0 or 1, the problem is referred to as a Bernoulli bandit [39]. For this type of bandit, Agrawal and Goyal [36] recommend using a Beta distribution for TS, since it updates cleanly with Bernoulli rewards. It has two parameters, $\alpha = s+1$ and $\beta = f+1$, where s and f are the number of successes and failures observed for that arm. Updating the posterior after each trial therefore requires only incrementing α by 1 for a success, or β by 1 for a failure.

2.3.2 Linear Thompson Sampling

In many scenarios, rewards depend not only on the action taken but also on additional features of the environment. In a contextual bandit setting, these factors are revealed each round and can be captured in a context vector. A commonly used algorithm for linear contextual bandits is Linear Upper Confidence Bound (LinUCB) [40], which extends UCB to incorporate contextual features. LinUCB has good theoretical regret, but Chapelle and Li [38] found it to have higher empirical regret in their experiments.

Alternatively, there is a linear contextual bandit algorithm based on TS called Linear Thompson Sampling (LinTS) [41]. This section focuses on "Algorithm 3" from Agrawal and Goyal [41] since it is widely used, such as in the TensorFlow implementation for LinTS [42]. The algorithm is shown in Algorithm 2.

In this algorithm, each arm a maintains a Bayesian linear model that relates the context vector $x(t)$ of length d to the expected reward. The model consists of B_a, initialized as a $d \times d$ identity matrix, and f_a, initialized as a zero

Algorithm 2 Linear Thompson Sampling with N arms, modified from [41].

Init: For each arm $a = 1, \dots, N$: $B_a \leftarrow I_d$, $\hat{\mu}_a \leftarrow \mathbf{0}_d$, $f_a \leftarrow \mathbf{0}_d$
for $t = 1, 2, \dots$ **do**
  Observe context vector $\mathbf{x}(t)$
  for each arm $a = 1, \dots, N$ **do**
    Sample $\theta_a(t) \sim \mathcal{N}\big(\hat{\mu}_a,\ \nu^2 B_a^{-1}\big)$
    Score $s_a(t) \leftarrow \mathbf{x}(t)^\top \theta_a(t)$
  end for
  Play arm $a(t) \leftarrow \arg\max_a s_a(t)$ and observe reward r_t
  $B_{a(t)} \leftarrow B_{a(t)} + \mathbf{x}(t)\mathbf{x}(t)^\top$
  $f_{a(t)} \leftarrow f_{a(t)} + r_t\,\mathbf{x}(t)$
  $\hat{\mu}_{a(t)} \leftarrow B_{a(t)}^{-1} f_{a(t)}$
end for

vector of length d. The posterior distribution for an arm is constructed with $\nu^2 B_a^{-1}$ as the covariance matrix that describes the shape of the distribution, and the mean $\hat{\mu}_a$ is given by $B_a^{-1} f_a$. The ν value is a hyperparameter related to exploration. Throughout the algorithm, the B_a matrix gathers information about the context vector, while f_a collects information on how context relates to reward.

At each round t, the algorithm observes the context vector $x(t)$. For each arm, it samples a weight vector $\theta_a(t)$ from the Gaussian distribution formed using $\hat{\mu}_a$ and $\nu^2 B_a^{-1}$. The score for the arm is then computed using $x(t)$ and $\theta_a(t)$, and the arm with the highest score is chosen. After observing the reward r_t, each component of the model ($B_{a(t)}$, $f_{a(t)}$, and $\hat{\mu}_{a(t)}$) for the selected arm is updated with the new context and reward information.

By maintaining independent posterior distributions for each arm, the algorithm is able to learn the relationship between context and reward for different actions. This makes it suitable for applications such as mutation scheduling, where the context may have a different impact depending on the mutation operator.

2.4 Related Works

This section reviews prior work on mutation scheduling in fuzzing and mutation approaches in greybox protocol fuzzing, with the inclusion of a state selection method that seeks to address a similar underlying problem. The objective is to provide an overview of existing approaches and to identify

techniques that are effective.

2.4.1 Mutation Operator Scheduling in Binary Fuzzing

Since 2018, methods for adaptively optimizing the selection of mutation operators has been explored in binary fuzzing research [43–45]. Mutation scheduling was popularized by MOpt [25] in 2019, and the technique was integrated in AFL++ [23] in 2020.

MOpt's [25] application of the Particle Swarm Optimization (PSO) to dynamically adjust operator probabilities in response to program feedback achieved higher coverage when integrated with existing fuzzers, with minimal overhead according to their results. DARWIN [46] introduced an evolutionary strategy that perturbs the current mutation operator probability distribution, known as the "parent", to create "child" distributions. After testing each "child" distribution, the best performing one is selected to be the next "parent". Compared to MOpt, DARWIN required less parameter tuning and achieved higher coverage across 10 open-source programs. AMSFuzz [47] employed a custom MAB algorithm for mutation operator selection, along with introducing seed slicing to improve efficiency of the deterministic phase. In the author's evaluation, it overall outperformed AFL [1], AFLFast [48], FairFuzz [49], and MOpt [25] in code coverage.

Contextual Mutation Operator Scheduling

Several fuzzers have incorporated context such as seed characteristics into their mutation scheduling. CMFuzz [50] used LinUCB to select operators based on features extracted from seeds. This yielded higher coverage and more crashes than non-contextual MAB-based schedulers in the author's experiments. SeamFuzz [51] clustered seeds based on similar characteristics and then applied a customized contextual Thompson Sampling for mutation operator selection. They also included triggering rare code transitions as a reward signal, since triggering new code coverage happens less than 0.001% of the time [51]. SeamFuzz achieved more coverage and crashes than AFL++ [23] and MOpt [25] in their experiments. MesFuzz [52] similarly clustered seeds, but used a multi-population evolutionary strategy for mutation scheduling. This is similar to DARWIN, but with a different evolving parent distribution for each seed cluster. MesFuzz outperformed DARWIN, SeamFuzz, and AFL++ on the LAVA-M dataset [53]. FA-Fuzz [54] optimized operator scheduling using the Firefly Algorithm (FA) [55] with one model per seed. After the first run for a seed, it sticks with this distribution and stops updating it

to reduce overhead. Integrated as FA-AFL, it generally performed best on test programs compared to AFL++ and MOpt, although those fuzzers achieved better performance on certain targets, showing how different strategies work better for different targets.

Combined Mutation Scheduling

Other works have combined mutation operator scheduling with the adaptive selection of stack size and mutation type in a similar manner. $Havoc_{MAB}$ [24] used a two-layer MAB to choose stack size and mutation type. Its experiments showed that smaller stack sizes (such as 2, 4, 8) often outperform larger ones (such as 64, 128), and that the best balance of mutation types (unit or chunk mutators) depended on the target. Their adaptive approach achieved superior edge coverage on most benchmarks compared to QSYM [56]. SLOPT [57] selected both mutation operators and stack sizes using MAB algorithms. They also used seed clustering to select stack sizes based on seed length. This fuzzer consistently achieved the best edge coverage across tested programs (AFL++ [23], MOpt [25], CMFuzz [50], Karamcheti [45], and HavocMAB [24]). Their experiments also showed that larger seeds benefit from higher stack sizes, while smaller seeds perform better with fewer mutations. Importantly, SLOPT compared multiple MAB algorithms from stochastic to adversarial, and found TS consistently performed best, suggesting mutation scheduling works well with Bayesian stochastic bandit models.

2.4.2 Protocol Fuzzing Mutation Strategies and Other Related Research

Yang et al. introduced mutation scheduling to greybox protocol fuzzing with implementing a hybrid FA [55] and PSO scheduler into AFLNet [13]. Their method used FA in the first phase of fuzzing when coverage was less than 30%, and switched to PSO after the coverage passed this threshold to improve efficiency. Their evaluation on average across eleven *ProFuzzBench* targets reported a 3.28% coverage increase compared to AFLNet. However, they did not perform statistical assessments, and their four experiment repetitions were well below the suggested minimum of ten [18], making the significance of their results unclear.

RLGFuzz [14] addressed the problem of excessively long message sequences during fuzzing, which degrade coverage, with a reinforcement learning model that guides seed and state selection based on their contribution to coverage.

Their experiments out-performed AFLNet and SMGFuzz [15], demonstrating that adaptive selection can mitigate this inefficiency.

In AFLNet-packmute [9], a syntax-aware mutation strategy was integrated into AFLNet for FTP targets which restricted mutable ranges for certain mutation operators. Additionally, an ablation study was conducted across all *ProFuzzBench* targets which revealed that disabling region mutations improved coverage for most targets. Only a few complex, stateful protocols benefitted from enabling region mutations. Furthermore, when we examine the highest performing configurations of the targets where region mutations are preferred, they were all where statefulness was enabled in AFLNet. In comparison, for the other protocol targets, the majority had their highest performance without statefulness enabled. This observation indicates a complimentary relationship between region mutations and statefulness in protocol fuzzing.

Other research on mutations in protocol fuzzing includes syntax, semantic, structure, and position-focused approaches [8, 10–12]. ChatAFL [8] significantly improved code coverage by integrating structure-aware mutations into AFLNet, using a large language model to build grammars from protocol specifications [8]. Han et al. segmented protocol messages into semantic fragments using target responses, enabling the application of mutations that conform to the inferred semantic constraints [11]. Hu et al. developed a fuzzer that used tree-based mutations to preserve TLS message structure [10]. GONet applied a neural network to guide mutation by selecting which bytes to mutate based on the model's gradient [12].

In this chapter, we saw how mutation operator scheduling has improved binary fuzzing from one layer scheduling methods to more recent multi-layered and contextual scheduling improvements. Finally, relevant research in protocol fuzzing was explored. From these related works, adaptive mutation operator scheduling shows good potential in improving the application of region mutations in greybox protocol fuzzing.

3 Methodology and Design

This research seeks to enhance AFLNET by implementing a target-adaptive mutation operator scheduling approach to improve code coverage and by extension, bug-finding. This chapter will outline the design, development, and evaluation of the mutation strategy, AFLNET-TAMS.

The first phase of this research involves a preliminary assessment of mutation operator performance through four experiments across five *ProFuzzBench* protocol targets. The first experiment tracks operator effectiveness throughout the fuzzing process, and the second varies the probability of region mutations. The third experiment tests different stack sizes for region mutations, and the fourth experiment introduces a new region mutation operator, *delete*. These experiments investigate how these factors influence performance and inform the TAMS design process.

The second phase is the design and development of AFLNET-TAMS, a derivative of AFLNET that uses a target-adaptive mutation operator strategy. Mutation operators are selected using the MAB algorithms: TS and LinTS. The TAMS process first chooses between the region and byte mutation categories using TS, followed by operator selection within the selected category. For region mutation operator selection, a LinTS agent considers sequence length as context to learn the relationship between seed length and operator effectiveness. A *delete* region mutation operator is introduced to aid in managing the growth of sequences caused by other region mutators.

The third phase verifies that AFLNET-TAMS behaves as intended. This entails verifying that both the mutation type and the region mutation operator selection are adapting according to their selected algorithms using the code coverage feedback.

The fourth phase performs the validation of the research aim. AFLNET-TAMS is evaluated against AFLNET with its default mutation strategy with region mutations and state-awareness enabled. The change in code coverage is measured for five different *ProFuzzBench* targets: LightFTP (FTP), DNSmasq (DNS), Exim (SMTP), OpenSSH (SSH), and Kamailio (SIP).

3.1 Preliminary Mutation Experiments

The preliminary experiments focus on evaluating the effectiveness of mutation operators and investigating additional mutation parameters to inform the design choices of AFLNET-TAMS and to better understand AFLNET's mutation process as a whole. Four experiments are conducted during this phase, each focusing on a specific aspect: operator performance, the probability of region mutations, stack size for region mutations, and the addition of the *delete* region operator. The details of these experiments are presented in Table 3.1.

Table 3.1: Preliminary Experiment Design.

Experiment	AFLNET Modification	Experimental Approach
Operator Performance	None	Analyze operator success rates across various targets and over time.
Probability of Region Mutation	Modify probability of selecting region mutation	Run experiments with the chance of selecting region mutations half and twice as likely compared to the default. Stacking behaviour remains unchanged.
Stack Size for Region Mutation	Separate region and byte mutation stacking; vary region stack size (4 fixed values)	Test four fixed stack sizes for region mutators: 1, 4, 16, 32. Byte mutator stack size remains at the default range (2 to 128). Region mutation rate remains at 19%.
Addition of Delete Operator	A *delete* region mutation operator is added to the region mutations	Compare coverage performance with new operator to that of the unmodified AFLNET to understand impact of *delete*.

As per best practice in research, each experiment is run for 24 hours. Furthermore, every experiment is run 10 different times to account for randomness. For better generalization, a variety of protocols are tested: FTP, DNS, SMTP, SSH, and SIP. Additionally, a mutation log is added to AFLNET for each experiment, tracking the operators applied to each seed and its outcome

("interesting" or no new bitmap coverage) to aid in analysis.

The performance of these experiments is primarily evaluated using code coverage results, where higher coverage relative to other configurations indicates better performance. The first experiment, which examines mutation operator performance, is analyzed differently by measuring the number of test cases that trigger new bitmap coverage for each mutation operator. Since no modifications are made to the fuzzer behaviour, the coverage results from this experiment also serve as the baseline AFLNET performance.

3.1.1 Mutation Operator Performance

Using the mutation log, the number of "interesting" test cases produced with each mutation operator is examined. This experiment provides insight into how operator effectiveness varies during the fuzzing process and across targets. These results help validate our hypothesis that mutation operator performance depends on the target.

3.1.2 Probability of Region Mutation

Experiments are conducted varying the probability of region mutations. The probability is made to be half (9.5%) and twice (38%) as likely relative to the default 19%. Other experimental factors remain unchanged, such as mixed type stacking. This section investigates our hypothesis that the optimal amount of region mutations varies depending on the target protocol.

3.1.3 Stack Size for Region Mutation

Another hypothesis in the design of AFLNET-TAMS is that region mutation operators have better performance with fewer stacked mutations. Since region mutations alter the seed's structure, they tend to result in more significant changes compared to byte mutations. It follows that stacking many region mutations could lead to overly distorted seeds. Therefore, lower stack sizes may be preferable when applying region mutations. This experiment tests different fixed stack sizes (1, 4, 16, 32) for region mutations to investigate their impact. AFLNET selects stack sizes using random powers of two from 2 to 128, so we adopt a similar range, concentrating on smaller sizes. The experiment helps validate our choice to reduce the stack size for region mutations.

3.1.4 Addition of *Delete* Operator

This experiment introduces a new region mutation operator: *delete*. This mutator randomly selects a message in the subsequence M_2 (as described in Section 2.2.1) to be deleted. The chance of selecting the operator is the same as the other mutation operators.

3.2 Development of AFLNET-TAMS

This section describes the design of AFLNET-TAMS, a derivative of AFLNET which adapts the selection of mutation operators based on their historical effectiveness during the fuzzing process. The key innovations designed into AFLNET-TAMS are two adaptive selection agents: a mutation type selector that chooses between region and byte mutations, and a region mutation operator selector that selects operators within the region category. Both components leverage MAB models, with the region operator selector also incorporating seed length as a contextual feature. Additionally, to help mitigate the uncontrolled sequence growth associated with the current region mutation operators, the *delete* region operator described in Section 3.1.4 is added.

The AFLNET-TAMS fuzzing process largely follows the same pipeline as AFLNET. First, a target state is selected using AFLNET's favor state selection strategy, and then a seed that reaches this state is chosen from the corpus. Once the seed is selected, AFLNET-TAMS uses its mutation type selector, a TS MAB algorithm which chooses whether to perform region or byte mutations. If region mutations are selected, this activates the second MAB agent, a LinTS model that decides which operators to apply based on their past performance and considering sequence length as a feature. Within the chosen category, mutation operator selection is repeated so that multiple different mutation operators are stacked together to create the final test case which is executed on the target server. Code coverage feedback from the execution is used as the reward signal to update both the mutation type selector and the region operator selector. Over successive rounds, the mutation type selector (TS) learns which type of mutation to favour, while the region operator selector (LinTS) learns a linear relationship between sequence length and the expected reward for each operator. Figure 3.1 illustrates the overall TAMS process, which will be further explained in this section.

Our adaptive strategy is applied to AFLNET's core havoc mutation stage, and by extension, the splicing stage, which reuses the havoc code after performing a crossover mutation of two seeds. The seed selection, state selection, and state modelling remain unchanged from AFLNET and are outside the

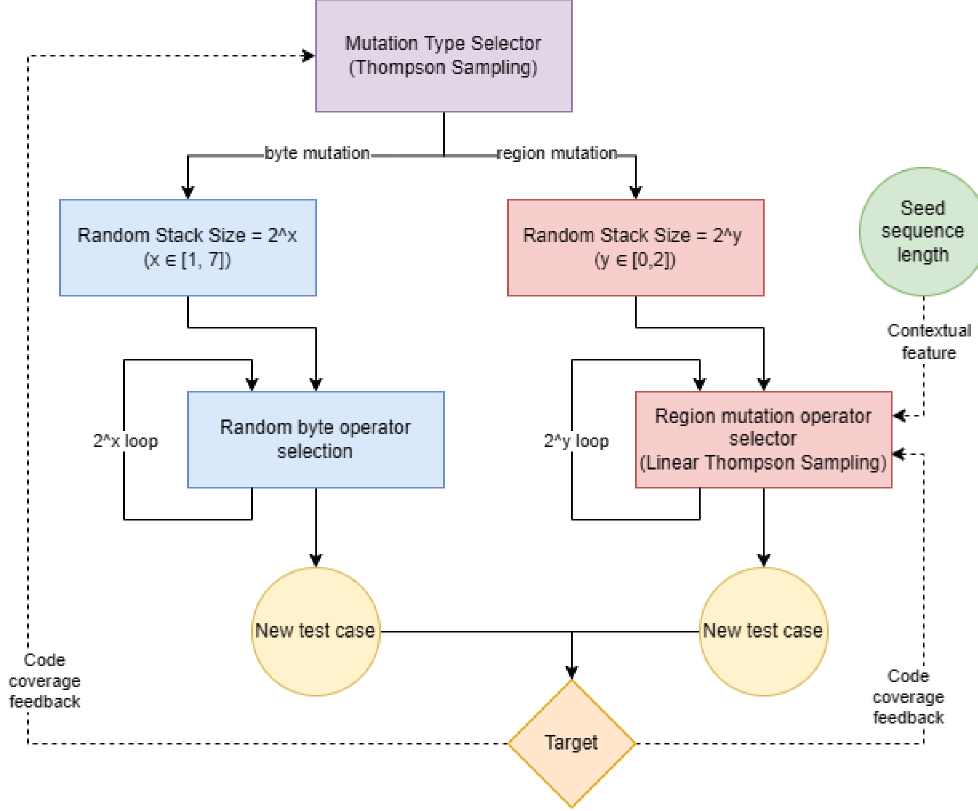

Figure 3.1: TAMS Process

scope of this research. Table 3.2 summarizes the main differences in mutation strategy between AFLNET and AFLNET-TAMS.

3.2.1 Mutation Type Selection

The first step in the TAMS process is to select between a region mutation and a byte mutation using a TS MAB agent. The theory behind this design is that the default amount of region mutations is not optimal for most targets. By adaptively selecting region and byte mutations based on their coverage performance, a more optimal amount of region mutations could be found for the target.

Region mutations and byte mutations serve different purposes. Region mutations tend to explore new protocol states by rearranging message sequences, while byte mutations focus more on exploring code within a selected

Table 3.2: Comparison of Mutation Strategies between AFLNET and AFLNET-TAMS.

Component	AFLNET	AFLNET-TAMS
Mutation Type Selection	No distinction between mutation types. All mutation operators are in a single random selection pool.	Uses a TS MAB to choose between region and byte categories, adapting based on which category historically yields more coverage.
Region Operator Selection	Uniform random selection of region mutation operators within the full mutation operator selection pool.	Uses LinTS to select operators, using seed sequence length as a feature. A *delete* message operator is added.
Byte Operator Selection	Random selection of byte mutation operators within the full mutation operator selection pool.	Similar to AFLNET, random selection of byte mutation operators is maintained (given that mutation type is selected).
Stack Size for Region Mutations	Random stack size: 2, 4, 8, 16, 32, 64, or 128. Stacked together with byte mutation operators from the full selection pool.	Random stack size: 1, 2, or 4. Only stacked together with other region mutation operators.
Stack Size for Byte Mutation	Random stack size: 2, 4, 8, 16, 32, 64, or 128. Stacked together with region mutation operators from the full selection pool.	Random stack size: 2, 4, 8, 16, 32, 64, or 128. Only stacked together with other byte mutation operators.

state. Furthermore, it is hypothesized that some protocols benefit from a reduced proportion of region mutations and vice versa. For instance, a protocol that has many states with significant depth, making it highly stateful, would benefit from more region mutations compared to another target. In contrast, simpler protocols such as FTP likely benefit from less region mutations as the state space is simpler, therefore more byte mutations should be used to focus on deeper exploration of existing states instead.

The TS MAB algorithm was chosen for its excellent theoretical performance [36, 37] and its superior empirical performance both broadly [38] and in mutation scheduling research in particular [57]. The reward in this scenario is binary: a reward of 1 is assigned if the resulting mutated test case increases code coverage (as detected by AFLNET's bitmap), and 0 otherwise. The mutation type selector is executed once per mutation process. Unlike AFLNET, AFLNET-TAMS does not mix region and byte mutations in the stacking of operators. Instead, once a type is chosen, all mutation operators stacked together for that mutation session are of the same type. Since protocol targets vary in stateful behaviour, the optimal balance between region and byte mutations is expected to be target-dependent. If both mutation types were stacked together, it becomes difficult to isolate how each type contributed to a coverage increase, creating a reward attribution issue with the MAB. Separating the stacking of the two types ensures the reward signal reflects the performance of each type individually, improving the decision-making of the MAB.

3.2.2 Region Mutation Operator Selection

The region operator selector uses the LinTS algorithm to choose among the region mutation operators. The current seed length is used as a contextual feature, as it is hypothesized that operator effectiveness can depend on sequence size. For instance, when sequences are very long, the *duplicate* operator which doubles sequence length is less preferable compared to others such as the new *delete* operation. By sampling from the posterior distribution, the agent introduces randomness that reflects uncertainty, allowing it to explore less frequently used operators while still prioritizing those that have performed well.

The stack size for region mutations is limited to a maximum of four for two reasons. First, region mutations have a larger impact on the seed compared to byte mutations since they are acting at a higher level and rearranging the message sequence. We hypothesize that the application of too many region mutation operators at once can excessively distort the seed, producing very long inputs with too many inserted, replaced, or duplicated messages, which may reduce the effectiveness. Second, with higher stack sizes comes the difficulty in attributing the success of various mutation operators. For instance, when many different mutation operators are stacked to create an "interesting" test case, it becomes difficult to say which mutation operator(s) led to the success of the test case. While limiting the stack size does not isolate success attribution, it reduces misattribution since there are fewer operators

involved. A stack size of four was used previously by Wang et al. [50] and Karamcheti et al. [45] with good results.

With the maximum mutation stack reduced, each operator that contributes to a test case resulting in increased code coverage has its corresponding MAB arm updated with the reward signal. All operators in the current region mutation stack receive equal credit (reward = 1), as the exact operator(s) responsible cannot be isolated. For the reward signal, AFLNET-TAMS utilizes the edge coverage feedback that is already employed by AFLNET as part of its feedback loop.

3.2.3 Byte Mutation Operator Selection

The uniform random selection of mutation operators is maintained from AFLNET for byte operator selection. Additionally, the default stack size is retained. Research has shown that stacking multiple mutation operators randomly like in the havoc stage has good fuzzing outcomes [24]. Since byte mutations make smaller overall changes to the seed compared to region mutation operators, it makes sense to stack more of them.

3.3 Verification

The purpose of the verification phase is to ensure the implementation of AFLNET-TAMS aligns with its intended design. It should select mutation type based on past coverage bitmap results and should select region mutation operators based on both current seed length and historical coverage results.

To verify this functionality, fuzzing is conducted against a *ProFuzzBench* target for a short duration. During these experiments, detailed logging of the MAB agents is enabled. These logs record information related to the MABs decision process. The type selector MAB log contains information such as the number of uses for each arm, rewards for each arm, sampled values, and chosen mutation type. The region operator selector MAB log includes each arm's posterior parameters, the current seed length, arm samples, and chosen region operator.

The verification involves analyzing these logs to confirm that the algorithms behave as expected. This includes confirming that samples align with their posterior distributions, that the arm with the highest sampled value is selected, and that the posterior parameters are updated correctly after the reward of the resulting mutated test case is received.

Additionally, verification of the *delete* operator is performed using a separate log which records the message sequence before and after each deletion.

Inspecting these logs confirms that the correct message is removed from the M_2 subsequence and that the remaining messages maintain their original order. The verification results are presented in Section 4.4.

3.4 Validation

During the final phase, the aim of the research is validated. The impact on code coverage of TAMS is measured to determine if it leads to measurable changes in code coverage performance.

To achieve this end, experiments are conducted on five different ProFuzzBench targets: LightFTP, DNSmasq, Exim, OpenSSH, and Kamailio. These targets were chosen to cover a variety of protocols with different state models and characteristics. Table 3.3 shows a comparison of the targets' features.

Table 3.3: Comparison of Selected Protocol Targets.

Target	**Protocol**	**Format**	**Encryption**	**Stateful aspects**
LightFTP	FTP	Text-based	Unencrypted	Authentication state, working directory, mode
DNSmasq	DNS	Binary	Unencrypted	DNS record caching
Exim	SMTP	Text-based	Unencrypted	Session progress, message queue
OpenSSH	SSH	Binary	Encrypted	Session progress, authentication state, session keys
Kamailio	SIP	Text-based	Unencrypted	User registrations, session progress

The two different MAB selectors of AFLNET-TAMS are evaluated together and separately to measure their individual performance and determine whether combining them results in a positive or negative effect. Table 3.4 shows the details of these test configurations. Each configuration is tested in 24-hour fuzzing sessions. The same experimental setup described in Section 3.1, including the mutation log, is used. Additionally, 20 individual runs are conducted for each target in the different configurations to better measure performance while minimizing the impact of randomness. The Mann-Whitney U

test is used to determine statistical significance of the results with a p-value less than 0.05 indicating a statistically significant difference.

Table 3.4: Validation Test Configurations.

Configuration	**Description**
Baseline (AFLNET)	Default mutation strategy of AFLNET.
TAMS: Type Selection only	Adaptive selection of mutation types based on coverage feedback, with separate stacking of the mutation categories.
TAMS: Region Operator Selection only	Adaptive selection of region mutation operators based on coverage feedback and seed length. Byte and region mutations are stacked separately with the chance of region mutation at default 19%.
TAMS: Full Combined	Combines adaptive mutation type selection with contextual selection of region operators, also limiting stack size to four.

Performance is assessed using median branch coverage metrics from *gcov*. The baseline AFLNET performance is compared to the TAMS variants to determine the change in coverage. The aim of this research is successfully validated once the performance difference is measured.

3.5 Summary

This research introduces a mutation strategy for AFLNET that adapts the application of region mutations using target feedback. Preliminary experiments evaluate how mutation operators perform under different configurations, guiding implementation of the new AFLNET variant, AFLNET-TAMS. The mutation strategy leverages TS for lightweight adaptive behaviour based on code coverage rewards. Two separate MAB agents are used, one for mutation type selection and the other for region mutation operator selection with seed length as a contextual feature. Additionally, a *delete* region mutation operator introduces more variety in the region mutations and assists in managing seed

growth. Experiments across five diverse protocol targets are used to determine the impact on code coverage of this novel mutation strategy compared to AFLNET's default mutation strategy.

4 Results

This chapter covers the development and results of AFLnet-TAMS. It starts with detailing the experimental design, before moving into the implementation of the design. In the verification section, the intended behaviour of the new AFLnet offshoot is confirmed to ensure it meets design specifications. The validation section will present the final performance results which validate our aim. Following this is a discussion section that analyzes the outcomes.

4.1 Experimental Design

The experiments are conducted using four Dell workstations with 24 cores and 32GB RAM. Each workstation runs two Virtual Machine (VM)s, each with 12 cores and 8GB of RAM assigned. Every VM runs ten fuzzing experiments in ten different Docker containers. This setup enables 80 experiments to be run concurrently across the four workstations.

To run experiments, all VMs were accessed remotely through the SSH protocol from a laptop computer. Each Dell workstation was connected to a router and configured to port forward port 1000 to its first VM and port 2000 to its second VM. This allowed all VMs to the accessed remotely at the same time from one computer connected to the network.

Since the *ProFuzzBench* Docker images clone code from external repositories, the experiment VMs required internet access through their router. *ProFuzzBench* uses Ubuntu 20.04 for its Docker images, which we maintain. The target seeds provided by *ProFuzzBench* were used for all experiments. Additionally, *ProFuzzBench* disables the AFL deterministic stage when running AFLnet, and this setting was retained for our experiments. Furthermore, the *ProFuzzBench* Docker files were modified to clone our AFLnet-TAMS project hosted on GitHub [58], with an additional argument added to allow selection of different project branches.

Our AFLnet-TAMS was built on AFLnet git commit version 96032f8,

the most recent available at the time. The *ProFuzzBench* version is git commit 8573ec8. The different versions of our experiments were managed using git branches.

A mutation log was added to AFLNET to track mutation specific details. The log was implemented as a Comma-Separated Value (CSV) file to be flexible and lightweight. Each time a stacked havoc mutation and its associated execution occur, a line with its details is logged in the file. The information logged includes mutation operators applied, seed used, time stamp, and if the test case was considered "interesting" or not. To track mutation operators, each one was assigned a lowercase character in the alphabet to represent it in the logs since there are 21 different mutation operators including our new *delete* operator. This helps to reduce the size of the logs, since thousands of executions are conducted during a fuzzing experiment. This mutation log aids in our analysis and verification that the correct experiment was run. We chose to conduct all experiments with mutation log enabled, making them comparable and enabling further analysis of the mutation process.

4.2 Preliminary Experiments on AFLNET

Preliminary experiments were conducted to better understand how AFLNET's mutation strategy functions and the impact of the mutation factors. The results of these experiments can be seen in Table 4.1, and the individual experiment results are discussed in Sections 4.2.2, 4.2.3, 4.2.4, and 4.2.5. The highest values in the median and mean columns are bolded, while values below 0.05 in the p-value column are bolded, representing statistically significant results.

4.2.1 Development of *Delete* Operator

To develop the new *delete* random message operator, we made use of AFLNET's existing *extract_requests* function which extracts the individual messages from M_2, the candidate subsequence, stored in a buffer. The result is returned in an array of *region_t* structures where each element correlates to a message. The structures specify the starting and ending bytes for their messages within the buffer. The message to delete is chosen by randomly selecting the index of a message using AFL's *UR* function. Then the buffer is reconstructed excluding the selected message's bytes.

In developing this new operator, a minor issue with DNSmasq's *extract_requests* function was encountered. When parsing a DNS message, the function expects that the first null byte after the DNS header indicates the end of the query

Table 4.1: Branch Coverage Results for Preliminary Experiments.

Experiment	Branch Coverage				p-value
	Min	Max	Med	Mean	
LightFTP					
Operator Performance (Baseline)	340	361	347.5	350.3	
Region Mutation Probability					
Double Probability	340	364	346.0	348.6	0.8794
Half Probability	341	366	**362.5**	**356.9**	0.0580
Region Mutation Stack Size					
Stack Size = 1	341	366	345.0	348.6	0.7322
Stack Size = 4	341	363	347.5	351.2	0.6488
Stack Size = 16	340	363	345.0	348.8	0.7613
Stack Size = 32	340	363	344.0	346.7	0.4028
Addition of *delete* Operator	341	366	345.0	346.2	0.3427
DNSmasq					
Operator Performance (Baseline)	1113	1130	1115.5	1117.8	
Region Mutation Probability					
Double Probability	1113	1115	1115.0	1114.5	**0.0232**
Half Probability	1112	1127	**1116.0**	1116.9	0.3507
Region Mutation Stack Size					
Stack Size = 1	1112	1127	**1116.0**	1118.4	0.8783
Stack Size = 4	1115	1118	**1116.0**	1115.9	0.6554
Stack Size = 16	1112	1116	1115.0	1114.8	0.2553
Stack Size = 32	1114	1128	1115.0	**1120.1**	0.9048
Addition of *delete* Operator	1115	1128	**1116.0**	1119.0	0.4067

Table 4.1: Branch Coverage Results for Preliminary Experiments. (continued).

Experiment	Branch Coverage				p-value
	Min	Max	Med	Mean	
Exim					
Operator Performance (Baseline)	2884	2977	2910.0	2920.7	
Region Mutation Probability					
Double Probability	2908	2958	2920.5	2930.2	0.2892
Half Probability	2860	2985	2928.0	2925.3	0.6230
Region Mutation Stack Size					
Stack Size = 1	2780	2942	2906.0	2881.3	0.1857
Stack Size = 4	2858	2971	2904.5	2912.3	0.6499
Stack Size = 16	2863	2963	2940.0	2928.6	0.6228
Stack Size = 32	2856	2989	2933.5	2926.3	0.6499
Addition of *delete* Operator	2871	3018	**2941.5**	**2937.0**	0.3640
OpenSSH					
Operator Performance (Baseline)	3326	3385	3355.0	3356.9	
Region Mutation Probability					
Double Probability	3299	3375	3345.0	3341.1	0.1617
Half Probability	3323	3387	3363.5	3360.4	0.5706
Region Mutation Stack Size					
Stack Size = 1	3339	3400	3372.0	3369.9	0.1984
Stack Size = 4	3334	3385	3355.0	3358.8	0.8795
Stack Size = 16	3322	3394	3362.0	3359.2	1.0
Stack Size = 32	3355	3408	**3376.0**	**3380.7**	**0.0257**
Addition of *delete* Operator	3319	3365	3357.5	3350.8	0.8499

Table 4.1: Branch Coverage Results for Preliminary Experiments. (continued).

Experiment	Branch Coverage				p-value
	Min	Max	Med	Mean	
Kamailio					
Operator Performance (Baseline)	9311	9930	**9638.5**	9578.6	
Region Mutation Probability					
Double Probability	9185	9752	9617.0	9535.4	0.9097
Half Probability	9047	9540	9201.5	9255.0	**0.0046**
Region Mutation Stack Size					
Stack Size = 1	9249	9606	9295.0	9346.8	**0.0058**
Stack Size = 4	8871	9354	9168.5	9140.5	**0.0003**
Stack Size = 16	9417	9777	9592.0	**9588.8**	0.9097
Stack Size = 32	9340	9698	9531.5	9519.8	0.7337
Addition of *delete* Operator	9352	9717	9578.5	9547.3	0.6776

and assumes that this null byte is followed by a four byte tail. However, after fuzzing, this assumption may no longer hold if bytes were deleted or otherwise modified. This behaviour could result in an *end_byte* value returned by the function being past the end of the buffer. We accounted for this issue by checking whether the message *end_byte* was past the end of the buffer and setting the *end_byte* to the last byte in the buffer when necessary.

4.2.2 Mutation Operator Performance

To better understand mutation operator performance, for each mutation operator on each target, we calculate a "success rate" which we define as:

$$\frac{\text{num interesting test cases where mut op was used}}{\text{num total test cases where mut op was used}} \times 100\% \qquad (4.1)$$

The values are shown in Appendix A.

From the observed success rate results, we note that the region mutation operator *duplicate* has the highest success rate for each target except OpenSSH. Other region mutation operators such as *prepend* and *append* were

ranked highly for targets LightFTP, DNSmasq, and Kamailio. This outcome is unexpected as previous research showed LightFTP and DNSmasq performed better when region mutations were disabled [9].

We theorize that the *duplicate* operator which duplicates the messages currently in M_2 could lead to code sections in the target program being executed more times, triggering the new hit count "interesting" result. This observation was further explored later in this document in Section 4.3.2.

The *replace* region operator consistently had a low success rate on the targets tested. This outcome is likely due to its behaviour of replacing the entire M_2 subsequence with a message randomly selected from another seed. This can overwrite mutations previously applied during the current mutation stack. Additionally, because the inserted message originates from an existing test case, it has already been seen by the target, and therefore we expect that it is less likely to trigger new coverage.

Since mutation operators are frequently stacked together, they often share rewards, which results in success rates being similar overall. However, there was still variation across targets. The byte operator *insert_bytes* performed highly on LightFTP, DNSmasq, Exim, and Kamailio, but it ranked rather low on OpenSSH. This difference could be related to the fact that it is the only target which uses encryption. Another mutation operator with varied performance is *overwrite_bytes* which had a high success rate on LightFTP, Exim, OpenSSH, and Kamailio but had lower performance on DNSmasq.

Figure 4.1 shows the average number of "interesting" test cases discovered per hour over the course of these experiments. The rate of discovery drops sharply within the first few hours, meaning that the MAB's reward data was dominated by operator successes from early in the fuzzing session. This could have negative impacts if a mutation operator performs well early on, leading the MAB to use it more often. It could become overused, and due to this early data bias, the MAB may fail to recognize and respond to a decline in its effectiveness.

Additionally, Figure 4.1 shows the difference in the number of "interesting" test cases discovered by each target. Many more "interesting" test cases were found for Kamailio than the other targets in a given time period, while fewer were found for LightFTP and DNSmasq.

Although there were several similarities in mutation operator performance across the targets, it still varied across the different targets. These similarities in the results largely come from the new hit count "interesting" test case which is explored further in Section 4.3.2. Furthermore, the rate of "interesting" test case discovery varies greatly over time and across targets.

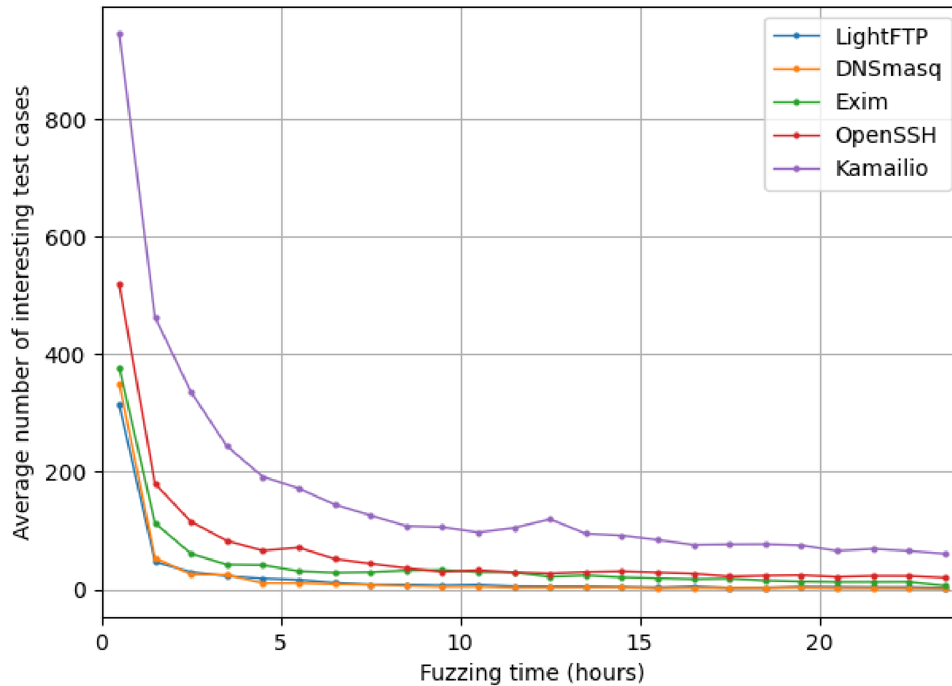

Figure 4.1: Number of "Interesting" Test Cases Discovered over Time.

4.2.3 Probability of Region Mutation

The second preliminary experiment tested different probabilities for region mutations. Specifically, experiments were conducted with the probability of region mutations doubled and halved. The coverage results are compared against the first preliminary experiment, which serves as the baseline using the default probability for region mutations.

Most results were not statistically significant when using the Mann-Whitney U test compared to the baseline AFLNET performance as shown in Table 4.1. The exceptions were a statistically significant decrease in performance when doubling the probability for DNSmasq and when halving the probability for Kamailio. This is consistent with our hypothesis that the beneficial proportion of region mutations varies across targets. When fewer region mutations are used, Kamailio experiences a performance decrease because the default proportion is more suitable for this target. On the other hand, DNSmasq shows decreased performance when the probability is doubled, likely reflecting that the overuse of region mutations is harmful for this target.

Although the other results did not reach statistical significance using the Mann-Whitney U test, it is noteworthy that halving the probability of region mutations resulted in higher mean and median coverage compared to the baseline AFLNET performance for all targets except Kamailio which had lower performance. Furthermore, doubling the probability resulted in lower mean and median coverage compared to the baseline for all targets except Exim. These results support the hypothesis that region mutations are overused for most targets, and further increasing their frequency can worsen performance. Kamailio and Exim are observed to be the targets that benefit from a higher proportion of region mutations compared to the other targets.

4.2.4 Stack Size for Region Mutation

In the third preliminary experiment, region and byte mutations were stacked separately and different stack sizes for region mutations were explored. Most results achieved similar coverage to baseline and were not statistically significant, except for Kamailio which saw a coverage decrease at stack sizes 1 and 4, and OpenSSH which exhibited higher coverage at stack size 32.

Targets LightFTP and DNSmasq do not see much change in coverage between the different stack sizes. Exim shows a slight coverage drop at stack sizes 1 and 4, and a small coverage increase at stack sizes 16 and 32. OpenSSH sees lower performance at stack 1, similar performance to baseline at stack 4 and 16, and increased coverage at stack 32. For Kamailio, stack 16 and stack 32 performed best, albeit still lower than baseline performance, while stack 1 and 4 saw a significant decrease in performance for this target. Since coverage was generally lower than the baseline, Kamailio might experience a benefit from mixed stacking of region and byte mutations, rather than applying region mutations in isolation.

In terms of seed size, smaller stack sizes for region mutations resulted in shorter average seed lengths across all targets, with stack sizes 1 and 4 seeing an average decrease of 65.4% and 48.6% respectively compared to the baseline. This demonstrates how applying multiple region mutation operators to a single seed contributes to overall seed length growth.

Additionally, OpenSSH and Kamailio have much larger initial seeds compared to the other targets, and both tended to perform better with larger stack sizes. This is consistent with findings from [57], which observed that larger seeds tend to benefit more from larger stack sizes.

4.2.5 Addition of *Delete* Operator

With the integration of the new *delete* region mutation operator, the results showed that the majority of targets achieved performance similar to the baseline. Kamailio was the most impacted, with median branch coverage decreasing by 60 branches, however, this difference was not statistically significant.

For the Exim target, median branch coverage increased by 31.5 branches. However, the corresponding p-value was 0.364, indicating that the difference was not statistically significant. Overall, adding the *delete* operator alone does not significantly affect AFLNET's performance in its original configuration.

4.3 AFLNET-TAMS

We run AFLNET and AFLNET-TAMS with the following options: state-awareness is enabled, the favor algorithm is used for state and seed selection, the deterministic stage disabled, code feedback only, and no memory limit. Listing 4.1 shows an example of the command used for LightFTP experiments. The protocol specific options change as required, while the general options remain the same across targets. However, the Exim and Kamailio targets require a longer server initialization waiting time (option -D). We use 20,000 and 50,000 microseconds respectively.

Listing 4.1: Example AFLNET command used

```
afl-fuzz -d -i /home/ubuntu/experiments/in-ftp -x
    /home/ubuntu/experiments/ftp.dict -o outdir -N
    tcp://127.0.0.1/2200 -m none -P FTP -D 10000 -q 3 -s 3 -E -K -R
    -c /home/ubuntu/experiments/ftpclean ./fftp fftp.conf 2200
```

4.3.1 Development

In AFLNET, the core fuzzing program is contained within *afl-fuzz.c* file. To smoothly integrate our new mutation strategy into the existing setup, it was implemented by making modifications to this code.

Random Sampling

Since TS and LinTS require random sampling, the GNU Scientific Library (GSL) [59] was used to achieve this in C. Since distribution sampling requires randomness, we must initialize an Random Number Generator (RNG) instance for GSL to use with the *gsl_rng_alloc* function. We use the default

RNG type *gsl_rng_mt19937* [60] and the RNG is seeded with the same constant each time for deterministic MAB behaviour to improve experiment consistency. The other sources of randomness in AFLNET result in variation across experiment trials.

The TS agent requires a Beta distribution, while the LinTS agent requires a Gaussian distribution. GSL's *gsl_ran_gaussian* function was used for Gaussian distribution random sampling, while their *gsl_ran_beta* function was used for the Beta samples. As inputs, both functions accept the RNG instance and the Gaussian function accepts the standard deviation, whereas the Beta distribution accepts parameters α and β. The return values are variates with mean zero. To produce a sample centered on a specific mean, the desired mean is simply added to the zero-centered variate returned by the GSL function.

Thompson Sampling Mutation Type Selector

AFLNET uses a function called *save_if_interesting* to determine whether a seed is "interesting" and should be saved to the seed corpus. The AFLNET-TAMS implementation makes use of this function as the reward signal to the MAB agents, in particular it uses the variable called *hnb* (short for "has new bits"). A new edge in the bitmap is represented by $hnb = 2$ and a different hit count on an edge is where $hnb = 1$. Using the variable allows differentiation between these two types of "interesting" test cases.

Algorithm 3 TAMS Mutation Type Selection Pseudocode

```
 1: for each mutation on the current seed do
 2:     mut_type ← TSMAB.choose()                              ▷ Byte or Region
 3:     if mut_type = Byte then
 4:         stack_size ← 2^(1+RAND(7))                          ▷ 2, 4, ..., 128
 5:         for i ← 1 to stack_size do
 6:             Select and apply byte mutation operator
 7:         end for
 8:     else if mut_type = Region then
 9:         stack_size ← randomly selected from {1, 2, 4}
10:         for i ← 1 to stack_size do
11:             Select and apply region mutation operator
12:         end for
13:     end if
14:     reward ← 1 if new edge coverage, else 0
15:     TSMAB.update(mut_type, reward)
16: end for
```

Algorithm 3 illustrates the implementation of the mutation type selection into AFLNET. The TS agent decides whether to perform a byte or region mutation. For byte mutations, the stack size and random selection remains unchanged from AFLNET's original implementation, apart from separating region mutations out from them. For region mutations, the stack size is limited to 1, 2, or 4 stacks chosen randomly.

The type selector MAB was implemented as a C structure. It contains members that maintain the MAB state: an array storing the number of times each arm has been selected, and a second array storing the accumulated rewards for each arm. The type selector structure also stores the number of arms in the MAB and a pointer to the GSL RNG used for sampling.

Linear Thompson Sampling Region Mutation Selector

Algorithm 4 Region Mutation Operator Selection Pseudocode

1: **for** each mutation on the current seed **do**
2: $mut_type \leftarrow UR(21)$ ▷ 19% region mutation
3: **if** $mut_type \leq 16$ **then** ▷ Byte
4: $stack_size \leftarrow 2^{1+RAND(7)}$ ▷ 2, 4, . . . , 128
5: **for** $i \leftarrow 1$ to $stack_size$ **do**
6: Select and apply byte mutation operator
7: **end for**
8: **else** ▷ Region
9: $stack_size \leftarrow$ randomly selected from $\{1, 2, 4\}$
10: **for** $i \leftarrow 1$ to $stack_size$ **do**
11: $region_op \leftarrow LinTSMAB.choose(seed_len)$
12: Apply region mutation operator $region_op$
13: **end for**
14: **end if**
15: $reward \leftarrow 1$ if new edge coverage, else 0
16: $LinTSMAB.update(region_op, seed_len, reward)$
17: **end for**

Algorithm 4 shows how the region mutation operator selector is integrated into the AFLNET code. The default 19% probability of performing a region mutation is maintained, while the LinTS agent selects each region mutation operator to apply.

In the original AFLNET implementation, a uniformly distributed random number from 0 to 20 (*UR(21)*) is generated to select one of the 20 muta-

tion operators (*delete_bytes* weighted twice as heavy). To preserve the same mutation proportions, we retain this setup even though the mutation process is now split into byte and region mutation types. Values from 0 to 16 correspond to performing a stacked byte mutation, while values from 17 to 20 mean performing a stacked region mutation.

Some simplifications were made to Algorithm 2 (Section 2.3.2) for our specific implementation. Since there is only one element in the context vector in our TAMS design, $x(t)$ can be treated as a scalar. This means that transpose operations have no effect on it. Furthermore, the dimension parameter d becomes 1, reducing other vectors and matrices to scalar as well. The algorithm becomes as shown in Algorithm 5. Overall, this simplification allows us to avoid doing expensive matrix inversion and matrix multiplication calculations during fuzzing, reducing computational overhead.

Similar to the mutation type selector, the LinTS region operator selector is also implemented as a structure in C with members for number of arms, RNG instance. The members which track the internal state of the MAB are B, f, and u from Algorithm 5, implemented as arrays where each index is associated with an arm.

Algorithm 5 Linear Thompson Sampling with one-dimensional context vector and N arms, modified from [41].

Init: For each arm $a = 1, \ldots, N$: $B_a \leftarrow 1$, $\hat{\mu}_a \leftarrow 0$, $f_a \leftarrow 0$
for $t = 1, 2, \ldots$ **do**
  Observe scalar context $x(t)$
  for each arm $a = 1, \ldots, N$ **do**
    Sample $\theta_a(t) \sim \mathcal{N}(\hat{\mu}_a,\ \nu^2/B_a)$
    Score $s_a(t) \leftarrow x(t)\theta_a(t)$
  end for
  Play arm $a(t) \leftarrow \arg\max_a s_a(t)$ and observe reward r_t
  $B_{a(t)} \leftarrow B_{a(t)} + x(t)^2$
  $f_{a(t)} \leftarrow f_{a(t)} + r_t\, x(t)$
  $\hat{\mu}_{a(t)} \leftarrow f_{a(t)}/B_{a(t)}$
end for

4.3.2 Tuning Experiments

Tuning experiments were conducted to improve the performance of AFLNET-TAMS, with ten trials being run for each. The two MAB agents were tested individually in different experimental setups to isolate their impact and tune

the results. For brevity, we introduce short names to refer to the different configurations in these experiments. The version of the fuzzer using only the TAMS type selector MAB is referred to as TAMS-Type, short for TAMS Type Selector. Similarly, the version with only the region mutation operator selector MAB present is called TAMS-RegOp for TAMS Region Operator Selector. The version with both MAB active is referenced as TAMS-Combined, while all versions overall are referenced under TAMS as an umbrella term.

Reward Signal: "Interesting" Test Case Types

Initial experiments with TAMS-Type resulted in a high proportion of region mutations for most targets. These results can be seen in Table 4.2. This result was contrary to the expected behaviour for most targets, since the preliminary experiments showed that most targets benefitted from a reduced proportion of region mutations. Additionally, coverage for LightFTP and Kamailio was noticeably decreased.

Table 4.2: TAMS-Type Branch Coverage Results using both "Interesting" Types as Reward.

Target	Min	Max	Med	Mean	p-value	Avg % Region
LightFTP	341	360	342.5	344.8	0.1710	53.61
DNSmasq	1113	1128	1115	1115.6	0.0667	42.89
Exim	2841	2976	2931	2922.8	0.7052	37.08
OpenSSH	3341	3407	3372.5	3374.4	0.1399	9.61
Kamailio	9187	9485	9336.5	9340.4	0.0173	22.32

The results of Section 3.1.1 showed region mutations to be surprisingly high in triggering "interesting" test cases. In particular, the *duplicate* operator had the highest success rate in this regard for all but one target. These results can explain why high percentages of region mutations are seen in targets such as LightFTP and DNSmasq.

Suspecting that new hit count "interesting" test cases are leading to this behaviour, we investigated the success rates for each mutation operator in triggering $hnb = 1$ (new hit count) and $hnb = 2$ (new edge) separately. Heatmaps of the success rates for $hnb = 1$ and $hnb = 2$ are shown in Figure 4.2 and Figure 4.3 respectively. Darker red indicates a higher success rate, while lighter yellow indicates a lower success rate. This data is based on ten experiment runs of AFLNet with its original mutation strategy. Success rates were nor-

malized to the targets to highlight the relative performance of operators within each target. The operators with the top five success rates are indicated with numbers showing their rank. DNSmasq and Kamailio do not have any dictionary extras in their *ProFuzzBench* setups, therefore, *overwrite_bytes_extra* and *insert_extra* are not used.

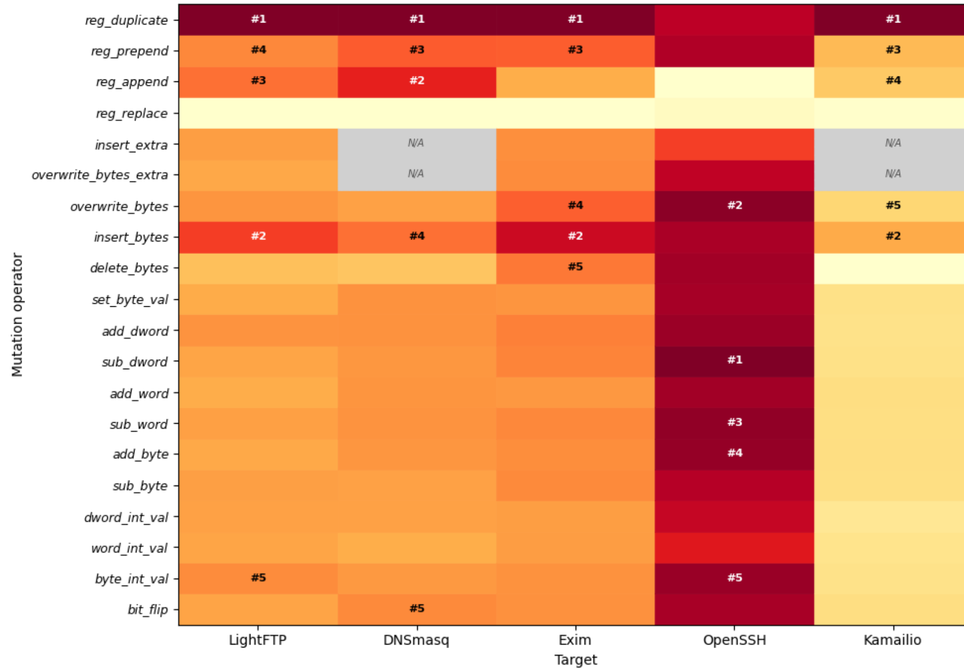

Figure 4.2: Success Rates of Each Mutation Operator at Triggering New Hit Counts (normalized to the targets).

Table 4.3: Average Percentage of "Interesting" Test Cases Types across Targets.

Target	Avg % new hit count	Avg % new edge
LightFTP	84.45	15.55
DNSmasq	90.84	9.16
Exim	84.02	15.98
OpenSSH	72.13	27.87
Kamailio	76.36	23.64

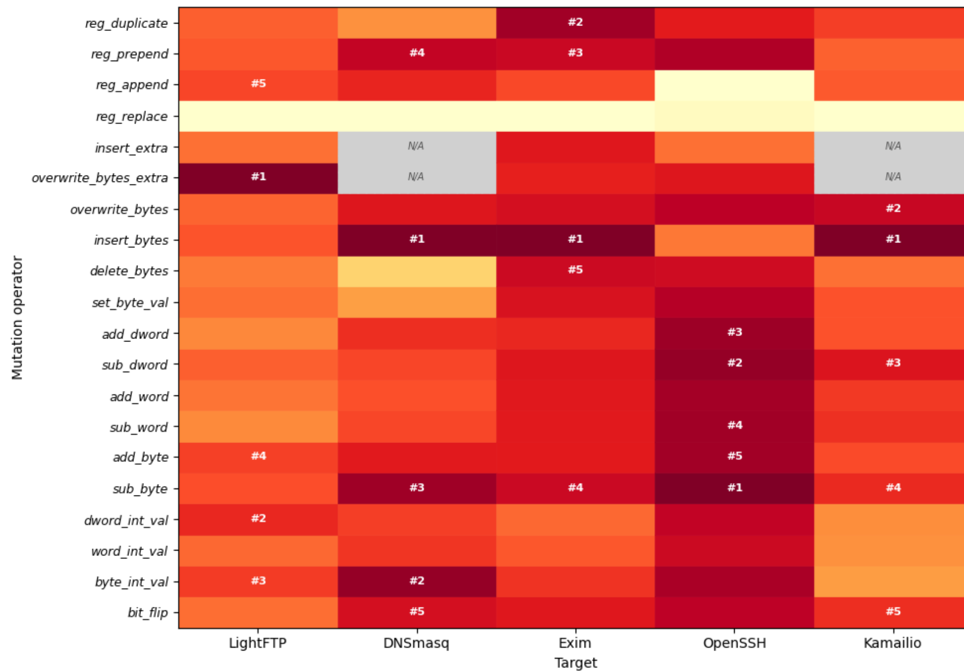

Figure 4.3: Success Rates of Each Mutation Operator at Triggering New Edges (normalized to the targets).

Additionally, we calculated the percentage of new edge versus new hit count "interesting" test cases for each target which is shown in Table 4.3. Immediately, it is apparent that new edges are much rarer than new hit counts which means treating both as equal reward heavily weights new hit count.

Comparing new hit count to new edge success rates, the performance of region mutations, and in particular the *duplicate* operator, drop significantly. This confirms that region mutations are particularly adept at triggering this type of "interesting" test case which leads to inflated region mutation usage in our TAMS-Type experiments. DNSmasq and LightFTP have the highest percentage of new hit count instances and also have the highest percentages of region mutations in these experiments.

The success rates of new edge and new hit count types differ substantially, particularly for region mutations. When both types are included as rewards, the large number of new hit count instances dominates the reward signal. This leads to undesirable outcomes such as decreased coverage and excessive region mutation usage in targets like LightFTP and DNSmasq, which previously showed some improved performance with fewer region mutations

in Section 4.2.3. Unlike the new edge type, the new hit count type has a less direct relationship to discovering new code coverage, making its empirical value less clear. Based on these observations, we decided to use only new edge "interesting" test cases as the reward signal for the MAB moving forward.

Exploration Parameter for Linear Thompson Sampling

Initial experiments with TAMS-RegOp showed very little change in region operator selection. Each operator was selected about 20% of the time (with five region operator present), making it akin to random selection. Since the reward signal is so sparse, and since mutation operators are stacked together, the MAB stays stuck in the exploration phase.

The exploration parameter v seen in Algorithm 2 can be decreased to lower exploration, thereby increasing exploitation. With lower variance in sampling, randomness is decreased and the MAB's decisions are more based on the historic performance of the operators. The values tested were: 1.0, 0.25, 0.1, 0.05, 0.001. The results are shown in Table 4.4, with the highest median for each target shown in bold.

Table 4.4: Median Coverage Results for Targets using Different v Values in TAMS-RegOp.

v	LightFTP	DNSmasq	Exim	OpenSSH	Kamailio
1	345.5	1115	2910	3359.5	9098
0.25	344	**1116**	2930	3359.5	9279
0.1	**349.5**	**1116**	2947.5	**3374.5**	**9300.5**
0.05	345	**1116**	2927	3366	9165
0.001	344	**1116**	**2957.5**	3370	9272

LightFTP, OpenSSH, and Kamailio achieved the best performance with v at 0.1. Each v experiment for DNSmasq resulted in the same median coverage of 1116, except for $v = 1.0$ which was slightly lower at 1115. Exim's best result was with $v = 0.001$, although $v = 0.1$ was the second highest. Exim might need more exploitation since it has the slowest execution speed among the targets, meaning the MAB has less data samples to work with. Overall considering these results, $v = 0.1$ is selected as the standard configuration which performs well for most targets.

Stack Size for Region Mutations

From the preliminary experiments, we saw how the region stack size can impact overall coverage obtained on a target during fuzzing. In both TAMS-Type and TAMS-RegOp, various stack sizes for region mutations were tested as shown in Table 4.5. We directly compared experiments where other parameters were kept consistent while the only difference was the maximum stack size for region mutations. For TAMS-RegOp, a consistent value of $v = 0.1$ was used for standardization. Another experimental variable was whether the new *delete* operator was included in the pool of region mutation operators. For TAMS-Type, the *delete* operator was not included in these experiments since its performance tends to be lower than the other region mutations which can lead to region mutations being selected less frequently. On the other hand, for TAMS-RegOp, the *delete* operator is included since the MAB is able to select among the region mutation operators, mitigating any negative impact that this mutation operator could have.

Table 4.5: Median Coverage Results for Targets across Different Stack Sizes in TAMS Configurations.

Config	Max Stack	Lightftp	DNSmasq	Exim	OpenSSH	Kamailio
TAMS-Type	4	**357.5**	1115	**2959.5**	3367	**9577**
TAMS-Type	8	348.5	**1116**	2903	3358.5	9223
TAMS-Type	16	346.5	**1116**	2937.5	3377.5	9327.5
TAMS-RegOp	4	349.5	**1116**	2947.5	3374.5	9300.5
TAMS-RegOp	8	346.5	1115.5	2938.5	**3382.5**	9348.5

The stack sizes tested were: {1,2,4}, {2,4,8}, and {2,4,8,16}. For simplicity, we refer to each configuration as its maximum stack value, therefore, stack sizes {1,2,4} is referred to as max stack 4 or stack 4. The stack 16 configuration was only conducted on TAMS-Type.

For LightFTP, results generally showed that the target performed better with a maximum stack size of 4 compared to 8 or 16. The highest performing version (2.9% increase compared to baseline) was with a maximum stack of 4. The other stack sizes resulted in similar outcomes (median branch coverage ranged from 346.5 to 349.5) with the lower stack experiments having slightly higher performance.

Meanwhile, DNSmasq had very similar results with the different stack

sizes. Slightly higher performance was observed when using larger stack sizes. In TAMS-Type, stack 8 and stack 16 had higher median coverage than stack 4. Whereas, with TAMS-RegOp, stack sizes 4 and 8 only had a 0.5 difference in branch coverage with 1116 and 1115.5 respectively.

Exim achieved higher coverage with stack size 4 than with the other stack sizes for both TAMS-Type and TAMS-RegOp. In TAMS-Type, stack 4 at 2959.5 branches saw much higher coverage compared to stack 8 at 2903 branches, while stack 16 at 2937.5 branches performed better than stack 8.

OpenSSH consistently performed best at higher stack sizes, with TAMS-Type attaining its highest coverage at stack 16 and TAMS-RegOp at stack 8. This result is consistent with the preliminary stack size experiments, where OpenSSH achieved the best coverage with a fixed stack of 32 (the largest value tested).

In TAMS-Type, Kamailio had the best results with stack 4. In TAMS-RegOp, the highest median coverage was achieved with stack 8, although stack 4 was fairly close behind.

Overall, the optimal stack size for region mutations varied across targets. In general, the maximum stack size of 4 for was most performant across the various targets, therefore, we choose this configuration for our final design.

4.4 Verification

The verification phase involved using Python code integrated in Jupyter notebooks to analyze the logs produced by AFLNET-TAMS. Three separate verifications were conducted. The first focused on the type selector MAB, the second on the region operator selector MAB, and the lastly on the *delete* operator. Each experiment consisted of running *ProFuzzBench* for ten minutes against the LightFTP target. Since these experiments focus on verification, this short experiment is sufficient since thousands of MAB decisions and *delete* operator uses are conducted during it.

4.4.1 Thompson Sampling Mutation Type Selector

The first major addition of our TAMS design is the TS agent which selects between the two mutation types: region and byte. An additional CSV log file was added to confirm the TS agent was behaving as intended. Each row in the file represents one MAB decision. The columns include the time stamp, number of uses and successes for each mutation type, as well as the distribution samples for each arm, and the selected type. In the actual TAMS runs, the

additional log tracking the MAB status was removed while keeping the MAB functions otherwise the same.

To verify the Beta distribution samples used by TS, we apply the Probability Integral Transform (PIT). For a continuous random variable X with a Cumulative Distribution Function (CDF) $F(x)$, the resulting distribution $Y = F(X)$ is uniformly distributed over $[0, 1]$ [61]. We computed $F(x)$ for each sample using SciPy's Beta CDF [62] with the α and β parameters being derived from the logged use and success counts. These computations are performed for each row of the CSV log file using *pandas* DataFrames [63].

The resulting CDF values were analyzed and determined to be consistent with a uniform distribution. The mean values for the byte and region arms were 0.5022 and 0.4958, respectively, which closely match the expected 0.5. The proportions of CDF values below and above 0.5 are 0.4997 and 0.5003 for the byte arm, and 0.5053 and 0.4947 for the region arm which is consistent with the expected 0.5. Furthermore, the proportion of values above and below 0.95 closely matched the expected 5% above and 95% below, providing additional evidence of uniformity. These results indicate the Beta samples align with their associated distributions.

Using *pandas* DataFrames [63], each row is analyzed to verify that the arm with the highest random variate is selected as the mutation type. Additionally, for every subsequent row, it is verified that the chosen type's uses counter is incremented, and that the successes counter is incremented when applicable. Expected values constructed from the previous row are compared with the observed results, and all values match, confirming the TS selection behaviour.

4.4.2 Linear Thompson Sampling Operator Selector

Similar to the first MAB, a log file was added to verify the LinTS agent behaviour. This file tracked the details of each MAB decision in a CSV file where each row represented a region operator selection made by the MAB. The logged information includes the distribution parameters for each arm (B, f, u), the sample from each distribution, and the seed length (context).

To verify the MAB behaviour, first, the sampled values are analyzed to determine whether they fit what is expected based on their distributions. Since the distributions are Gaussian, the *68-95-99.7* rule can be applied which states that approximately 68% of samples fall within one standard deviation of the mean, 95% of samples are within two standard deviations, and 99.7% of samples are within three standard deviations [64].

To obtain the number of standard deviations for each sample, the standard score (also known as z-score) is calculated for each arm's sample which

represents how many standard deviations a sample is from the mean [65]. The z-score is computed for each sample as a new column and the *68-95-99.7* percentages are calculated. Among the arms, percentages within one standard deviation ranged from 67.3% to 67.9%, for two standard deviations, from 95.0% to 95.7%, and for three standard deviations, from 99.6% to 99.7%. These values align closely with the expected ranges, confirming that the samples follow their distributions.

The arm with the highest score is verified to be selected each time using *pandas*, as was done with the type selector MAB. The parameter updates are calculated for each row after the first using the values from the previous row to ensure correctness. In particular, each arm involved in the previous mutation should have its B, f, and $\hat{\mu}$ updated as shown in Algorithm 5. To predict the values for the next row, the NumPy *where* function is used to modify the values depending on conditions such as whether the test case was "interesting" and whether it was used in the previous region mutation [66]. The calculated next row is compared to the actual logged row to verify that they match. All rows matched successfully, confirming the MAB was behaving as intended.

4.4.3 *Delete* Operator

To verify that the new *delete* region mutation operator functions as expected, a version of the fuzzer was created which logs each *delete* operation. This is implemented as a text file which displays the full M_2 subsequence, lists the index of the message chosen to be deleted, and shows the M_2 sequence after removing the message. Since the messages' formats end up distorted during fuzzing, this verification was done manually. After running this version of the fuzzer on the *ProFuzzBench* LightFTP target for ten minutes, the *delete* log was inspected to ensure that the correct message was removed and the order of the remaining messages remains unchanged. Since manual verification was used, only the first ten complete *delete* operations were inspected.

4.5 Validation

To validate the aim of this research, the code coverage results of our AFLNET-TAMS are compared to the original AFLNET. This evaluation focuses on comparing median coverage which limits the influence of outlier runs on overall results. Three different versions of our design test are assessed: TAMS-Type, TAMS-RegOp, and TAMS-Combined. Additionally, the Mann-Whitney U test is applied to determine if the results are statistically significant.

For the baseline AFLNET performance, the coverage results from Section 4.2.2 are used. To strengthen the statistical significance and lower the effect of randomness on our results, an additional ten baseline fuzzing runs were conducted, bringing the total to twenty runs. Furthermore, twenty fuzzing sessions are conducted for each validation configuration compared to the preliminary and tuning experiments which only contained ten runs. Since the addition of the new *delete* region operator can affect results, experiments are performed with and without its inclusion. The validation results for each target are presented in Table 4.6, Table 4.7, Table 4.8, Table 4.9, and Table 4.10.

Table 4.6: Branch Coverage Results for TAMS Validation on the LightFTP Target.

Experiment	Delete	Min	Max	Median	Mean	p-value	% Diff.
Baseline	False	340	361	345	347.8	–	–
TAMS-Type	True	340	369	345.5	348.5	0.7650	+0.145
TAMS-Type	False	340	372	**348.5**	350.9	0.4557	+1.014
TAMS-RegOp	True	341	378	**348.5**	353.45	0.0799	+1.014
TAMS-RegOp	False	340	365	345.5	347.85	0.7753	+0.145
TAMS-Comb	True	340	363	346	350.0	0.5060	+0.290
TAMS-Comb	False	340	362	344.5	346.6	0.6937	-0.145

Table 4.7: Branch Coverage Results for TAMS Validation on the DNSmasq Target.

Experiment	Delete	Min	Max	Median	Mean	p-value	% Diff.
Baseline	False	1113	1130	1115	1117.1	–	–
TAMS-Type	True	1111	1129	1115	1117.0	0.7299	+0.000
TAMS-Type	False	1113	1128	1115	1115.85	1.0000	+0.000
TAMS-RegOp	True	1113	1128	**1116**	1118.65	0.2940	+0.090
TAMS-RegOp	False	1111	1128	1115	1116.7	0.8350	+0.000
TAMS-Comb	True	1113	1131	**1116**	1119.2	0.1778	+0.090
TAMS-Comb	False	1112	1128	1115	1117.15	0.7282	+0.000

Table 4.8: Branch Coverage Results for TAMS Validation on the Exim Target.

Experiment	Delete	Min	Max	Median	Mean	p-value	% Diff.
Baseline	False	2884	2977	2934.5	2931.2	–	–
TAMS-Type	True	2855	2966	2928.0	2923.45	0.4733	-0.222
TAMS-Type	False	2870	2972	**2945.5**	2939.75	0.2614	+0.375
TAMS-RegOp	True	2824	2979	2928.5	2930.25	0.8817	-0.204
TAMS-RegOp	False	2865	2961	2938.5	2929.25	0.9138	+0.136
TAMS-Comb	True	2846	2981	2945.0	2931.75	0.6167	+0.358
TAMS-Comb	False	2837	2972	2932.5	2920.6	0.6358	-0.068

Table 4.9: Branch Coverage Results for TAMS Validation on the OpenSSH Target.

Experiment	Delete	Min	Max	Median	Mean	p-value	% Diff.
Baseline	False	3326	3385	3353	3354.35	–	–
TAMS-Type	True	3336	3408	3374.0	3374.2	0.0019	+0.626
TAMS-Type	False	3329	3403	3365.5	3364.05	0.0881	+0.373
TAMS-RegOp	True	3330	3399	3372.5	3365.9	0.0600	+0.582
TAMS-RegOp	False	3342	3408	**3374.5**	3374.0	0.0039	+0.641
TAMS-Comb	True	3332	3409	3373.5	3373.05	0.0065	+0.611
TAMS-Comb	False	3337	3391	3369	3365.2	0.102	+0.477

Table 4.10: Branch Coverage Results for TAMS Validation on the Kamailio Target.

Experiment	Delete	Min	Max	Median	Mean	p-value	% Diff.
Baseline	False	9225	9930	9471.0	9530.45	–	–
TAMS-Type	True	9054	9603	9361.5	9354.25	0.0051	-1.156
TAMS-Type	False	9159	9827	9450.5	9463.05	0.3169	-0.216
TAMS-RegOp	True	9059	9587	9294.5	9310.95	0.0003	-1.864
TAMS-RegOp	False	9197	9846	**9496.5**	9482.6	0.6456	+0.269
TAMS-Comb	True	9086	9649	9425.5	9398.55	0.0679	-0.480
TAMS-Comb	False	8958	9533	9310	9285.4	0.000104	-1.700

4.5.1 TAMS Type Selector

The TAMS-Type validation experiments did not show a significant change in coverage performance for most targets, except for OpenSSH (Table 4.9) which improved in coverage. In particular, with TAMS-Type (*delete* present) OpenSSH achieved 0.626% more branch coverage compared to baseline with a p-value of 0.0019, indicating a statistically significant improvement. The version without *delete* also saw a coverage improvement on this target, however, it fell just short of statistical significance ($p = 0.0881$). The targets LightFTP, DNSmasq, and Exim, all obtained similar results to the baseline with p-values well above the significance range. Kamailio's performance was also very similar with the *delete* operator absent, however, when *delete* was included, the median coverage dropped by 1.156%.

4.5.2 TAMS Region Operator Selector

The TAMS-RegOp validation experiments produced very similar results to TAMS-Type, with coverage remaining comparable to baseline in most cases. The OpenSSH target once again saw median coverage improvements, increasing by 0.641% when the *delete* operator absent. With the *delete* operator included, coverage for this target also increased but narrowly missed the p-value significance mark at 0.06. LightFTP, DNSmasq, and Exim obtained similar performance to baseline results, however, LightFTP's coverage increased by 1.014% with TAMS-RegOp (*delete* present) but this result did not cross the p-value significance threshold. Kamailio again experienced a significant coverage decrease when *delete* was included, with the other configuration performing similar to baseline.

4.5.3 TAMS Combined Design

The TAMS-Comb experiments resulted in no statistically significant coverage change for most targets. In particular for LightFTP, DNSmasq, and Exim, coverage remained comparable to the baseline regardless of whether the *delete* operator was enabled. For OpenSSH, coverage improved when the *delete* operator was included. While the configuration without *delete* also improved coverage for OpenSSH, it did not meet the significance threshold ($p = 0.102$). For Kamailio, the configuration without *delete* resulted in a statistically significant decrease in coverage, while the configuration with *delete* experienced a smaller decrease which was not statistically significant.

4.6 Discussion

This section examines the reasons behind the results observed in Section 4.5. While the validation results showed improvements for the OpenSSH, the remaining targets achieved comparable performance to baseline with one target having decreased performance in some cases. We present reasons for these differences.

4.6.1 Mutation Type Selector

For most targets, the mutation type selector MAB decisions resulted in fewer region mutations being used compared to the original 19%, as shown in Table 4.11. The reason that the region and byte percentages in this table do not sum exactly to 100% is that there are cases where AFLNET does not perform a mutation due to its current implementation. In the current implementation, the mutation operator is selected before verifying that the current seed satisfies the requirements of that operator. This check is instead performed after the operator has already been selected, and if the seed does not meet the conditions, execution breaks out of that branch without applying the mutation operator. Not all operators have requirements, but some do. For example, the *delete_bytes* operator is not applied when the seed is below a minimum length, while the *duplicate* operator is skipped when the seed exceeds a certain length. When the mutation stack size is very small, such as one or two, these skipped mutations can allow the seed to pass through the mutation stage without any mutation being applied, resulting in this edge case.

Table 4.11: Mutation Type Statistics for TAMS-Type (*delete* disabled).

	LightFTP	**DNSmasq**	**Exim**	**OpenSSH**	**Kamailio**
Region Percentage (%)	12.636	9.266	23.287	8.520	3.111
Byte Percentage (%)	87.361	90.536	76.699	91.466	96.682
Region Success Rate (%)	0.0109	0.0034	0.0626	0.0219	0.1780
Byte Success Rate (%)	0.0193	0.0107	0.0726	0.0285	0.2895
Region to Byte Absolute Difference (%)	0.0084	0.0073	0.0099	0.0066	0.1115
Region to Byte Success Rate Increase (%)	77.686	217.407	15.820	30.214	62.611

Across the targets, Kamailio experienced the greatest reduction in region mutations with runs having an average of 3.1% region mutations. This obser-

vation was surprising since our preliminary experiments showed that Kamailio performed better with 19% and 38% region mutations than it did with 9.5% region mutations. To delve deeper into these results, we examine the success rates of region and byte mutations shown in Table 4.11.

The region to byte success rate increases in Table 4.11 indicate the percentage change in success rate from region to byte mutations. This relative difference ranged from +15.8% for Exim to +217.4% for DNSmasq. Despite DNSmasq having the largest relative difference between region and byte mutation performance, it still had a region percentage of 9.3%, higher than two other targets tested. Meanwhile, Kamailio had a significantly lower percentage of region mutations, however, the relative difference from region to byte was lower than two other targets at a 62.6% increase. When we compare these results to DNSmasq which had a significantly higher percentage of region mutations, this appears counterintuitive given region mutations were much less effective than byte on this target.

Our mutation type success rates are not exactly equivalent to their posterior distribution mean rewards. However, the success rate equation is equivalent to $\frac{s}{t} \times 100$ (where s represents number of successes and t represents number of total uses for that arm), while the posterior mean is equivalent to $\frac{s+1}{t+2}$. As s and t grow into the thousands as observed in our fuzzing, the +1 and +2 terms have little impact, so we approximately compare these in our conceptualization. Since TS chooses the arm with the highest posterior sample, the probability of selecting the weaker arm (region mutation in the targets tested) depends on the overlap between the arms' posterior distributions. This overlap is dictated by the difference in the arms' means and the posteriors' variance which is based on the total success and fail numbers. The mutation type selector's decisions are therefore not based on relative performance alone, but rather on the posterior samples derived from each arm's mean and variance.

The differing relationship between region relative performance and its selection rate appears to be driven by the reward density of the target. Targets such as DNSmasq, where fewer new edges are found, have sparser rewards. When there are fewer successful cases, the variance in sampling remains high, resulting in more explorative MAB decisions. Whereas with more rewards, the MAB variance is lower, meaning more exploitative decision-making. Kamailio's relatively high success rates also result in significantly larger absolute differences between the success rates of region and byte mutations at 0.1115%, as shown in Table 4.11. With a larger gap in mean between region and byte mutations, the posterior distributions have less overlap, leading to more decisive behaviour and the weaker arm being selected less frequently. On the

other hand, DNSmasq had the lowest overall success rates and a much lower absolute difference at 0.0073%, meaning the smaller gap in mean combined with higher sampling variance leads to region mutations being selected more frequently despite their lower relative performance.

Exim was the only target which saw an increase in the percentage of region mutations (from 19.1% to 23.3%). This is consistent with it having both the smallest relative performance gap between region and byte mutations (15.8%) and moderate overall success rates (not extreme enough to cause highly exploitative nor explorative behaviour). Similarly, OpenSSH had moderate success rates resulting in behaviour that was not too exploitative nor explorative leading to a region percentage of 8.5% with a relative difference of 30.2%. Meanwhile, LightFTP had a very low average success rate, and we speculate that the resulting higher exploration led to region mutations being selected more frequently than would be ideal, at 12.6% with a relative difference of 77.7%.

Although the optimal proportion of region mutations is not known for each target, preliminary experiments provide a useful reference point. For instance, LightFTP achieved higher median coverage when the region mutation probability was set to 9.5% in the preliminary experiments, compared to the approximately 12.6% observed under TAMS-Type (*delete* disabled). This suggests that the mutation type selector's decisions may not have resulted in the most effective balance of region and byte mutations for this target. Since the amount of exploration to exploitation varies across the targets, this directly influences the proportion of region mutations, leading to differing effectiveness.

Furthermore, exploration tuning is particularly important in our application. In the classical stochastic MAB setting, each arm has a fixed reward distribution and the objective is to identify and repeatedly select the arm with the highest expected reward. In our setting, TS is used to adapt the selection of region and byte mutations based on their observed effectiveness during fuzzing. Exploration is a key factor in enabling region mutations to continue to be selected even when byte mutations exhibit higher reward rates. Overall, the mutation type selector demonstrated the ability to adjust the proportion of mutation types based on their observed performance, with individual target differences resulting in varying outcomes.

4.6.2 Region Operator Selector

The region operator selector increased coverage for OpenSSH, while no statistically significant coverage impact was observed for the remaining targets. To understand how the selector influenced mutation, we first examine how

frequently each region mutation operator was chosen. For this analysis, the selection percentage of each region mutation operator is calculated as the number of times the operator was selected divided by the total number of region operator selections. These values are shown in Table 4.12 for TAMS-RegOp (*delete* enabled), where the two highest percentages per target are shown in green, while the two lowest are in red.

Table 4.12: Average Region Mutation Operator Usage Percentages for TAMS-RegOp (*delete* enabled).

Operator	LightFTP	DNSmasq	Exim	OpenSSH	Kamailio
reg_replace	19.824%	19.883%	19.533%	17.109%	17.315%
reg_prepend	20.293%	20.121%	20.582%	17.650%	22.152%
reg_append	20.404%	20.047%	20.289%	21.693%	21.084%
reg_duplicate	19.857%	20.020%	19.655%	20.855%	20.394%
reg_delete	19.620%	19.929%	19.941%	22.693%	19.055%

Table 4.13: Average Region Mutation Operators Success Rates for TAMS-RegOp (*delete* enabled).

Operator	LightFTP	DNSmasq	Exim	OpenSSH	Kamailio
reg_replace	0.01103	0.00068	0.04695	0.01290	0.07967
reg_prepend	0.01503	0.00091	0.07408	0.01204	0.10974
reg_append	0.01446	0.00245	0.10185	0.02773	0.11540
reg_duplicate	0.00977	0.00246	0.08693	0.02154	0.10738
reg_delete	0.00948	0.00102	0.06499	0.02767	0.09929

Since five region mutation operators are available when *delete* is enabled, uniform random selection would result in approximately 20% usage for each operator. In our TAMS-RegOp experiments, deviations from this value indicate that the MAB has learned to favour certain operators over others.

To evaluate whether the MAB selections reflect operator effectiveness, we compare the selection percentages with the individual operator success rates shown in Table 4.13. For easy visual comparison between tables, the same colour scheme from Table 4.12 is maintained with the two highest success rates for a target highlighted in green and the lowest two in red. In general,

the expectation is that operators with higher success rates will be selected more frequently by the MAB which aims to maximize new edge discovery.

In comparing Table 4.12 and Table 4.13, the operators with the highest and lowest success rates correspond closely with those having the highest and lowest selection percentages for the targets LightFTP, OpenSSH, and Kamailio. This alignment suggests that the MAB was able to learn which region operators were more effective at triggering new edges.

A noticeable discrepancy is observed in operator selection and performance for the Exim and DNSmasq targets. In Exim's case, the *duplicate* operator has the second lowest selection percentage despite having the second highest success rate. Further inspection reveals that this result is influenced by a few outlier runs. The median selection percentage for this operator is 20.30%, showing that it was generally favoured, which better reflects its performance.

The *prepend* operator was selected 20.1% on average despite performing second lowest on DNSmasq. An explanation for this observation is likely the sparsity of rewards on this target. When new edge discovery is very rare, a successful test case substantially increases an operator's observed success rate. Since stacking operators leads to a reward attribution issue where we cannot determine which operator resulted in the test case's success, its performance can become inflated, particularly if the operator is chosen during the early stages of fuzzing when new edges are easier to discover. Overall, the LinTS operator selector generally identified the most effective region mutation operators in terms of new edge discovery for the different targets, though some variability across runs was present.

4.6.3 TAMS Combined Design

In our combined TAMS design, both the TS mutation type selector and the LinTS region operator selector are present. Overall, its performance largely mirrored that of TAMS-Type and TAMS-RegOp. OpenSSH showed improved coverage, while the other targets maintained coverage similar to the baseline, and Kamailio experienced slightly weaker performance.

Examining the percentage of region mutations in Table 4.14, the region proportion increased for all targets except Exim, which showed a similar but slightly lower percentage. We hypothesize that this increase occurs because the region operator selector improves the effectiveness of region mutations by favouring the selection of operators that are more likely to trigger new edges.

In TAMS-Comb, interactions occur between the two MAB components. In TAMS-RegOp, the percentage of region mutations is fixed at 19.1%. During the tuning experiments, the exploration parameter was tuned for each

Table 4.14: Mutation Type Statistics for TAMS-Combined (*delete* disabled).

	LightFTP	**DNSmasq**	**Exim**	**OpenSSH**	**Kamailio**
Region Percentage (%)	15.070	9.785	23.065	13.632	7.087
Byte Percentage (%)	84.923	90.016	76.925	86.357	92.712
Region Success Rate (%)	0.0117	0.0033	0.0655	0.0223	0.2703
Byte Success Rate (%)	0.0182	0.0105	0.0769	0.0270	0.3286
Region to Byte Absolute Difference (%)	0.0066	0.0072	0.0113	0.0048	0.0583
Region to Byte Success Rate Increase (%)	56.484	214.213	17.253	21.430	21.553

target under this setup. However, in TAMS-Comb the percentage of region mutations can vary substantially. For example, in Kamailio the proportion of region mutations drops to less than half of its original value. This significantly changes the number of region mutations performed and can affect exploitation, because the MAB has less information available to guide its decisions.

A notable difference from the other TAMS configurations is that Kamailio performed worse when the *delete* operator was present. This contrasts with the results of TAMS-Type and TAMS-RegOp, where Kamailio's performance dropped when *delete* was enabled but remained similar when *delete* was disabled. A possible contributing factor is that when the proportion of region mutations is higher, the *delete* operator becomes more useful because it can reduce overall seed length.

Table 4.15: Average Region Mutation Operator Usage Percentages for TAMS-Comb (*delete* enabled).

Operator	LightFTP	DNSmasq	Exim	OpenSSH	Kamailio
reg_replace	20.034%	19.756%	19.748%	16.872%	17.452%
reg_prepend	20.112%	19.948%	20.320%	16.645%	20.283%
reg_append	20.518%	20.227%	20.832%	21.702%	22.769%
reg_duplicate	19.468%	20.093%	19.517%	22.091%	20.398%
reg_delete	19.869%	19.977%	19.583%	22.690%	19.098%

Examining the breakdown of region operator usage and success rates in Table 4.15 and Table 4.16 reveals slightly different patterns compared to

Table 4.16: Average Region Mutation Operator Success Rates for TAMS-Comb (*delete* enabled).

Operator	LightFTP	DNSmasq	Exim	OpenSSH	Kamailio
reg_replace	0.01162	0.00259	0.04260	0.01477	0.14161
reg_prepend	0.01680	0.00329	0.07826	0.01265	0.17192
reg_append	0.01718	0.00622	0.08748	0.02733	0.22174
reg_duplicate	0.00772	0.00657	0.06843	0.02465	0.18749
reg_delete	0.00913	0.00436	0.06155	0.02930	0.15865

TAMS-RegOp. OpenSSH remains very similar, as two region operators are clearly less effective, which likely explains why this target benefits from our approach. In Kamailio, all region operators show increased success rates compared to TAMS-RegOp, however, this improvement does not translate into increased overall coverage. This finding suggests that optimizing the discovery of new edges may be less effective at improving overall coverage for this target than for the others. DNSmasq similarly had a notable increase in success rate of region mutations, with the highest performing operator going from 0.00246% to 0.00657%. This change also did not result in higher overall coverage. LightFTP and Exim showed little change in success rates.

4.6.4 Impact of TAMS on Seed Length and Execution Speed

To analyze the effect of TAMS on seed length and execution speed, we present Table 4.17 and Table 4.18. The data demonstrates that TAMS consistently results in much smaller average seed lengths compared to the baseline. On average, seeds were 52.2% smaller using TAMS, with most targets seeing a greater than 60% reduction in seed length. The most significant change in average seed length was observed for LightFTP, which saw an average decrease of 84.2% across the TAMS configurations. DNSmasq, Exim, and Kamailio experienced similar reductions in seed length percentage ranging from an average of 62.5% to 65.4%. The smallest change in seed length was seen in OpenSSH, decreasing by about 15.6%.

This change can mostly be attributed to fewer region mutations being stacked together since in the preliminary experiments on region stack size, lower stack sizes resulted in reduced seed lengths. The inclusion of the *delete* operator generally resulted in slightly smaller average seed sizes, although this is not true in all cases.

Table 4.17: Average Seed Length (bytes), Average Execution Speed, and Median Branch Coverage across Targets for Baseline and TAMS Configurations.

Configuration	Metric	LightFTP	DNSmasq	Exim	OpenSSH	Kamailio
Baseline	seed len	7416.27	156201.35	7166.84	40001.59	18166.13
	exec/s	5.455	5.515	2.803	23.553	4.299
	coverage	345	1115	2934.5	3353	9471
TAMS-Type (*delete* disabled)	seed len	1215.65	55690.34	2009.12	41853.41	5680.06
	exec/s	5.245	5.257	2.332	23.507	4.480
	coverage	348.5	1115	2945.5	3365.5	9450.5
TAMS-Type (*delete* enabled)	seed len	1508.17	40012.34	2771.22	31650.21	5993.84
	exec/s	5.571	6.570	2.523	23.464	4.408
	coverage	345.5	1115	2928	3374	9361.5
TAMS-RegOp (*delete* disabled)	seed len	1052.79	93199.19	2847.66	35757.16	8170.78
	exec/s	5.743	6.088	2.737	20.757	4.062
	coverage	345.5	1115	2938.5	3374.5	9496.5
TAMS-RegOp (*delete* enabled)	seed len	891.02	55377.22	3049.15	36200.43	6015.10
	exec/s	5.681	6.933	2.652	22.593	4.362
	coverage	348.5	1116	2928.5	3372.5	9294.5
TAMS-Comb (*delete* disabled)	seed len	1262.41	61346.93	2408.66	28614.84	5957.16
	exec/s	5.675	5.864	3.026	23.777	4.292
	coverage	344.5	1115	2932.5	3369	9310
TAMS-Comb (*delete* enabled)	seed len	1088.29	46264.97	2643.49	28608.96	5906.69
	exec/s	5.277	7.526	3.028	19.902	4.447
	coverage	346	1116	2945	3373.5	9425.5

Table 4.18: TAMS Percent Change in Average Seed Length Relative to Baseline.

Configuration	LightFTP	DNSmasq	Exim	OpenSSH	Kamailio
TAMS-Type (*delete* disabled)	-83.6%	-64.3%	-72.0%	+4.6%	-68.7%
TAMS-Type (*delete* enabled)	-79.7%	-74.4%	-61.3%	-20.9%	-67.0%
TAMS-RegOp (*delete* disabled)	-85.8%	-40.3%	-60.3%	-10.6%	-55.0%
TAMS-RegOp (*delete* enabled)	-88.0%	-64.5%	-57.5%	-9.5%	-66.9%
TAMS-Comb (*delete* disabled)	-83.0%	-60.7%	-66.4%	-28.5%	-67.2%
TAMS-Comb (*delete* enabled)	-85.3%	-70.4%	-63.1%	-28.5%	-67.5%
Average	**-84.2%**	**-62.5%**	**-63.4%**	**-15.6%**	**-65.4%**

The smaller average seed size did not consistently effect execution speed. Some TAMS configurations resulted in higher average execution speeds, while others resulted in lower speeds. LightFTP and DNSmasq tended to show slightly higher speed compared to the baseline, whereas Exim and OpenSSH generally had reduced speeds.

Furthermore, higher execution speeds do not appear to correlate with higher coverage. Despite OpenSSH executing fewer inputs per second in several configurations, it achieved better coverage. We hypothesize that test cases that trigger new coverage could take longer to execute compared to poorer quality fuzz. A test case that quickly causes an error in the server may terminate early which prevents it from reaching deeper regions of the program. Such a test case would likely execute faster than one that maintains a longer interaction with the server. As a result, execution speed can vary depending on the seed, and seeds that lead to new coverage may require longer execution times.

Similarly, seed size does not appear to have a strong relationship with coverage. Our TAMS model was able to achieve similar coverage to baseline with smaller seeds. Overall, the positioning and content of mutations likely matter more than the overall size of the seed. Long seeds do not necessary mean

slower execution speeds, and a well-crafted shorter seed may have a longer execution time due to the server executing more of its code and having more exchanges. Finally, the overhead introduced by the MAB components does not appear to have a significant impact on execution speed when compared to the baseline.

Appendix B presents images showing the selection of region operator for different seed lengths for TAMS-RegOp experiments. We examine these results to discuss whether our LinTS agent learned to apply the region mutations differently depending on seed length. To make these images, we divided each seed mutation into buckets depending on their seed length. We choose five bins, where each contains around 20% of the mutations.

For the *replace* operator, its selection increases with seed length for DNSmasq and Exim, but does not show a meaningful change for the other targets. For the *append* operator, selection decreases for larger seeds on Kamailio, but a clear pattern is not observed for other targets. As seed length grows, all targets see the *prepend* operator being used less frequently. The *duplicate* operator experiences lower selection as seed length increases for Exim, but has mixed results on other targets. For the *delete* operator, its selection increases with longer seeds for LightFTP, Exim, and Kamailio. Overall, some learned behaviour related to seed length is observed, although the differences are relatively small. Some desirable behaviour is observed, such as *delete* being used more often on longer seeds, and *prepend* being used less with longer seeds. This behaviour could contribute to regulating overall seed length.

4.7 Summary

This chapter presented the results of the preliminary experiments, and the development, verification, and validation of AFLNET-TAMS. The preliminary results showed that mutation operator performance varies across targets with some general trends. In particular, region mutations were found to have high rates of triggering new hit count type “interesting” test cases. Furthermore, we saw how most targets performed better with fewer region mutations compared to the default amount, although Kamailio experienced an overall coverage reduction with a lower proportion of region mutations. A fixed stack size of 4 for region mutations generally performed well across targets, whereas OpenSSH attained the highest coverage with larger stack sizes. Finally, the addition of the *delete* region mutation operator to the mutation operator pool generally did not impact coverage, except for Kamailio which experienced slightly decreased coverage and Exim which had slightly increased coverage.

Overall, these initial experiments gave us more insight into the impact of region mutations and highlighted individual target differences in parameter performance.

The experimental design and implementation details were described. Our experimental setup included four different machines accessed remotely, with each capable of running 20 Docker containers for *ProFuzzBench* experiments at a time. Our experiments included the addition of a mutation log to assist in the analysis of results. The implementation of the TS mutation type selector and the LinTS region operator selector within the *afl-fuzz.c* main program file was explained. The tuning experiments revealed that a maximum stack size of 4 and a LinTS exploration parameter v value of 0.1 performed well on most targets, therefore, this was selected as our AFLNET-TAMS default configuration.

The verification results confirmed that the two MAB agents and the *delete* operator behaved as intended. For each MAB, additional log files were created, the samples were checked to confirm that they matched the expected distributions, and the MAB parameters were verified to be updated correctly with the reward information. An additional log file was also added for the *delete* operator to confirm that the randomly selected message was removed while the remaining messages stayed in order and intact.

The validation experiments showed that AFLNET-TAMS generally achieved coverage comparable to the baseline AFLNET, with statistically significant improvements observed for OpenSSH. Three main configurations were evaluated: TAMS-Type (type selector MAB only), TAMS-RegOp (region operator selector MAB only), and TAMS-Comb (both selector MABs present). Ultimately, the results of each configuration were largely similar. Additionally, these experiments were conducted with and without the addition of the *delete* operator. Both results were similar, but generally showed a decrease in overall performance for Kamailio when the *delete* operator was present, except in the TAMS-Comb configuration.

The discussion section highlighted factors that help explain these outcomes. In particular, the differences in the rate of discovering "interesting" test cases influenced how the mutation type selector balanced exploration and exploitation. This resulted in the proportion of region to byte mutations not necessarily reflecting the relative performance difference between region and byte mutations in some targets.

Additionally, the difference in effectiveness among the region mutation operators had an impact on whether adaptive operator selection provided benefit. OpenSSH had two consistently low performing region operators, leading to enhanced coverage when their usage was reduced by the region operator

selector. Exploration also played an important role in favouring the selection of high performing mutation operators without neglecting lower ones. Exim showed a statistically significant improvement with $v = 0.001$, but operator selection at $v = 0.1$ was too uniform to benefit.

In the combined TAMS implementation, interactions between the mutation type and region operator selectors influenced performance. Changes in the proportion of region mutations affected the amount of feedback available to guide operator selection, complicating exploration tuning that was originally calibrated for the default proportion.

Our TAMS design focused on improving the application of region mutations, which are unique to the greybox protocol fuzzer, AFLNET. The aim was validated by measuring the code coverage difference between AFLNET-TAMS and AFLNET, where most targets showed similar performance and OpenSSH experienced a coverage improvement. Overall, these findings indicate that adaptive mutation scheduling can improve performance in protocol fuzzing, despite certain challenges being encountered. This research also showed how different targets benefit from different proportions of region mutations, and how the performance of the individual region mutations varies across the targets, demonstrating the need for a target-adaptive approach. While the performance of TAMS was sensitive to other target-specific mutation characteristics, such as stack size and exploration tuning of the MAB agents, it demonstrated consistent improvement for one target and highlights the potential for further adaptive enhancements.

5 Conclusion

Securing networks relies on ensuring protocol implementations are free of vulnerabilities. Greybox mutational protocol fuzzing, including tools such as AFLNet, has proven effective in testing this type of software. There are additional layers of complexity to consider in protocol fuzzing: the stateful nature of the exchanges, and the syntactic and semantic constraints required to reach deeper layers of the program. Most research in greybox mutational protocol fuzzing has focused on improving the state modelling, while optimizing its mutation strategy has received relatively little attention. Format-aware mutation methods, such as ChatAFL, have shown good results, demonstrating the potential that improvements to AFLNet's mutation strategy can have.

This research modified AFLNet's mutation strategy to select mutation operators based on target feedback, rather than the original purely random approach. This was implemented as two MAB models, one for mutation type selection, and the other for region mutation operator selection. Our aim was achieved by measuring the code coverage before and after implementing this technique into AFLNet using *ProFuzzBench* targets and experimental setup.

Experiments resulted in improvements for OpenSSH, with the best configuration seeing a +0.641% increase in median branch coverage. Other targets experienced comparable coverage to the baseline, but Kamailio saw decreased coverage in some configurations. The addition of the MAB agents did not have any noticeable impact on execution speeds, and TAMS experiments resulted in smaller average seed lengths across all targets.

5.1 Contributions

This research made the following contributions:

- An evaluation of the impact of mutation parameters, such as proportion of region mutations and region stack size, on five different network protocol targets;

- An extension of AFLNET's region mutation set through the addition of a *delete* region mutation operator which randomly selects and removes a message from the seed;
- A mutation type selector for AFLNET using Thompson Sampling which dynamically adjusts the proportion of byte and region mutations according to their observed success;
- A region mutation operator selector for AFLNET using Linear Thompson Sampling which adaptively chooses region mutation operators based on their previous performance and considering the current seed length;
- Empirical evidence that mutation operator performance in greybox protocol fuzzing varies depending on the target;
- An experimental analysis which demonstrates that adjusting mutation type proportions and region operator selection can improve coverage for certain targets, with measurable gains observed for OpenSSH; and
- An analysis showing that TAMS does not have a noticeable impact on execution speed, indicating that the MAB overhead is minimal.

5.2 Limitations

This research used average median branch coverage to approximate for bug finding. The targets tested are quite mature, and no crashes were detected for the majority of the targets tested. To better compare small coverage increases across the targets, we used code coverage which is strongly correlated with bug-finding, however, a fuzzer which covers more code does not always discover more bugs [67].

The experiments were limited to five protocol targets only due to time constraints. Since target characteristics have a large impact on overall performance, generalizability to additional targets is unknown. However, the selected targets included a range of protocols to partially mitigate this limitation.

Our validation experiments each lasted 24 hours and twenty runs of each configuration were conducted. Although this repetition minimizes the fluctuations in performance due to the randomness of fuzzing, results are still impacted by it. Since protocol fuzzing is significantly slower than binary fuzzing, it may be necessary to run fuzzing for a longer minimum time to better show the effects of different mutation strategies. Particularly for targets such as Kamailio who continue to achieve coverage gains fairly regularly later in the fuzzing run. Environments were standardized by using Docker containers, however, there can still be environmental factors which result in variation

in fuzzing speed. With a longer run, the coverage discovery rate will drop off and variations in speed will have less of an impact.

5.3 Future Work

This research focused on improving greybox mutational fuzzing by modifying its mutation strategy. Some improvements to our design and other areas for future research are:

- **Adaptive exploration in MAB selector agents.** Our experiments saw how the MAB's exploration varied across targets with different chances of reward and different execution speeds. As we did not tune exploration for each individual target since our design aims to be target-adaptive, an improvement would be to implement a design that is able to adjust its exploration by analyzing the speed and rate of reward during fuzzing.
- **Alternative adaptive algorithms.** While this research employs TS and LinTS, future work could explore alternative bandit algorithms, swarm intelligence, evolutionary strategies, or reinforcement learning approaches for achieve adaptive selection.
- **Adaptive stack size selection.** Our preliminary and tuning experiments showed how stack size performance was very dependent on the target. Implementing a method to dynamically adjust stack size selection to targets could improve performance.
- **Adaptive byte mutation selection.** While this research focuses on adaptive selection of mutation type and region operators, the byte operators could be similarly dynamically selected using their historic performance.
- **Seed trimming in protocol fuzzing.** Unlike AFL, AFLNET does not implement seed trimming, which can lead to uncontrolled seed growth. The design and implementation of a trimming strategy compatible with protocol fuzzing could improve efficiency and effectiveness.
- **Establishment of a greybox protocol fuzzing bug-finding benchmark.** Since our ultimate goal is to improve the security of protocol servers through identifying bugs using fuzzing, the creation of a benchmark dataset with bugs could be developed to better compare the bug-finding ability of greybox protocol fuzzers.
- **Different reward signals.** The MABs in this research used new edge coverage as a reward signal, this may not always translate to better overall branch coverage, other reward signals could be explored, such as partial rewards for triggering rare edges or new hit counts for edges.

- **Additional contextual features.** Future work could integrate additional contextual features such as protocol state or other seed characteristics to further inform the adaptive selection process.
- **Reworking of *replace* region operator.** The current *replace* region operator replaces the entire M_2 subsequence which can overwrite any other mutations that were made. An improvement to this implementation could be to only replace one message in the M_2 subsequence.
- **Modify the definition of "interesting" test cases.** The "interesting" definition (test cases added to corpus) could be modified to exclude new hit count, as it is not directly linked to new code coverage, dominated in number over new edges, and was frequently triggered by the *duplicate* operator, leading to longer seeds that may hinder fuzzing overall.

List of References

[1] M. Zalewski, "American fuzzy lop - whitepaper," 2016. [Online]. Available: https://lcamtuf.coredump.cx/afl/technical_details.txt

[2] Google. OSS-Fuzz: Continuous fuzzing for open source software. [Online]. Available: https://github.com/google/oss-fuzz

[3] Google. ClusterFuzz. [Online]. Available: https://github.com/google/clusterfuzz

[4] Microsoft. RESTler. [Online]. Available: https://github.com/microsoft/restler-fuzzer?tab=readme-ov-file#restler

[5] Microsoft. OneFuzz. [Online]. Available: https://github.com/microsoft/onefuzz

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

[7] 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, no. 4, p. 960–974, Apr. 2025.

[8] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[9] S. Généreux, "Syntax-aware greybox protocol fuzzing," Master's thesis, Royal Military College of Canada, 2025. [Online]. Available: https://espace.rmc.ca/jspui/bitstream/11264/2423/1/genereux-masc-thesis_FINAL.pdf

[10] F. Hu, J. Ji, H. Shu, Z. Li, T. Liu, and C. Zhang, "Formatted stateful greybox fuzzing of TLS server," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2024, pp. 151–160.

[11] Z. Han, L. Li, L. Chen, and Y. Lin, "Network protocol greybox fuzzing based on semantic fragments," in *2024 6th International Academic Exchange Conference on Science and Technology Innovation (IAECST)*. IEEE, 2024, pp. 424–429.

[12] Q. Tao, "GONet: Gradient oriented fuzzing for stateful network protocol: Improving and evaluating fuzzing efficiency of stateful protocol by mutating based on gradient information," Master's thesis, KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Stockholm, Sweden, Jun. 2023.

[13] K. Yang, W. Lin, Y. Zhu, J. Huang, W. Zhao, and J. Wang, "Dynamic adjustment optimization method of mutation operator based on hybrid fa-pso," in *2025 10th International Conference on Intelligent Computing and Signal Processing (ICSP)*, 2025, pp. 882–890.

[14] Y. Shen, Y. Liu, and Y. Zhou, "RLGFuzz: Reinforcement learning guided fuzzing with state-coverage mapping environment," in *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, 2024, pp. 1–9.

[15] L. Yu, S. Yanlong, and Z. Ying, "Stateful protocol fuzzing with statemap-based reverse state selection," 2024. [Online]. Available: https://arxiv.org/abs/2408.06844

[16] 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*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138.

[17] 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.

[18] 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)*. IEEE, May 2024, p. 1974–1993.

[19] S. B. Andarzian, C. Daniele, and E. Poll, "On the (in)efficiency of fuzzing network protocols," *Annals of Telecommunications*, vol. 80, no. 7, pp. 659–668, 2025.

[20] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "NSFuzz: Towards efficient and state-aware network service fuzzing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, Sep. 2023.

[21] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: Packet sequence oriented fuzzing for protocol implementations," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4481–4498. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/luo-zhengxiong

[22] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, Oct. 2022.

[23] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: combining incremental steps of fuzzing research,” ser. WOOT’20. USA: USENIX Association, 2020.

[24] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, “One fuzzing strategy to rule them all,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1634–1645.

[25] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, “MOPT: Optimized mutation scheduling for fuzzers,” in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC’19. USA: USENIX Association, 2019, p. 1949–1966.

[26] M. Eddington, “Peach fuzzer,” https://peachtech.gitlab.io/peach-fuzzer-community/WhatIsPeach.html.

[27] P. Amini and A. Portnoy, “Fuzzing Sucks! - Introducing Sulley Fuzzing Framework,” in *Black Hat USA*, Las Vegas, NV, USA, Aug. 2007, presented at Black Hat USA.

[28] 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*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 662–665. [Online]. Available: https://doi.org/10.1145/3460319.3469077

[29] D. Merkel, “Docker: lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[30] Free Software Foundation, *gcov - A Test Coverage Program*, GNU. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[31] H. Zheng, F. Toffalini, M. Böhme, and M. Payer, “MendelFuzz: The return of the deterministic stage,” *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025.

[32] A. Slivkins, *Introduction to Multi-Armed Bandits*, 2019.

[33] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time analysis of the multiarmed bandit problem,” *Machine Learning*, vol. 47, pp. 235–256, 05 2002.

[34] S. Bubeck and A. Slivkins, “The best of both worlds: Stochastic and adversarial bandits,” in *Proceedings of the 25th Annual Conference on Learning Theory*, ser. Proceedings of Machine Learning Research, S. Mannor, N. Srebro, and R. C. Williamson, Eds., vol. 23. Edinburgh, Scotland: PMLR, 25–27 Jun 2012, pp. 42.1–42.23. [Online]. Available: https://proceedings.mlr.press/v23/bubeck12b.html

[35] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3-4, pp. 285–294, 12 1933.

[36] S. Agrawal and N. Goyal, "Analysis of thompson sampling for the multi-armed bandit problem," in *Conference on learning theory*. JMLR Workshop and Conference Proceedings, 2012, pp. 39–1.

[37] E. Kaufmann, N. Korda, and R. Munos, "Thompson sampling: An asymptotically optimal finite-time analysis," in *International conference on algorithmic learning theory*. Springer, 2012, pp. 199–213.

[38] O. Chapelle and L. Li, "An empirical evaluation of thompson sampling," in *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, Eds., vol. 24. Curran Associates, Inc., 2011. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2011/file/e53a0a2978c28872a4505bdb51db06dc-Paper.pdf

[39] T. Lattimore and C. Szepesvári, *Bandit Algorithms*. Cambridge University Press, 2020. [Online]. Available: https://tor-lattimore.com/downloads/book/book.pdf

[40] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proceedings of the 19th international conference on World wide web*. ACM, Apr. 2010, pp. 661–670.

[41] S. Agrawal and N. Goyal, "Thompson sampling for contextual bandits with linear payoffs," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 127–135. [Online]. Available: https://proceedings.mlr.press/v28/agrawal13.html

[42] "TensorFlow agents: Linear thompson sampling agent," https://github.com/tensorflow/agents/blob/master/tf_agents/bandits/agents/linear_thompson_sampling_agent.py.

[43] K. Bottinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *2018 IEEE Security and Privacy Workshops (SPW)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2018, pp. 116–122.

[44] W. Drozd and M. D. Wagner, "FuzzerGym: A competitive framework for fuzzing and learning," 2018.

[45] S. Karamcheti, G. Mann, and D. Rosenberg, "Adaptive grey-box fuzz-testing with thompson sampling," 08 2018.

[46] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "DARWIN: Survival of the fittest fuzzing mutators," in *Proceedings 2023 Net-*

work and Distributed System Security Symposium, ser. NDSS 2023. Internet Society, 2023.

[47] X. Zhao, H. Qu, J. Xu, S. Li, and G.-G. Wang, "AMSFuzz: An adaptive mutation schedule for fuzzing," *Expert Systems with Applications*, vol. 208, p. 118162, 2022.

[48] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2019.

[49] C. Lemieux and K. Sen, "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 475–485.

[50] X. Wang, C. Hu, R. Ma, D. Tian, and J. He, "Cmfuzz: context-aware adaptive mutation for fuzzers," *Empirical Softw. Engg.*, vol. 26, no. 1, Jan. 2021.

[51] M. Lee, S. Cha, and H. Oh, "Learning seed-adaptive mutation strategies for greybox fuzzing," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 384–396.

[52] W. Jiao, X. Li, Q. Li, F. Cao, X. Li, and S. Yue, "Adaptive mutation based on multi-population evolution strategy for greybox fuzzing," *Inf. Sci.*, vol. 705, no. C, Apr. 2025.

[53] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 110–121.

[54] Z. Gao, H. Xiong, W. Dong, R. Chang, R. Yang, Y. Zhou, and L. Jiang, "FA-Fuzz: A novel scheduling scheme using firefly algorithm for mutation-based fuzzing," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 1–15, 2024.

[55] X.-S. Yang, "Firefly algorithm, stochastic test functions and design optimisation," *Int. J. Bio-Inspired Comput.*, vol. 2, no. 2, p. 78–84, Mar. 2010.

[56] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[57] Y. Koike, H. Katsura, H. Yakura, and Y. Kurogome, "SLOPT: Bandit optimization framework for mutation-based fuzzing," in *Proceedings of*

the 38th Annual Computer Security Applications Conference, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 519–533.

[58] "Github," https://github.com.

[59] "GNU scientific library - random number distributions," https://www.gnu.org/software/gsl/doc/html/randist.html.

[60] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, p. 3–30, Jan. 1998.

[61] M. H. DeGroot and M. J. Schervish, *Probability and Statistics*, 4th ed. Boston: Addison-Wesley, 2012.

[62] SciPy, *scipy.stats.beta — A Beta continuous random variable*, https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.beta.html, aPI documentation for SciPy. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.beta.html

[63] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020.

[64] D. M. Diez, C. D. Barr, and M. Çetinkaya-Rundel, *OpenIntro Statistics*, 2nd ed. OpenIntro, Aug. 2012, available online at https://www.openintro.org. Licensed under CC BY-NC-SA 3.0.

[65] D. D. Wackerly, W. M. III, and R. L. Scheaffer, *Mathematical Statistics with Applications*, seventh edition ed. Duxbury Advanced Series, 2002.

[66] NumPy Developers, *numpy.where - NumPy v2.5.dev0 Manual*, NumPy. [Online]. Available: https://numpy.org/devdocs/reference/generated/numpy.where.html

[67] 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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1621–1633. [Online]. Available: https://doi.org/10.1145/3510003.3510230

Appendices

A Success Rates for Mutation Operator Performance Preliminary Experiments

This appendix presents tables detailing the success rates for each mutation operator during the experiments described in Section 3.1.1. The definition of “interesting” used here includes both sub-types: new edge and new hit count, as these were logged together. Rows highlighted in blue indicate region mutation operators, while byte mutation operators remain unhighlighted.

Table A.1: Mutation Operator Success Rate Statistics for LightFTP.

Mutation Operator	**Min (%)**	**Max (%)**	**Mean (%)**	**Std. Dev.**
reg_duplicate	0.11410	0.14624	0.13558	0.00882
insert_bytes	0.11041	0.13534	0.12461	0.00732
reg_append	0.10427	0.12870	0.11882	0.00759
reg_prepend	0.09972	0.12469	0.11792	0.00724
overwrite_bytes	0.09945	0.12745	0.11620	0.00793
dword_int_val	0.10752	0.12452	0.11567	0.00559
overwrite_bytes_extra	0.09861	0.12341	0.11484	0.00750
sub_word	0.09958	0.12483	0.11405	0.00822
sub_dword	0.09761	0.12188	0.11346	0.00802
bit_flip	0.09628	0.12570	0.11342	0.00924
byte_int_val	0.09750	0.12708	0.11331	0.00760
insert_extra	0.09608	0.12689	0.11306	0.00904
add_word	0.09805	0.12636	0.11302	0.00850
word_int_val	0.09786	0.12366	0.11300	0.00765
sub_byte	0.09839	0.12506	0.11300	0.00767
add_dword	0.10030	0.13010	0.11298	0.00895
set_byte_val	0.09916	0.12012	0.11283	0.00750
add_byte	0.10032	0.11879	0.11227	0.00617
delete_bytes	0.09662	0.11800	0.10947	0.00597
reg_replace	0.08648	0.10544	0.09642	0.00588

Table A.2: Mutation Operator Success Rate Statistics for DNSmasq.

Mutation Operator	**Min (%)**	**Max (%)**	**Mean (%)**	**Std. Dev.**
reg_duplicate	0.11649	0.13771	0.12864	0.00682
reg_append	0.10142	0.12890	0.11538	0.00883
reg_prepend	0.09918	0.12694	0.11362	0.00756
insert_bytes	0.09573	0.12080	0.10954	0.00822
sub_dword	0.09350	0.11938	0.10883	0.00853
add_dword	0.09427	0.12059	0.10826	0.00925
sub_byte	0.09092	0.11569	0.10759	0.00786
set_byte_val	0.09427	0.11668	0.10688	0.00713
add_word	0.09258	0.11491	0.10683	0.00798
dword_int_val	0.09368	0.11693	0.10670	0.00775
add_byte	0.09015	0.12020	0.10645	0.00814
byte_int_val	0.09299	0.11434	0.10630	0.00748
bit_flip	0.09160	0.11663	0.10602	0.00841
sub_word	0.09011	0.11813	0.10601	0.00811
overwrite_bytes	0.09271	0.12144	0.10527	0.00865
word_int_val	0.09161	0.11588	0.10464	0.00675
delete_bytes	0.09113	0.11100	0.10155	0.00659
reg_replace	0.07986	0.10051	0.08988	0.00655

Table A.3: Mutation Operator Success Rate Statistics for Exim.

Mutation Operator	**Min (%)**	**Max (%)**	**Mean (%)**	**Std. Dev.**
reg_duplicate	0.42033	0.51136	0.45051	0.02717
insert_bytes	0.38640	0.46389	0.41987	0.02266
overwrite_bytes	0.34049	0.40922	0.37952	0.02093
delete_bytes	0.33652	0.42457	0.37849	0.02572
reg_prepend	0.34077	0.42675	0.37825	0.02681
add_word	0.33309	0.40989	0.36874	0.02355
sub_dword	0.33224	0.39656	0.36771	0.02087
sub_byte	0.34057	0.40666	0.36598	0.02276
bit_flip	0.32775	0.41731	0.36528	0.02634
insert_extra	0.33898	0.41393	0.36444	0.02293
overwrite_bytes_extra	0.33554	0.40095	0.36304	0.01810
sub_word	0.32662	0.39770	0.36257	0.02072
byte_int_val	0.32152	0.42260	0.36245	0.02569
add_dword	0.32044	0.39637	0.36148	0.02016
add_byte	0.32432	0.40408	0.36034	0.02208
set_byte_val	0.33093	0.39865	0.36031	0.02084
dword_int_val	0.31726	0.40276	0.35325	0.02377
word_int_val	0.31053	0.37811	0.34707	0.02053
reg_append	0.31182	0.37734	0.33667	0.02041
reg_replace	0.24727	0.30105	0.27500	0.01797

Table A.4: Mutation Operator Success Rate Statistics for OpenSSH.

Mutation Operator	Min (%)	Max (%)	Mean (%)	Std. Dev.
add_byte	0.04669	0.07912	0.05866	0.00985
overwrite_bytes	0.04304	0.08190	0.05845	0.01130
sub_dword	0.04652	0.07758	0.05840	0.00942
add_dword	0.04449	0.08434	0.05817	0.01090
sub_word	0.04334	0.07910	0.05769	0.01053
bit_flip	0.04520	0.07865	0.05768	0.00976
sub_byte	0.04601	0.07610	0.05766	0.00961
add_word	0.04564	0.07932	0.05745	0.01047
delete_bytes	0.04512	0.07758	0.05739	0.00973
reg_duplicate	0.04447	0.07595	0.05735	0.00968
set_byte_val	0.04404	0.07762	0.05653	0.01021
byte_int_val	0.04429	0.07920	0.05648	0.01023
reg_prepend	0.04288	0.07792	0.05639	0.01049
overwrite_bytes_extra	0.04323	0.07498	0.05614	0.00998
word_int_val	0.04357	0.07689	0.05575	0.01031
insert_bytes	0.04227	0.07797	0.05537	0.01071
dword_int_val	0.04093	0.07744	0.05456	0.01064
insert_extra	0.03959	0.07420	0.05330	0.00982
reg_replace	0.02367	0.07087	0.04390	0.01279
reg_append	0.02169	0.06163	0.04129	0.01103

Table A.5: Mutation Operator Success Rate Statistics for Kamailio.

Mutation Operator	**Min (%)**	**Max (%)**	**Mean (%)**	**Std. Dev.**
reg_duplicate	1.44361	1.78619	1.62396	0.10740
insert_bytes	1.33834	1.67321	1.48791	0.11030
reg_prepend	1.31730	1.65578	1.46760	0.10932
overwrite_bytes	1.30889	1.65477	1.46650	0.11724
reg_append	1.29843	1.63351	1.45592	0.11211
sub_byte	1.28423	1.61432	1.44323	0.10547
add_word	1.27565	1.62161	1.44151	0.11066
sub_dword	1.27913	1.61008	1.44046	0.11058
add_byte	1.29376	1.61634	1.44019	0.11203
bit_flip	1.30390	1.60492	1.43985	0.10256
sub_word	1.27619	1.60414	1.43796	0.11023
set_byte_val	1.27562	1.61785	1.43680	0.11172
add_dword	1.29506	1.60160	1.43422	0.10432
byte_int_val	1.27195	1.58940	1.43356	0.10769
dword_int_val	1.27330	1.62747	1.43018	0.10886
word_int_val	1.26622	1.61324	1.41989	0.11259
delete_bytes	1.21256	1.54693	1.37979	0.11262
reg_replace	1.22398	1.55159	1.37341	0.10346

B LinTS Region Operator Selection Based on Seed Length

This appendix presents graphs showing the region operator selection percentages across different seed lengths for the five targets tested. This data is from the first 10 runs of TAMS-RegOp (*delete* enabled).

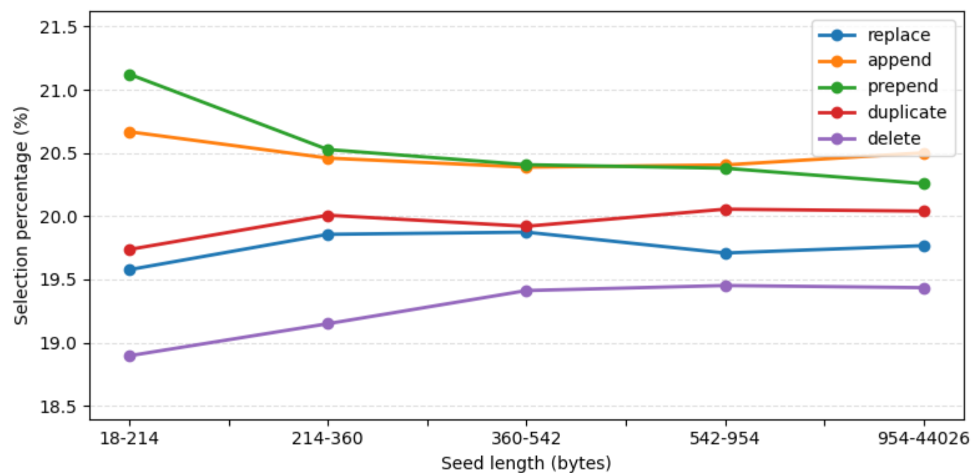

Figure B.1: TAMS-RegOp Region Operator Selection across Seed Lengths for LightFTP.

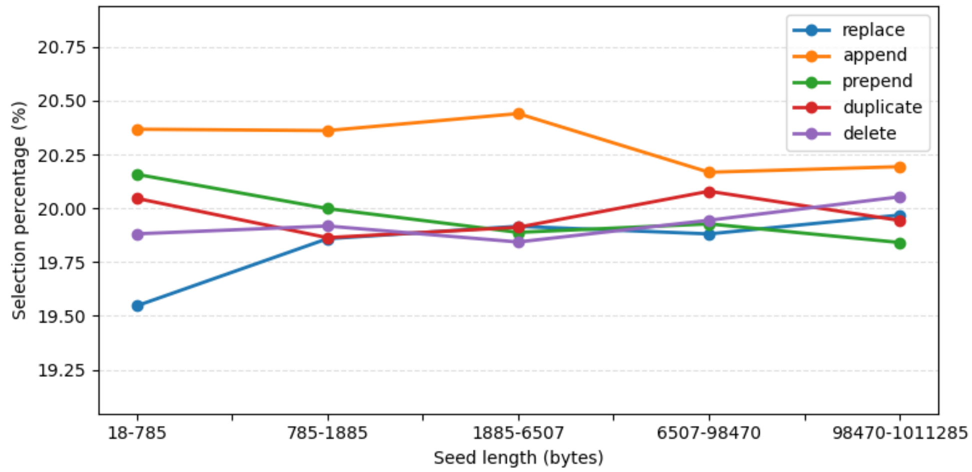

Figure B.2: TAMS-RegOp Region Operator Selection across Seed Lengths for DNSmasq.

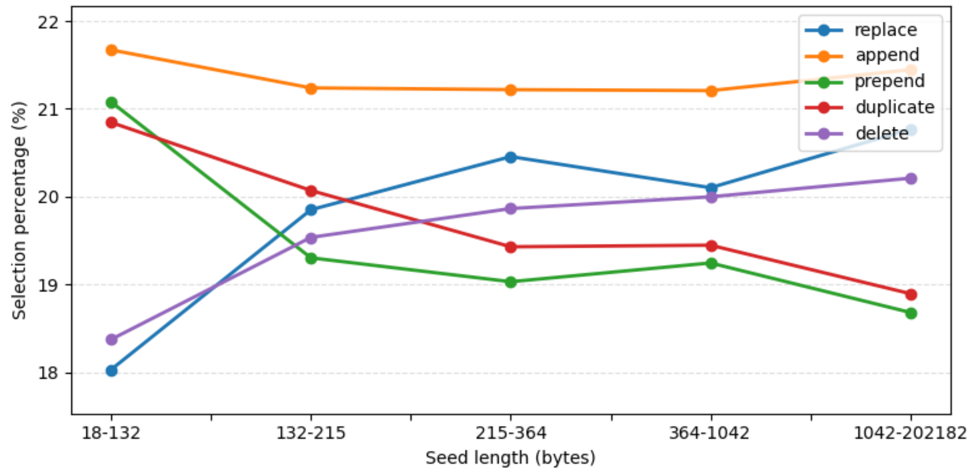

Figure B.3: TAMS-RegOp Region Operator Selection across Seed Lengths for Exim.

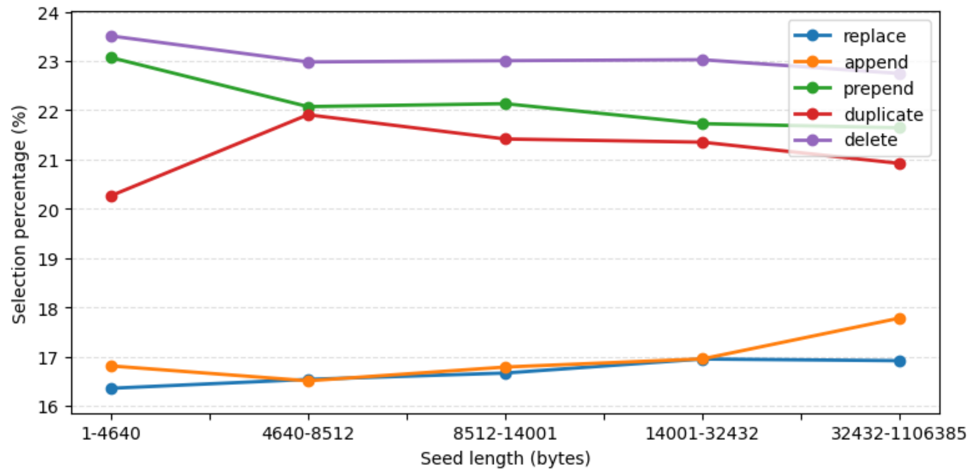

Figure B.4: TAMS-RegOp Region Operator Selection across Seed Lengths for OpenSSH.

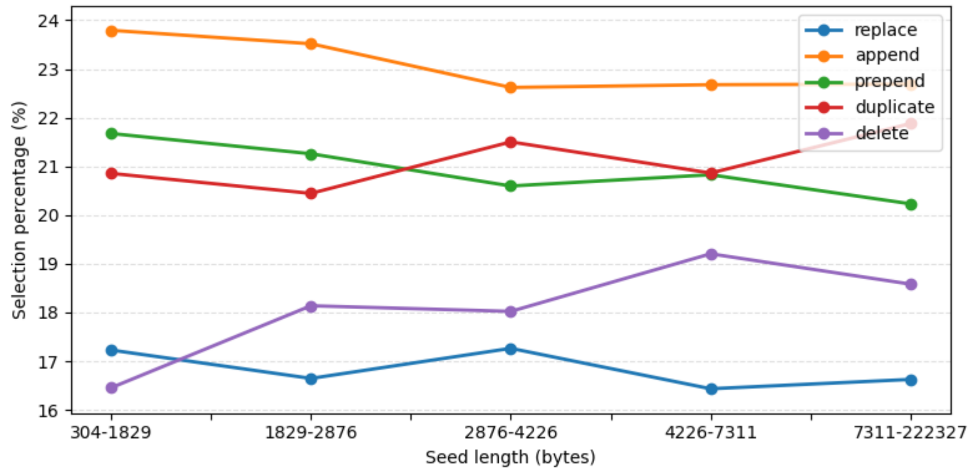

Figure B.5: TAMS-RegOp Region Operator Selection across Seed Lengths for Kamailio.