Understanding the Effectiveness of Mutators in Mutation-based Protocol Fuzzing

Xiyuan Zhang, Jiayi Jiang, Yiutak Choi, Ting Su*, Haiying Sun, Chengcheng Wan, Geguang Pu
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China
zxy6538@gmail.com, jyjiangsunny@gmail.com, jeremyarg9@gmail.com, tsu@sei.ecnu.edu.cn,
hysun@sei.ecnu.edu.cn, ccwan@sei.ecnu.edu.cn, ggpu@sei.ecnu.edu.cn

Abstract—**Mutation-based protocol fuzzing is one of the most widely adopted techniques for finding vulnerabilities in communication protocol implementations. As a key component of such fuzzers, *mutators* are critical for fuzzing performance, which are mostly inherited from general-purpose greybox fuzzers like AFL. However, protocols possess unique characteristics, including taking sequences of structured messages as inputs and requiring stateful interactions. The effectiveness of these mutators in protocol fuzzing is under-explored.**

To this end, this paper conducts the *first* empirical study on 13 real-world, diverse protocol implementations to understand these mutators' effectiveness. With respect to the protocol characteristics, the study covers three important dimensions: *mutation granularity*, *mutation level*, and *interplay with splicing*. The study obtains several interesting findings, most of which are unknown or unclear before. We find that the mutators' effectiveness is closely related to the characteristics of protocols. Based on these findings, we propose a mutator selection strategy and apply it to AFLNet, a state-of-the-art mutation-based protocol fuzzer. Evaluated on 15 protocol implementations, this strategy improves branch coverage up to 12.67% compared to AFLNet, and uncovers 8 unique bugs which AFLNET fails to find. We believe that this study sheds light on future work for selecting and designing better mutators for improving protocol fuzzing effectiveness. All the artifacts in our study have been made publicly available at *https://github.com/ecnusse/AFLNet_Mutators_Study*.

Index Terms—**Protocol fuzzing, mutator, network security**

I. Introduction

Communication protocols are the foundation of many critical infrastructures (*e.g.*, internet, industrial controls, medical systems) [1]. Securing network protocols is thus important (*e.g.*, preventing denial-of-service attacks). To this end, *protocol fuzzing* is one of the widely-used approaches to help find vulnerabilities in protocol implementations [2], [3], in addition to formal verification [4].

In protocol fuzzing, mutation-based protocol fuzzing [5]–[16] is one prominent fuzzing technique. Unlike generation-based protocol fuzzing [17], [18], it does not require the substantial manual efforts and expertise of writing protocol state and data models. Its core idea is influenced by the well-known coverage-guided greybox fuzzing like AFL [19]. It repeatedly uses a number of *mutation operators* (*mutators* for short), based on the feedback of coverage, to generate new, interesting seeds. These seeds are used to find potential bugs. Therefore, the mutators are critical for the fuzzing effectiveness.

Specifically, AFLNET [6], [20] is the first *and* state-of-the-art mutation-based protocol fuzzer built on AFL [19]. It was released in 2020 and has garnered over 950 stars on GitHub. In practice, AFLNET has found many critical protocol vulnerabilities [21]–[25] and has been extended by several subsequent works [5], [7], [26]. Internally, AFLNET reuses the common *byte-level* mutators (*e.g.*, bit flips, arithmetic changes, block duplication) from AFL and designs some *region-level* mutators to perform protocol-aware mutations (*e.g.*, replacement, insertion, duplication, and deletion of messages).

However, unlike traditional program, communication protocols have their own unique characteristics. For example, the input to be fuzzed in protocol fuzzing is a sequence of protocol messages rather than raw file-based inputs. One message usually contains multiple fields with different lengths, and some fields might have correlations. Moreover, many protocols are *stateful* — interpreting and handling some messages may depend on the prior ones and the current protocol states [27], [28]. Additionally, protocols are usually implemented in different business logic. The aforementioned unique characteristics of protocols lead to some important questions, which are under-explored: *How effective are the mutators in existing mutation-based protocol fuzzers on fuzzing protocol implementations? Which mutators are more effective? Are there any differences between these mutators?*

To this end, this paper takes the *first* step to study the effectiveness of the mutators in mutation-based protocol fuzzers. This study could (1) deepen the understanding of these mutators in protocol fuzzing for the community, *and* (2) shed lights on selecting and designing better mutators for effective protocol fuzzing. To perform this study, we select AFLNET as the target fuzzer because it is the most representative mutation-based protocol fuzzer. Specifically, we explore the following three research questions from the perspective of the characteristics of protocols:

- **RQ1 (Mutation granularity)**. A protocol message usually contains multiple fields with different lengths. Meanwhile, the common byte-level mutators in AFLNET (inherited from AFL) mutate the seeds at different granularities (stepovers), *e.g.*, *bits*, *bytes* or *random-length bytes*. Thus, we aim to investigate how would the byte-level mutators with different mutation granularities affect protocol fuzzing?
- **RQ2 (Mutation level)**. Many protocols are stateful in which the messages have dependency and affect the internal

* Ting Su is the corresponding author.

states of the protocols. To this end, AFLNET designs region-level mutators to perform protocol-aware mutations (*e.g.*, replacement, insertion, duplication, and deletion of messages). Meanwhile, AFLNET uses the common byte-level mutators to mutate the messages as raw bytes. Thus, we aim to investigate how would the mutators of these two different mutation levels (region-level and byte-level) perform in protocol fuzzing (*e.g.*, which mutation level is more effective?)

- **RQ3 (Interplay with Splicing)**. In mutation-based fuzzing, splicing is a mutator of combining two parent inputs with the goal of finding a new interesting input. It is usually activated when other mutators cannot find any new interesting inputs, and has already shown its effectiveness on traditional program. However, splicing leads to drastic changes on the inputs. Thus, we aim to investigate how would byte-level mutators as well as region-level mutators interplay with splicing on fuzzing protocols which are usually stateful. How would splicing affect protocol fuzzing?

To investigate these RQs, we selected 13 different protocol implementations (corresponding to ten protocols) as the subjects from PROFUZZBENCH [29], a widely-used benchmarking platform for evaluating protocol fuzzers [5], [7], [13], [30]. It comprises a suite of mature and open-source programs that implement well-known network protocols. Our investigation took substantial machine hours for fuzzing these protocol implementations with different configurations of the mutators.

In response to RQ1, we find that aligning mutation granularity with the characteristics of a protocol's field lengths is crucial for effective fuzzing. Specifically, the mutators that operate on random-length bytes are more aligned with text-based protocols, while those that operate on bit and fixed-length bytes are more aligned with binary protocols. This alignment results in a 5%-13% increase in branch coverage for the former (random-length bytes mutators) on text-based protocols, and a 30%-40% increase for the latter (bit and fixed-length byte mutators) on binary protocols. Moreover, each achieves statistically significantly *more* branch coverage than the other. Regarding RQ2, byte-level mutators outperform region-level ones by an average of 23.3% in branch coverage. However, each type demonstrates distinct strengths: byte-level mutators are more effective at covering parser-related branches, whereas region-level mutators are more effective at exploring state-dependent branches. As for RQ3, we find that splicing could lead to gains or losses on branch coverage for region-level and byte-level mutators. However, splicing does not lead to statistically significant change on overall fuzzing performance in terms of branch coverage.

Informed by our findings, we introduce a mutator selection strategy for mutation-based protocol fuzzing based on AFLNET. We evaluate this strategy on all the 13 protocols from PROFUZZBENCH and two new protocols which have not been studied in our RQs. The results show that this strategy can improve branch coverage up to 12.67% compared to AFLNET. This strategy also helps uncover 8 unique bugs in the tested protocol implementations which fail to be found by AFLNET. These results suggest that selecting appropriate mutators based on the characteristics of protocols can improve fuzzing performance. In summary, this paper has made the following main contributions:

- To our knowledge, we conduct the *first* study to investigate the effectiveness of mutators in mutation-based protocol fuzzing, which is under-explored.
- We study the effectiveness of mutators from three perspectives based on the characteristics of protocols. We distill several interesting findings, most of which are unknown or unclear before in the community.
- We design a mutation selection strategy based on our findings. This strategy indicates that selecting an appropriate subset of mutators could outperform the original AFLNET in branch coverage and bug finding.

II. BACKGROUND

A. Communication Protocols and Protocol Characteristics

This section briefly introduces the background of communication protocols and their characteristics. A communication protocol typically involves a *server* and *clients*. It defines the rules and regulations on how the data (represented as *messages*, or named as *packets*) is transmitted between the server and clients. The *input* of a server is thus *a sequence of messages* from the clients. A client sends *request* messages to the server, and the server gives *responses*.

Typically, a message m is composed of a number of *fields*. Each field f is represented as a unit of data and serves a specific purpose. A field f could be of different lengths (defined in either *fixed* or *variable* lengths in bits or bytes). Based on how the data of the fields in a protocol are represented, a protocol can be classified into (1) *text-based* protocols when the fields are encoded in human-readable characters; (2) *binary* protocols when the fields are encoded in raw binary form (0s and 1s); and (3) *mixed* protocols when some fields are encoded in human-readable characters and some are in raw binary form.

A protocol could be *stateful* or *stateless*. When a protocol is stateful, the server may update its internal state to remember the status of interaction with the clients upon receiving a sequence of request messages M (M=$[m_1, \ldots, m_n]$). In such cases, the messages $m_1, \ldots, m_n$ in M may have dependency. For a stateless protocol, the messages are independent. In practice, a server usually implements a *parser* to read and sanitize the request messages, and later uses *handlers* to process and respond valid messages.

Example. DICOM (Digital Imaging and Communications in Medicine) is a mixed, stateful protocol designed for the storage and transmission of medical images and related data. Figure 1 presents one typical sequence of DICOM messages. Message 1 is an associate request. It contains a PDU header (including 1-byte PDU Type, 1-byte padding, and 4-byte PDU Length) and a payload. The payload includes (1) several fixed-length, binary fields, *i.e.*, the Protocol Version (2-byte), the Called and Calling Entity Title (16-byte each), and some paddings

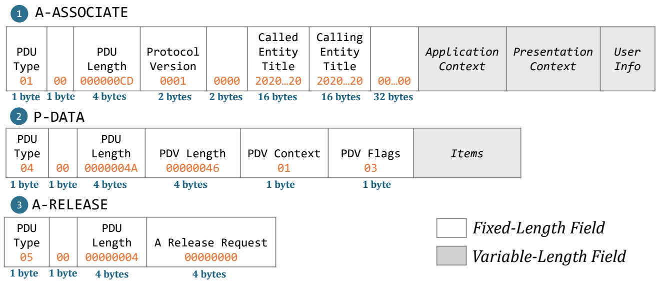


Fig. 1. An example of a sequence of messages of DICOM.

TABLE I
MUTATION OPERATORS IN AFLNET

Mutator	Description
M_1	Flip a specific bit
M_2	Set a specific byte to an interesting value
M_3	Set a specific word to an interesting value
M_4	Set a specific dword to an interesting value
M_5	Decrease a specific byte value randomly
M_6	Increase a specific byte value randomly
M_7	Decrease a specific word value randomly
M_8	Increase a specific word value randomly
M_9	Decrease a specific dword value randomly
M_{10}	Increase a specific dword value randomly
M_{11}	Set a specific byte value to a random number
M_{12}	Delete random-length bytes
M_{13}	Insert random-length content (75% original, 25% random)
M_{14}	Replace random-length content (75% original, 25% random)
M_{15}	Replace a specific position with a random token
M_{16}	Insert a random token at a specific position
M_{17}	Randomly replace the region with the previous region
M_{18}	Insert a region before the current region
M_{19}	Insert a region after the current region
M_{20}	Overwrite the current region

in between; and (2) three variable-length fields including the Application Context (an object-identifier string), Presentation Context, and User Info — each of these three fields is composed of a mix of fixed-length and variable-length subfields.

In Figure 1, the three messages A-ASSOCIATE, P-DATA and A-RELEASE have dependency. In the A-ASSOCIATE message, the fields Called Entity Title, Calling Entity Title and Application Context together setup the communication session and some other fields define how the subsequent messages are interpreted. For example, Presentation Context in A-ASSOCIATE defines the command semantics and encoding rules that the subsequent P-DATA messages should follow. The server expects a P-DATA message should carry specific items that conform to the active presentation context. The A-RELEASE message will only be processed after the A-ASSOCIATE and P-DATA messages are successfully processed. If these three messages arrive out of order or lack the necessary context, the server may reject them.

B. Mutation-based Protocol Fuzzing

AFLNET [6], [20] is the state-of-the-art mutation-based protocol fuzzer. Table I lists all the mutators in AFLNET. AFLNET reuses the common byte-level mutators (M_1, …, M_{16} in Table I) from AFL, and designs four additional region-level mutators (M_{17}, …, M_{20} in Table I) to perform the protocol-aware mutations.

TABLE II
DETAILED INFORMATION OF TARGET PROTOCOL IMPLEMENTATIONS

Implementation	Protocol	State	Content	Commit
BFTPD	FTP	Stateful	Text	5.7
Forked-daapd	DAAP	Stateless	Text	2ca10d9
Dnsmasq	DNS	Stateless	Mixed	b8f1655
Exim	SMTP	Stateful	Text	86e5b23
LightFTP	FTP	Stateful	Text	5980ea1
Live555	RTSP	Stateful	Text	31284aa
ProFTPD	FTP	Stateful	Text	4017eff
PureFTPD	FTP	Stateful	Text	c21b45f
Kamailio	SIP	Stateful	Text	2648eb3
DCMTK	DICOM	Stateful	Mixed	7f8564c
TinyDTLS	DTLS	Stateful	Binary	06995d4
OpenSSH	SSH	Stateful	Mixed	7cfea58
OpenSSL	TLS	Stateful	Binary	0437435

AFLNET structures its mutation process around the idea that a message sequence should be split into three parts in order to preserve the protocol state being exercised. Given a selected state s and an original sequence $\mathcal{M}$, the sequence is divided into a prefix $\mathcal{M}_1$ that is necessary to reach s, a candidate subsequence $\mathcal{M}_2$ that contains messages which may be mutated while still remaining in state s, and a suffix $\mathcal{M}_3$ that follows $\mathcal{M}_2$ so that $\langle \mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3 \rangle = \mathcal{M}$. AFLNet produces a mutated sequence $\mathcal{M}' = \langle \mathcal{M}_1, mutate(\mathcal{M}_2), \mathcal{M}_3 \rangle$. By preserving $\mathcal{M}_1$, the fuzzer ensures that $\mathcal{M}'$ still arrives at the intended state, and by executing $\mathcal{M}_3$ after the mutated candidate $\mathcal{M}_2$ it observes how changes propagate to later responses. More details can be found in Meng et al. [20].

When mutating the candidate subsequence $\mathcal{M}_2$, mutators are operated in a *stacked* manner known as *havoc*: region-level and byte-level mutators are applied in sequence to produce a single mutated candidate for $\mathcal{M}_2$. In addition, *splicing* combines two different seeds to generate hybrid inputs when other mutations fail to yield progress.

III. EXPERIMENTAL SETUP

To facilitate our experiment, we introduced some *switches* into AFLNET to control the activation of different mutators. We followed the default settings of AFLNET in our experiment and used all the 13 protocol implementations from PROFUZZBENCH [29] as our subjects. PROFUZZBENCH is a widely-used benchmarking platform for evaluating protocol fuzzers [5], [7], [13], [30]. It comprises a suite of mature and open-source programs that implement well-known network protocols (*e.g.*, SSH and FTP). Table II lists the 13 protocol implementations (corresponding to ten different protocols) used in our experiment. We used the default seed corpus from PROFUZZBENCH for fuzzing. We conducted a 24-hour fuzzing campaign for each round of evaluation and repeated each campaign five times to mitigate randomness.

IV. RQ1: MUTATION GRANULARITY

Setup. RQ1 aims to investigate the impact of mutation granularity of the byte-level mutators on the effectiveness of

TABLE III

AVERAGE NUMBERS OF BRANCHES COVERED BY THE FIVE GROUPS OF MUTATORS WITH DIFFERENT MUTATION GRANULARITIES.

Implementation	Protocol	Content	**G$_{Bit}$**	**G$_{Byte}$**	**G$_{Word}$**	**G$_{Dword}$**	**G$_{S}$**
BFTPD	FTP	Text	**423 (0.01)**	430 (0.14)	438 (0.14)	436 (0.06)	447
Forked-daapd	DAAP	Text	2203 (1.00)	2167 (0.42)	2145 (0.15)	2225 (0.55)	2195
Dnsmasq	DNS	Mixed	**784 (<0.01)**	**759 (<0.01)**	**819 (0.02)**	**811 (0.03)**	869
Exim	SMTP	Text	**3196 (<0.01)**	**3172 (0.01)**	**3199 (0.02)**	**3178 (<0.01)**	3291
LightFTP	FTP	Text	**334 (0.01)**	**344 (0.02)**	346 (0.17)	**341 (0.02)**	352
Live555	RTSP	Text	**2739 (<0.01)**	**2789 (0.03)**	**2788 (0.04)**	**2778 (0.02)**	2853
ProFTPD	FTP	Text	**4466 (<0.01)**	4646 (1.00)	4717 (0.70)	4732 (0.07)	4669
PureFTPD	FTP	Text	**699 (<0.01)**	**734 (0.01)**	**880 (0.03)**	**808 (0.02)**	1086
Kamailio	SIP	Text	**7221 (0.04)**	8369 (0.22)	8332 (0.15)	8327 (0.15)	8955
DCMTK	DICOM	Mixed	**7382 (<0.01)**	**7179 (<0.01)**	**7265 (<0.01)**	**7239 (<0.01)**	6906
TinyDTLS	DTLS	Binary	**592 (0.01)**	**560 (0.01)**	**635 (0.01)**	**614 (0.01)**	382
OpenSSH	SSH	Mixed	**3339 (<0.01)**	**3334 (<0.01)**	**3322 (<0.01)**	**3325 (<0.01)**	3137
OpenSSL	TLS	Binary	**10055 (<0.01)**	**10056 (<0.01)**	**9971 (<0.01)**	**10146 (<0.01)**	8805

* The numbers in parentheses denote the p-values between G$_S$ and the other categories G$_{Bit}$, G$_{Byte}$, G$_{Word}$, and G$_{Dword}$.

* Cells shaded in red indicate that G$_S$ achieves statistically significantly more coverage than G$_{Bit}$, G$_{Byte}$, G$_{Word}$, or G$_{Dword}$, while cells shaded in blue indicate that G$_S$ achieves statistically significantly less coverage than G$_{Bit}$, G$_{Byte}$, G$_{Word}$, or G$_{Dword}$. Cells with bold texts indicate a large effect size ($\hat{A}_{12} > 0.71$).

TABLE IV

CATEGORIES OF BYTE-LEVEL MUTATORS BY MUTATION GRANULARITY

Mutation Granularity	**Mutators**
G$_{Bit}$	M$_1$
G$_{Byte}$	M$_2$, M$_5$, M$_6$, M$_{11}$
G$_{Word}$	M$_3$, M$_7$, M$_8$
G$_{Dword}$	M$_4$, M$_9$, M$_{10}$
G$_S$	M$_{12}$, M$_{13}$, M$_{14}$

protocol fuzzing. To this end, we classified the byte-level mutators into five groups based on their mutation granularity. Table IV shows these five groups, *i.e.*, G$_{Bit}$ (one bit), G$_{Byte}$ (one byte), G$_{Word}$ (two bytes), G$_{Dword}$ (four bytes) and G$_S$ (random-length bytes ranging from one byte to a calculated maximal length). To independently evaluate the impact of each granularity group, we enable the mutators in one granularity group (*e.g.*, enabling G$_{Bit}$) while disabling the mutators in the others (*e.g.*, disabling G$_{Byte}$, G$_{Word}$, G$_{Dword}$ and G$_S$) during the fuzzing campaign. We measure the number of branches covered by each granularity group.

Results. Table III presents the average number of branches covered by G$_{Bit}$, G$_{Byte}$, G$_{Word}$, G$_{Dword}$ and G$_S$, respectively. We pairwisely compared the covered branches of the five granularity groups. We employed statistical analysis (Mann-Whitney U-test) to assess whether the differences of covered branches are statistically significant and quantified the effect sizes (Vargha-Delaney's $\hat{A}_{12}$) of the differences.

According to the pairwise comparison results, we find that the differences between G$_S$ and the other four categories G$_{Bit}$, G$_{Byte}$, G$_{Word}$, G$_{Dword}$ are the *most significant* (shaded in Table III). Specifically, (1) on the *text-based protocols* (except Forked-daapd), G$_S$ achieves statistically significantly *more* branch coverage than G$_{Bit}$, G$_{Byte}$, G$_{Word}$, or G$_{Dword}$ (cells shaded in red); (2) on the *binary protocols*, G$_S$ achieves statistically significantly *less* branch coverage than G$_{Bit}$, G$_{Byte}$, G$_{Word}$, or G$_{Dword}$ (cells shaded in blue). Moreover, all these shaded cells indicate a large effect ($\hat{A}_{12} > 0.71$).

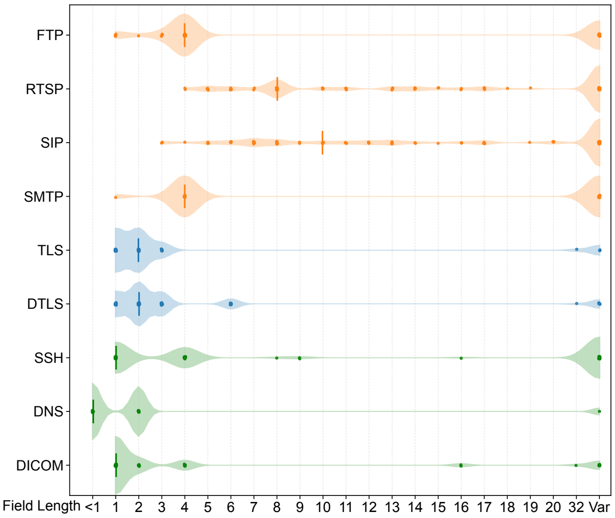


Fig. 2. Distribution of field lengths of the studied text-based (orange), binary (blue) and mixed (green) protocols. "Var" denotes the variable-length fields.

In detail, on all the text-based protocols, G$_S$ covers 13%, 9%, 5%, 6% more branches than G$_{Bit}$, G$_{Byte}$, G$_{Word}$ and G$_{Dword}$, respectively. Specifically, G$_S$ always achieves statistically significant improvements over G$_{Bit}$ (except Forked-daapd). Meanwhile, on the binary protocols (TinyDTLS and OpenSSL), G$_{Bit}$, G$_{Byte}$, G$_{Word}$ and G$_{Dword}$ outperform G$_S$ by covering 35%, 30%, 40% and 38% more branches, respectively. On the binary protocol TinyDTLS, G$_{Dword}$ covers 61% more branches than G$_S$, and also outperforms G$_{Bit}$ and G$_{Byte}$ with statistically significance.

Analysis. Based on the preceding results, our intuition is that *fuzzing is more effective when the mutation granularity of the mutators aligns with the characteristics of a protocol's field lengths*. To verify our intuition, we referred to the RFCs of these protocols, and collected the statistics of fields and their lengths. Figure 2 presents the distributions of the field lengths of the studied protocols by violin plots. The x-axis denotes the field lengths including *fixed-lengths* (denoted by the scales ranging from bits to the maximal 32 bytes) and *variable-lengths* (denoted by "Var" at the far right). The width of each violin plot indicates the density (proportion) of the fields with that length (annotated by a dot) within the corresponding protocol.

Figure 2 shows that, for the two binary protocols (TLS and DTLS), most fields are fixed-length (1, 2 or 3 bytes). It conforms to the characteristics of binary protocols which feature short, fixed-length fields for ensuring communication efficiency. Importantly, this distribution aligns well with the mutation granularity (stepovers) of the mutators in G_{Bit}, G_{Byte}, G_{Word} and G_{Dword}. A mutator is most effective when its mutation granularity is aligned with a field length, as it can efficiently explore the value space of this field. As a result, G_{Bit}, G_{Byte}, G_{Word} and G_{Dword} significantly outperform G_S on the binary protocols in Table III.

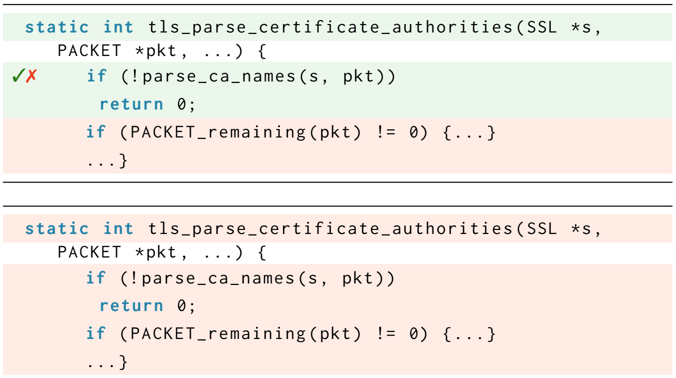
Figure 2 shows that the text-based protocols (FTP, RTSP, SIP, SMTP) have higher proportion of longer fixed-length fields and variable-length fields than the binary protocols. It conforms to the characteristic of text-based protocols: the fields are encoded in human-readable characters and usually occupy more bytes than raw binary. This characteristic aligns well with the mutation granularity (stepovers) of the mutators in G_S.

Figure 2 also shows the distributions of the mixed protocols (SSH, DNS and DICOM) which incorporate both text-based and binary fields. Our intuition also applies to SSH and DICOM which have the similar characteristic of field lengths with the two studied binary protocol. Thus, G_{Bit}, G_{Byte}, G_{Word} and G_{Dword} outperform G_S on OpenSSH and DCMTK. But the protocol DNS is an outlier although it has the similar characteristic with the binary protocols. The reason is that the field `domain name` in DNS supports the most critical functionality of query processing and response generation. This field is the only text-based, variable-length field in DNS and ranges from 1 to 255 bytes. As a result, G_S is most effective in mutating this field, leading to significantly higher coverage on Dnsmasq than the other granularity groups.

Forked-daapd differs from other protocol implementations. It processes HTTP requests via API endpoints rather than message fields. As a result, its messages are mainly composed of different API calls and parameters rather than fields. This fundamental difference reduces the impact of mutation granularity of mutators, leading to similar fuzzing effectiveness across different granularity groups.

Case Study. To illustrate the impact of mutation granularity, we compare coverage results from two protocol implementations—one binary (OpenSSL [31]) and one text-based (Live555 [32]). These examples demonstrate how mutators with different granularity interact with protocol implementations, ultimately influencing the coverage.

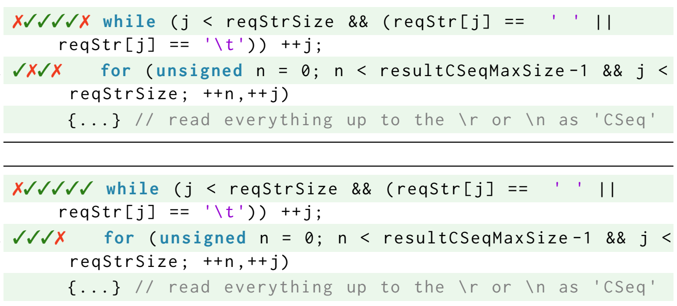
OpenSSL implements the TLS protocol, where the ClientHello message includes multiple structured fields, each beginning with a fixed two-byte *identifier* followed by data. Take Figure 3 and `certificate_authorities` extension as an example, whose identifier field is set to `0x002f`. Since this value not included in the initial seed corpus, mutators have to generate the exact bytes to trigger the parser. Figure 3 shows the code coverage of G_{Bit} and G_S, with the green/red for covered/uncovered code lines, and ✓/✗ for covered/uncovered branches. G_{Bit} consistently reach this function and even partially traverse the internal parsing branches. In contrast, G_S fail in all trials. This result demonstrates that for fixed-length fields in OpenSSL, mutators that operate on bit and fixed-length bytes are significantly more effective at precisely modifying the required bytes to reach deep states.



```
 static int tls_parse_certificate_authorities(SSL *s,
    PACKET *pkt, ...) {
✓✗     if (!parse_ca_names(s, pkt))
        return 0;
       if (PACKET_remaining(pkt) != 0) {...}
       ...}
```

```
 static int tls_parse_certificate_authorities(SSL *s,
    PACKET *pkt, ...) {
       if (!parse_ca_names(s, pkt))
        return 0;
       if (PACKET_remaining(pkt) != 0) {...}
       ...}
```

Fig. 3. Coverage comparison of G_{Bit} (upper) and G_S (lower) in OpenSSL.



```
✗✓✓✓✓✗ while (j < reqStrSize && (reqStr[j] ==  ' ' ||
    reqStr[j] == '\t')) ++j;
✓✗✓✗    for (unsigned n = 0; n < resultCSeqMaxSize-1 && j <
     reqStrSize; ++n,++j)
     {...} // read everything up to the \r or \n as 'CSeq'
```

```
✗✓✓✓✓✓ while (j < reqStrSize && (reqStr[j] ==  ' ' ||
    reqStr[j] == '\t')) ++j;
✓✓✓✗    for (unsigned n = 0; n < resultCSeqMaxSize-1 && j <
     reqStrSize; ++n,++j)
     {...} // read everything up to the \r or \n as 'CSeq'
```

Fig. 4. Coverage comparison of G_{Bit} (upper) and G_S (lower) in Live555.

In contrast, Live555, which implements the RTSP protocol, handles variable-length fields such as the `CSeq` header using delimiter-based parsing. As shown in Figure 4, the parsing logic of this variable-length field differs significantly from OpenSSL's fixed-length extension identifier. Instead of expecting an exact byte count, Live555 parses headers like `CSeq` by iterating through the input buffer until it encounters a delimiter (`space` or `\t`), without enforcing strict byte-length constraints. Variable-length G_S outperform G_{Bit} in this case, achieving higher coverage across all runs. These results reinforce our earlier findings: aligning mutation granularity with message field structure significantly enhances fuzzing effectiveness.

Findings of RQ1: When the mutation granularity of the mutators is more aligned with the characteristics of a protocol's field lengths, the more effective fuzzing is. Specifically, (1) G_S is more aligned with text-based protocols, while G_{Bit}, G_{Byte}, G_{Word}, and G_{Dword} are more aligned with binary protocols, thus achieving statistically significantly *more* branch coverage than each other respectively; (2) on the mixed protocols, fuzzing effectiveness may depend on the lengths of the critical message fields.

TABLE V
BRANCHES COVERED BY DIFFERENT MUTATION LEVELS AND THE COMBINATION.

Impl.	Region	Byte	Region + Byte	Region Alone	Byte Alone	Region/Byte Common
BFTPD	**426**	457	477	5	55	427
Forked-daapd	**2157**	2320	2243	62	345	2148
Dnsmasq	854	859	847	5	28	905
Exim	3310	**3281**	3461	84	141	3281
LightFTP	**322**	372	366	2	25	353
Live555	**2663**	**2850**	2889	11	64	2899
ProFTPD	4579	4744	4686	318	513	4562
PureFTPD	**713**	1141	1030	5	462	865
Kamailio	6714	8729	8753	57	950	8904
DCMTK	**7086**	**7332**	7579	831	1198	6517
TinyDTLS	**237**	553	564	0	51	510
OpenSSH	**2759**	**3282**	3347	40	85	3300
OpenSSL	8650	10026	10038	92	118	10065

* Bold texts indicate statistically significance compared to Region+Byte.

TABLE VI
AVERAGE BRANCHES COVERED OF SOURCE FILES CONTAINING PARSER COMPONENTS UNDER DIFFERENT MUTATION LEVELS

Implementation	Parser Files	Region	Byte	P-value	$\hat{A}_{12}$
BFTPD	`commands.c`	142	**163**	0.14	0.80
Forked-daapd	`src/SMARTPLParser.c` `src/SMARTPLLexer.c`	214	**312**	<0.01	1.00
Dnsmasq	`rfc1035.c`	114	127	1.00	0.50
Exim	`src/parse.c`	89	**98**	<0.01	1.00
LightFTP	`ftpserv.c`	254	**304**	0.01	1.00
Live555	`RTSPServer.cpp` `RTSPCommon.cpp`	383	**537**	0.02	0.96
ProFTPD	`src/main.c`	246	**259**	<0.01	1.00
PureFTPD	`src/ftp_parser.c`	174	**216**	<0.01	1.00
Kamailio	`core/parser/msg_parser.c`	198	**319**	0.02	0.96
DCMTK	`dcmnet/libsrc/dimcmd.cc` `dcmnet/libsrc/dulparse.cc`	416	**545**	0.01	1.00
TinyDTLS	`dtls.c`	136	**342**	0.02	0.96
OpenSSH	`packet.c`	287	334	0.09	0.84
OpenSSL	`ssl/record/rec_layer_s3.c` `ssl/record/ssl3_record.c`	380	434	0.06	0.88

* Statistically significant differences are highlighted in bold.

V. RQ2: MUTATION LEVEL

Setup. AFLNET offers two mutation levels: (1) byte-level mutators ($M_1 \sim M_{16}$ in Table I), which mutate a seed as raw bytes; and (2) region-level mutators ($M_{17} \sim M_{20}$ in Table I), which mutate a seed as a sequence of messages. RQ2 aims to investigate how different these two mutation levels may perform in protocol fuzzing. To independently evaluate the impact of each mutation level, we enable one level while disabling the other during the fuzzing campaign. We measure the number of branches covered by the mutators of the level under evaluation. We also additionally enable both levels of mutators to observe the combination effect.

Results and Analysis. Table V shows the average number of branches covered by (1) region-level mutators ("Region"), (2) byte-level mutators ("Byte") and (3) region-level and byte-level both enabled ("Region+Byte"). We observe that byte-level mutators is more effective than region-level mutators in protocol fuzzing, achieving an average of 23.3% higher branch coverage across all the protocol implementations. When both mutation levels are enabled, 8 and 4 out of 13 protocol implementations respectively achieved statistically significantly more branch coverage, compared to region-level and byte-level mutation alone. It indicates enabling both mutation levels is better than single mutation level in protocol fuzzing.

To further analyze the differences between byte-level and region-level mutations, we compute the branches commonly covered by byte- and region-level ("Region/Byte Common") and covered by them alone ("Region Alone" and "Byte Alone"), based on the union of branches covered across all the repeated fuzzing campaigns. The results show that on the majority of protocol implementations (12 out of 13), byte-level and region-level mutators have their own respective contributions. But the coverage contribution of byte-level mutators ("Byte Alone") is substantially larger than that of region-level mutators ("Region Alone"), achieving 12.1× more branch coverage across all the studied protocol implementations.

Moreover, we analyze how different the byte-level and region-level mutators perform on the parsers of protocol implementations (which are the critical components in protocol implementations to read and sanitize inputs). To this end, we identified the source files relevant to the parsers of each protocol implementation and computed the average branches covered by each mutation level. The results in Table VI reveal that byte-level mutators exhibit superior coverage in parser-related code, covering 34% more branches on average than region-level mutators. It partially explains why byte-level mutators could contribute more than region-level ones in terms of overall branch coverage by stress-testing the parsers.

Case Study (Byte-level Mutations). To further understand the impact of byte-level mutations on protocol parsers, we selected Forked-daapd as a representative case, since this implementation features a highly complex parsing component. Unlike typical protocol parsers that handle only basic message parsing, Forked-daapd processes both API requests and their associated parameters while also parsing playlist files through an ANTLR-generated parser and lexer. These ANTLR-generated components rely on complex switch-case structures to match tokens and construct parse trees. When fuzzing this implementation using only byte-level mutators, the achieved coverage even surpasses AFLNET, demonstrating the effectiveness of byte-level mutators for such complex parsers can enhance overall coverage.

Moreover, as Table V shows, region-level mutators can cover branches that byte-level mutators fail to reach. It suggests that although region-level mutators achieve lower overall coverage, they are still capable of triggering complementary branches that byte-level mutators miss. Region-level mutators are particularly effective in exploring branches that depend on message sequences, such as field correlations across messages and state transitions. These branches occur in implementations that enforce constraints or dependencies between messages, requiring mutators that modify message structure while preserving message sequence relationships.

Case Study (Region-level Mutations). We examined the BFTPD branches that were exclusively covered by region-level mutators presented in Table V. Two modes of data connection

```
void command_port(char *params){
      ... // process port arguments
✓✓  if (pasv) {
✓✓      if(sock != 2) close(sock); // Close passive socket
        pasv = 0;} // Switch to active mode
```

```
void command_port(char *params){
      ... // process port arguments
✗✓  if (pasv) {
        if(sock != 2) close(sock); // Close passive socket
        pasv = 0;} // Switch to active mode
```

Fig. 5. Effect of region-level mutators (upper) and byte-level mutators (lower) on state-dependent code in BFTPD.

TABLE VII
BRANCHES COVERED BY AFLNET AND MUTATOR COMBINATIONS OF DIFFERENT MUTATION LEVELS WITH SPLICING ENABLED.

Implementation	Region + Splicing	Byte + Splicing	Region+Byte	Region+Byte +Splicing
BFTPD	**454 (7%)**	**487 (6%)**	477	485
Forked-daapd	2205 (2%)	2406 (4%)	2243	2271
Dnsmasq	871 (2%)	851 (-1%)	847	862
Exim	**3523 (6%)**	**3531 (8%)**	3461	3559
LightFTP	**344 (7%)**	**362 (-3%)**	366	363
Live555	2781 (4%)	**2928 (3%)**	2889	2895
ProFTPD	4577 (0%)	**5024 (6%)**	4686	4976
PureFTPD	772 (8%)	1106 (-3%)	1030	1023
Kamailio	7438 (11%)	8617 (-1%)	8753	9193
DCMTK	7015 (-1%)	7382 (1%)	7579	7688
TinyDTLS	412 (74%)	550 (-1%)	564	537
OpenSSH	2729 (-1%)	**3379 (3%)**	3347	3355
OpenSSL	8396 (-3%)	10051 (0%)	10038	10047

* Percentages in parentheses denote the coverage changes *w.r.t.* the cases without splicing in Table V. Bold texts indicate statistically significance.

exist between the client and the server: active and passive. In the active mode, the client issues a PORT command to inform the server of its IP address and an open port number, allowing the server to actively initiate the data connection. Conversely, in the passive mode, the server responds a PASV or EPRT command by sending its own ip address and an open port number, enabling the client to establish the connection instead. During a session, client may switch between these modes by issuing PASV or PORT commands. As shown in Figure 5, the highlighted code segment in BFTPD handles the transition from passive to active mode when processing the PORT command. This stateful logic is triggered only after the server *previously* received a PASV or EPRT command (pasv is set to 1). During our experiments, region-level mutators (upper) consistently covered all branches of the mode-switching logic shown in the figure across all runs. In contrast, byte-level mutators failed to explore these branches in any run.

Findings of RQ2: Region-level and byte-level mutators have complementary contribution on branch coverage. On average, byte-level mutators have 23.3% higher branch coverage than region-level mutations. Byte-level mutators are more effective in covering parser branches, while region-level mutators are more effective in exploring state-dependent branches. Nevertheless, enabling both mutation levels is better than single level in protocol fuzzing.

VI. RQ3: INTERPLAY WITH SPLICING

Setup. Splicing is a mutator by combining two parent inputs with the goal of finding a new interesting input. It is activated when the other mutators (including the byte-level and region-level mutators) cannot find any new interesting inputs. RQ3 aims to investigate how the byte-level mutators and region-level mutators would perform when splicing is enabled or disabled, and how would splicing affect protocol fuzzing.

Results. In Table VII, “Region+Splicing” and “Byte+Splicing” give the numbers of covered branches when splicing is enabled for region-level mutators and byte-level mutators, respectively. The numbers in parentheses characterize the impact on branch coverage when splicing is enabled, compared to the case when splicing is disabled (corresponding to “Region” and “Byte” in Table V). We also calculated the number of covered branches when the region- and byte-level mutators are both enabled without and with splicing (denoted by “Region+Byte” and “Region+Byte+Splicing”), respectively.

From Table VII, we observe that splicing can enhance the mutator effectiveness in some cases (*e.g.*, the region-level mutators on TinyDTLS). But the impact of splicing could lead to gains as well as losses on branch coverage for region-level or byte-level mutators. In most cases, the gains or losses do not have statistically significance. Moreover, the differences of covered branches between “Region+Byte“ and “Region+Byte+Splicing” are not statistically significant on all the studied protocol implementations. It indicates that enabling splicing does not lead to statistically significantly more or less branch coverage for protocol fuzzing.

Case study. Further analysis of the coverage reports reveals that the large-scale mutations introduced by splicing are particularly effective at exploring *parser* and *error-handling* branches. Consequently, when region-level or byte-level mutators demonstrate limited effectiveness in covering these specific branches, splicing can enhance the overall fuzzing performance. To illustrate this, we analyzed branches (Figure 6) in the Exim implementation that were covered by both byte-level and region-level mutators with splicing enabled, but were missed by both when it was disabled. These branches correspond to the error-handling logic of email address verification. When an address is checked and determined to be invalid, execution enters the “hard failure” branch where rc == FAIL. Without splicing, neither the byte-level nor region-level mutators triggered this branch in any run. Once splicing was enabled, however, the same branch was triggered in 80% of runs, demonstrating that the large-scale recombination introduced by splicing effectively enhances exploration of these error-handling paths.

However, the benefit of splicing is not universal. In validation-heavy protocols like DTLS and TLS, splicing often reduces total branch coverage. We examined the branch coverage reports and identified that mutations introduced by splicing often fail to pass through critical validation branches. Consequently, this prevents further exploration of deeper branches, thereby reducing overall coverage.

```
int varify_address(address_item * vaddr, FILE * fp){...
✓✓ if (rc == FAIL) { ... // Hard failure
✓✓  if (!full_info){ // Quick exit with error
       yield = copy_error(vaddr, addr, FAIL);
       goto out;}}}
```

```
int varify_address(address_item * vaddr, FILE * fp){...
✗✓ if (rc == FAIL) { ... // Hard failure
    if (!full_info){ // Quick exit with error
       yield = copy_error(vaddr, addr, FAIL);
       goto out;}}}
```

Fig. 6. Effect of splicing on error-handling code in Exim.

```
    static int dtls_check_tls_extension(...) {
      ... // init and parse extension list length
✓✗ if (data_length < j) goto error;
   while (data_length) {
   if (data_length < sizeof(uint16_t) * 2) goto error;
      ... // update pointer and data_length
   if (data_length < j) goto error;
   switch (i) {...}}} // TLS extension type cases
```

```
    static int dtls_check_tls_extension(...) {
      ... // init and parse extension list length
✓✓ if (data_length < j) goto error;
✓✓ while (data_length) {
✓✓ if (data_length < sizeof(uint16_t) * 2) goto error;
      ... // update pointer and data_length
✓✓ if (data_length < j) goto error;
✓✓✓✓✓✓  switch (i) {...}}} // TLS extension type cases
```

Fig. 7. Coverage comparison on the code in TinyDTLS with heavy validations with and without splicing.

To illustrate this limitation, we present a case study on TinyDTLS [33] (Figure 7), comparing branch coverage of an extension validation function with splicing enabled (upper) and disabled. This function contains a series of validation, each leading to an error-handling path upon failure. Across multiple runs, enabling splicing frequently produced inputs that failed early validations, and even successful inputs typically triggered only a single case in the final switch statement. In contrast, disabling splicing allowed inputs to pass all validations and cover all switch cases across every experimental rounds. These findings indicate that large-scale mutations introduced by splicing are not well-suited for strict, validation-heavy protocols. Disabling splicing allows for more effective mutations for these protocol implementations, thereby increasing the likelihood of covering deeper branches, such as the switch cases in Figure 7.

Findings of RQ3: Splicing could lead to gains or losses on branch coverage for region-level and byte-level mutators. It is effective in exploring error-handling and parser branches but less effective on protocol implementations with heavy validations. However, splicing does not lead to statistically significant change on overall fuzzing performance in terms of branch coverage.

VII. Characteristics-Aware Mutator Selection

In this section, we introduce a characteristics-aware mutator selection strategy to apply our findings distilled from RQ1∼RQ3. It aims to select an appropriate (sub)set of mutators from AFLNet based on the characteristics of a protocol to improve fuzzing performance. We apply this strategy on all the 13 protocol implementations from ProFuzzBench. To mitigate the potential biases, we additionally include two new protocols MQTT and IPP (and their implementations Mosquitto and IPPsample).

Mutator Selection Strategy. The findings of RQ1 indicate that aligning mutation granularity with the characteristics of a protocol's field lengths is crucial for effective fuzzing. Thus, we select the mutators that operate on random-length bytes (G_S) for fuzzing text-based protocols, and the mutators that operate on bit and fixed-length bytes (G_{Bit}, G_{Byte}, G_{Word} and G_{Dword}) for fuzzing binary protocols. For mixed protocols, we additionally consider the lengths of the critical fields. For example, MQTT is a binary protocol, we select G_{Bit}, G_{Byte}, G_{Word} and G_{Dword} for fuzzing its implementation Mosquitto; IPP is a mixed protocol whose critical field command is a small fixed-length field, we also select G_{Bit}, G_{Byte}, G_{Word} and G_{Dword} for fuzzing its implementation IPPsample. Forked-daapd is an exceptional case as its messages are composed of API calls rather than the message fields analyzed in RQ1. Thus, we select all the byte-level mutators at different granularities G_{Bit}, G_{Byte}, G_{Word}, G_{Dword} and G_S.

The findings of RQ2 indicate that enabling both region-level mutators with byte-level mutators is better than single level because they have complementary contributions. Thus, we enable both levels simultaneously, with byte-level mutators configured according to RQ1. The findings of RQ3 indicate that splicing does not lead to statistically significant change on overall fuzzing performance in terms of branch coverage; but splicing shows effectiveness in covering parser and error-handling related code. Thus, we experiment with two setups, one with splicing disabled and another with splicing enabled.

Table VIII shows the selected mutators for each protocol implementation (denoted by "Selected Mutators") according to our strategy. Thus, in this experiment, we evaluated:

- AFLNet: the original AFLNet (all mutators and splicing are enabled); ("AFLNet" in Table VIII)
- AFLNet-α: a variant of AFLNet that incorporates the selected mutators with splicing disabled ("AFLNet-α" in Table VIII);
- AFLNet-β: a variant of AFLNet that incorporates the selected mutators with splicing enabled ("AFLNet-β" in Table VIII);

We compare the fuzzing performance between AFLNet, AFLNet-α and AFLNet-β to evaluate our characteristic-aware mutator selection strategy. We follow the same experimental setup in Section III.

Coverage. Table VIII gives the results of covered branches. Both AFLNet-α and AFLNet-β outperform AFLNet on

TABLE VIII
BRANCH COVERAGE OF CHARACTERISTIC-AWARE MUTATOR SELECTION STRATEGY.

Implementation	Selected Mutators	AFLNet	AFLNet-α	Imp.	P-value	$\hat{A}_{12}$	AFLNet-β	Imp.	P-value	$\hat{A}_{12}$
BFTPD	G_S, T_{Region}	485	481	-0.70%	0.34	0.30	484	-0.17%	0.92	0.46
Forked-daapd	G_{Bit}, G_{Byte}, G_{Word}, G_{Dword}, G_S, T_{Region}	2271	2384	4.98%	<0.01	1.00	2383	4.95%	<0.01	1.00
Dnsmasq	G_S, T_{Region}	862	898	4.25%	0.22	0.76	971	12.67%	<0.01	1.00
Exim	G_S, T_{Region}	3559	3641	2.30%	0.21	0.76	3665	3.00%	0.02	0.96
LightFTP	G_S, T_{Region}	363	371	2.15%	<0.01	1.00	370	1.93%	<0.01	1.00
Live555	G_S, T_{Region}	2895	2896	0.03%	0.69	0.60	2909	0.47%	0.46	0.66
ProFTPD	G_S, T_{Region}	4795	4869	1.56%	0.15	0.80	5032	4.95%	0.15	0.80
PureFTPD	G_S, T_{Region}	1023	1060	3.62%	0.69	0.60	1034	1.08%	1.00	0.52
Kamailio	G_S, T_{Region}	9193	9943	8.16%	1.00	0.48	9827	6.89%	0.69	0.40
DCMTK	G_{Bit}, G_{Byte}, G_{Word}, G_{Dword}, T_{Region}	7688	8118	5.60%	0.10	0.84	7691	0.04%	0.55	0.64
TinyDTLS	G_{Bit}, G_{Byte}, G_{Word}, G_{Dword}, T_{Region}	537	562	4.77%	0.60	0.62	588	9.62%	0.10	0.84
OpenSSH	G_{Bit}, G_{Byte}, G_{Word}, G_{Dword}, T_{Region}	3355	3364	0.27%	0.15	0.80	3372	0.50%	0.09	0.84
OpenSSL	G_{Bit}, G_{Byte}, G_{Word}, G_{Dword}, T_{Region}	10047	10196	1.48%	0.01	1.00	10182	1.35%	<0.01	1.00
IPPsample	G_{Bit}, G_{Byte}, G_{Word}, G_{Dword}, T_{Region}	3056	3197	4.61%	0.01	1.00	3183	4.16%	0.01	1.00
Mosquitto	G_{Bit}, G_{Byte}, G_{Word}, G_{Dword}, T_{Region}	2176	2235	2.71%	0.01	1.00	2210	1.56%	0.06	0.88
				Avg: 3.05%				**Avg**: 3.53%		

* T_{Region} denotes the region-level mutators (M_{17}, …, M_{20} in Table I).

TABLE IX
BUG FINDING RESULTS

Unique Bug	AFLNet	AFLNet-α	AFLNet-β
DCMTK-1	5/5	5/5	5/5
Dnsmasq-1	2/5	2/5	4/5
TinyDTLS-1	-	-	1/5
TinyDTLS-2	-	-	1/5
TinyDTLS-3	4/5	5/5	5/5
TinyDTLS-4	-	3/5	3/5
TinyDTLS-5	5/5	5/5	5/5
TinyDTLS-6	-	1/5	3/5
TinyDTLS-7	5/5	5/5	5/5
TinyDTLS-8	2/5	-	1/5
TinyDTLS-9	-	2/5	2/5
TinyDTLS-10	-	1/5	-
Mosquitto-1	5/5	5/5	5/5
Mosquitto-2	-	2/5	2/5
IPPsample-1	-	1/5	1/5
IPPsample-2	5/5	-	5/5
#Found Bugs	8	12	15

nearly all (14 out of 15) the tested protocol implementations. In detail, AFLNet-α and AFLNet-β on average cover 3.05% and 3.53% more branches than AFLNet, respectively, showing the effectiveness of selected mutators (with splicing enabled or disabled). We use the same methods as in previous sections to assess the statistical significance and effect size of these coverage results. Among the 15 protocol implementations, AFLNet-α and AFLNet-β achieved statistically significantly more coverage on 5 and 6 subjects, respectively.

Furthermore, we conducted controlled experiments with misaligned mutation granularity on the two new protocols implementations IPPsample and Mosquitto. We replace the selected mutators G_{Bit}, G_{Byte}, G_{Word} and G_{Dword} (aligned) by G_S (misaligned). The results are presented in Table X. The results are negative: (1) the coverage in all cases drops below the original AFLNet; (2) the coverage in all cases is statistically significant less than AFLNet-α and AFLNet-β, respectively. The results clearly suggest that misaligned mutation granularity could substantially reduce fuzzing effectiveness, supporting our findings of RQ1.

TABLE X
NEGATIVE RESULTS OF USING MISALIGNED MUTATION GRANULARITY.

Implementation	G_S, T_{Region} w/o Splicing		G_S, T_{Region} w/ Splicing	
	Branches	P-value	Branches	P-value
IPPsample	2918 (-8.7%)	<0.01	2901 (-8.9%)	<0.01
Mosquitto	2097 (-6.2%)	<0.01	2088 (-5.5%)	<0.01

* The parenthesized percentages indicate the coverage decrease compared to AFLNet-α and AFLNet-β in Table VIII.

Bug Finding. Table IX presents all the unique bugs found by AFLNet, AFLNet-α and AFLNet-β during the five repeated rounds of fuzzing campaign. We also give the number of rounds in which a bug was found. For example, 5/5 indicates a bug was found in each of the five rounds. Note that each found bug has been manually verified and reproduced on the corresponding protocol implementation. In total, AFLNet-α found 12 unique bugs, AFLNet-β found 15, and AFLNet found only 8. Specifically, AFLNet failed to find 8 bugs, which were successfully found by AFLNet-α or AFLNet-β. By comparing AFLNet-α and AFLNet-β, we can see that splicing is useful for finding additional bugs although it may not be able to significantly improve coverage. By comparing AFLNet and AFLNet-β, we can see that employing the mutators with aligned mutation granularity is crucial.

To illustrate that aligned mutators significantly enhance bug discovery capability, we investigated Mosquitto-2 and IPPsample-1, which is only missed by AFLNet. These two bugs share similar causes. When the fuzzer sends a malformed `CONNACK` command (configured by `MessageType` field with 1-byte length) to the server, a null pointer dereference occurs. A similar null pointer dereference bug of IPPsample is triggered when an `IPP_OP_CANCEL_JOBS` command (configured by `operation_id` field with 2-byte length) is sent to the server without a corresponding local `AuthType` configuration. Neither of these malformed command codes appears in the initial seed corpus. Both bugs are triggered by the mutators that operate on bit and fixed-length bytes, which are aligned with the lengths of these command fields.

We further investigate IPPsample-2, the bug only missed by AFLNet-α. It arises during HTTP header parsing, which are used to encapsulate IPP messages. When an HTTP request contains multiple identical header fields (in the form of variable-length texts), the server attempts to combine them, leading to a stack-use-after-scope bug. AFLNet-α failed to uncover this bug due to the absence of G_S mutators. However, once splicing was enabled, AFLNet-β successfully triggered the bug in all runs, as the large-scale mutations introduced by splicing effectively compensate its ability to discover vulnerabilities triggered by these variable-length fields.

Conclusion: The characteristic-aware mutator selection strategy inspired by our study's findings improves fuzzing performance. It increases branch coverage on average by 3.53% with splicing, and 3.05% without splicing, respectively. It achieves coverage improvements in 14 out of 15 targets, with up to 12.67% of coverage improvement. It also significantly enhances bug finding capability, finding 8 additional unique bugs that AFLNet fails to uncover.

VIII. Discussion

Generability of Our Findings. Our study investigates the effectiveness of the mutators in AFLNet. Many protocol fuzzers, *e.g.*, StateAFL [30], ChatAFL [5] and NSFuzz [7] are built on AFLNet without modifying the mutators. Thus, our proposed mutator selection strategy is orthogonal to the improvements made by these subsequent protocol fuzzers [5], [7], [30]. Our findings and mutator selection strategy could be directly applied to these fuzzers. Other mutation-based protocol fuzzers like SGFuzz [34] (built on LibFuzzer [35]) also use similar mutators in AFLNet. Thus, our findings could also benefit them. Generation-based fuzzers [17], [18], [28], [36]–[42] provide mutators for fuzzing in state/data models. While our findings may not directly apply to generation-based fuzzers, the insights from our study — particularly the relationship between protocol characteristics and mutator performance — remain valuable. These findings can guide the design of more effective mutators in generation-based fuzzers, *e.g.*, incorporating region-level mutators and splicing.

Suggestions for Mutation-based Protocol Fuzzing. When fuzzing a new protocol implementation, our findings can guide mutator selection from three dimensions. (1) We recommend selecting G_{Bit}, G_{Byte}, G_{Word} and G_{Dword} for binary protocols and G_S for text-based protocols; for mixed protocols, the selection of mutators should also consider the field lengths of critical message fields. (2) By default, enabling both byte-level and region-level mutators to obtain complementary benefits. If the goal is to stress-test the protocol parser, only enabling byte-level mutators is preferred. If the goal is to explore protocol state-dependent code, only enabling region-level mutators is preferred. (3) Splicing can be enabled or disabled for protocol fuzzing. If the goal is to test more error-handling and parser-related code, enabling splicing is preferred. If a protocol implementation includes strict validation checks, disabling splicing is preferred to avoid generating inputs which are likely to be rejected by the validation checks.

IX. Threats to Validity

Internal Validity. Our results may be affected by the inherent randomness of fuzzing. To mitigate this threat, we evaluate all compared settings under identical environments and with the same testing budgets, and we use repeated experiments to reduce the influence of randomness.

External Validity. The findings of our work may not generalize to all the protocol implementations. To mitigate this, we conducted our experiments on ProFuzzBench, which includes 13 widely adopted and diverse protocol implementations. Moreover, we further included two new protocols which are not in ProFuzzBench to validate the generalizability of our findings. Although our study is only conducted on AFLNet, AFLNet is the most representative mutation-based protocol fuzzer.

Construct Validity. Our evaluation uses commonly adopted metrics in protocol fuzzing, such as branch coverage and vulnerability exposure. However, these metrics may not fully capture all aspects of fuzzing efficiency and effectiveness. To mitigate this threat, we additionally conduct controlled experiments as complementary evaluation (in Table X) and keep the metrics consistent across all compared settings.

X. Related Work

Empirical Studies on Mutators and Mutation Strategy. Kukucka et al. [43] empirically evaluate the performance of AFL++ [44] mutators, including havoc, splicing, and RedQueen, across a diverse set of general-purpose targets. Their key finding is that disabling certain mutators can improve branch coverage and bug finding. They find no statistically significant difference in branch coverage across individual havoc mutators on Magma [45] benchmark. Our work focuses on *protocol* fuzzing and reveals that mutator effectiveness varies significantly across protocol implementations. We further explore how different protocol characteristics influence mutator effectiveness. Wu et al. [46] focus on the havoc mutation strategy commonly employed by the coverage-guided fuzzers. They examine the overall efficacy of the havoc strategy and found that pure havoc achieves superior edge coverage and bug detection. We analyze mutators in protocol fuzzer around three dimensions: mutation granularity, mutation level, and their interplay with splicing. MOPT [47] focuses on optimizing mutation scheduling to improve the efficiency of vulnerability discovery. Our work examines how the characteristics of protocol implementations influence the effectiveness of mutators. Building on our findings, we design a characteristic-aware mutator selection that further improves protocol fuzzing performance.

Evaluating Protocol Fuzzers. Much work has been proposed for protocol fuzzing [5]–[9], [12]–[14], [17], [18], [26], [30], [34], [48]–[53]. However, there are relatively few studies focus on evaluating such fuzzers. ProFuzzBench [29] is a comprehensive benchmark for evaluating protocol fuzzers, containing

13 representative protocol implementations. It evaluates the performance of three representative fuzzers, AFLNWE [54], AFLNET [6] and STATEAFL [30]. Liu et al. [55] systematically evaluate three state selection algorithms built on AFLNET to study their effectiveness for stateful protocol fuzzing. Meng et al. [20] empirically analyze the effectiveness of state feedback in AFLNET and the impact of AFLNET's different seed selection strategies. Different from prior work, our study focuses on studying the effectiveness of the *mutators* in AFLNET from the perspective of protocol characteristics to provide insights that could improve the efficacy of protocol fuzzing.

XI. Conclusion

In this paper, we present the first study on the effectiveness of mutators in protocol fuzzing. By examining mutation granularity, mutation level, and their interplay with splicing, we obtain new interesting findings, most of which are unknown or unclear for the community before. Specifically, our findings reveal how protocol characteristics influence mutator effectiveness. Guided by findings of our study, we design a mutator selection strategy. Evaluations confirm that this strategy can improve both branch coverage and bug-finding capability compared to the baseline tool AFLNET.

XII. Data Availability

We have made all the artifacts in our study publicly available at *https://github.com/ecnusse/AFLNet_Mutators_Study*.

Acknowledgment

We thank the anonymous SANER reviewers for their valuable feedback. This work was supported in part by National Key Research and Development Program (Grant 2022YFB3104002), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and the Chenguang Program of Shanghai Education Development Foundation and Shanghai Municipal Education Commission (Grant 23CGA33).

References

[1] A. S. Tanenbaum, "Network protocols," *ACM Computing Surveys (CSUR)*, vol. 13, no. 4, pp. 453–489, 1981.

[2] Z. Hu and Z. Pan, "A systematic review of network protocol fuzzing techniques," in *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, vol. 4. IEEE, 2021, pp. 1000–1005.

[3] Z. Zhang, H. Zhang, J. Zhao, and Y. Yin, "A survey on the development of network protocol fuzzing techniques," *Electronics*, vol. 12, no. 13, p. 2904, 2023.

[4] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, ser. SPIN '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 103–122.

[5] 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.

[6] V. 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] S. Qin, F. Hu, Z. Ma, B. Zhao, T. Yin, and C. Zhang, "NSFuzz: Towards efficient and state-aware network service fuzzing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–26, 2023.

[8] P. C. Amusuo, R. A. C. Méndez, Z. Xu, A. Machiry, and J. C. Davis, "Systematically detecting packet validation vulnerabilities in embedded network stacks," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 926–938.

[9] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ics protocol," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[10] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*, 2018, pp. 1–15.

[11] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 484–500.

[12] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, 2021, pp. 337–350.

[13] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-Net: network fuzzing with incremental snapshots," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 166–180.

[14] J. Li, S. Li, G. Sun, T. Chen, and H. Yu, "Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 2673–2687, 2022.

[15] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 19–36.

[16] F. He, W. Yang, B. Cui, and J. Cui, "Intelligent fuzzing algorithm for 5g nas protocol based on predefined rules," in *2022 International Conference on Computer Communications and Networks (ICCCN)*, 2022, pp. 1–7.

[17] (2025) Peach. A fuzzing framework which uses a DSL for building fuzzers and an observer based architecture to execute and monitor them. https://github.com/MozillaSecurity/peach.git.

[18] (2025) Boofuzz. A fork and successor of the Sulley Fuzzing Framework. https://github.com/jtpereyda/boofuzz.git.

[19] (2025) American fuzzy lop (AFL) fuzzer. https://lcamtuf.coredump.cx/afl/technical_details.txt.

[20] R. Meng, V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNet five years later: On coverage-guided protocol fuzzing," *IEEE Trans. Softw. Eng.*, vol. 51, no. 4, p. 960–974, Jan. 2025.

[21] N. Group, "The challenges of fuzzing 5g protocols," *5G Secur. Smart Environ.*, 2021.

[22] M. Nedyak, "How to hack medical imaging applications via dicom," in *Hack In The Box Security Conference*, 2020.

[23] (2025) Webinar: Demystifying a current trend - security fuzz testing in the context of iso/sae 21434. https://youtu.be/HDfkD67UUSw.

[24] (2025) A use-after-free vulnerability discovered by aflnet. https://github.com/project-chip/connectedhomeip/pull/33148/.

[25] (2025) A critical memory (heap) leak vulnerability discovered by aflnet. https://github.com/project-chip/connectedhomeip/pull/32970/.

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

[27] S. Jiang, Y. Zhang, J. Li, H. Yu, L. Luo, and G. Sun, "A survey of network protocol fuzzing: Model, techniques and directions," *arXiv preprint arXiv:2402.17394*, 2024.

[28] X. Zhang, C. Zhang, X. Li, Z. Du, B. Mao, Y. Li, Y. Zheng, Y. Li, L. Pan, Y. Liu, and R. Deng, "A survey of protocol fuzzing," *ACM Comput. Surv.*, vol. 57, no. 2, Oct. 2024.

[29] R. Natella and V.-T. Pham, "ProFuzzBench: A benchmark for stateful protocol fuzzing," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, 2021, pp. 662–665.

[30] R. Natella, "StateAFL: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, p. 191, 2022.

[31] (2025) OpenSSL. A robust, commercial-grade, full-featured Open Source Toolkit for the TLS (formerly SSL), DTLS and QUIC protocols. https://github.com/openssl/openssl.

[32] (2025) Live555. A complete RTSP server application. http://www.live555.com/.

[33] (2025) tinydtls. A library for Datagram Transport Layer Security (DTLS) covering both the client and the server state machine. https://projects.eclipse.org/projects/iot.tinydtls.

[34] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3255–3272.

[35] (2025) libFuzzer. A library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[36] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 535–552.

[37] G. S. Reen and C. Rossow, "DPIFuzz: A differential fuzzing framework to detect dpi elusion strategies for quic," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 332–344.

[38] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1492–1504.

[39] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, "SweynTooth: Unleashing mayhem over bluetooth low energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 911–925.

[40] M. E. Garbelini, C. Wang, and S. Chattopadhyay, "Greyhound: Directed greybox wi-fi fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 817–834, 2022.

[41] M. E. Garbelini, V. Bedi, S. Chattopadhyay, S. Sun, and E. Kurniawan, "BrakTooth: Causing havoc on bluetooth link manager via directed fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1025–1042.

[42] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, "Automata-based automated detection of state machine bugs in protocol implementations." in *NDSS*, 2023.

[43] J. Kukucka, L. Pina, P. Ammann, and J. Bell, "An empirical examination of fuzzer mutator performance," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1631–1642.

[44] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[45] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[46] 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 (ICSE)*, 2022, pp. 1634–1645.

[47] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 1949–1966.

[48] 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)*, 2023, pp. 4481–4498.

[49] D. Maier, O. Bittner, M. Munier, and J. Beier, "FitM: binary-only coverage-guided fuzzing for stateful network protocols," in *Workshop on Binary Analysis Research (BAR)*, vol. 2022, 2022.

[50] F. Zuo, Z. Luo, J. Yu, Z. Liu, and Y. Jiang, "PAVFuzz: State-sensitive fuzz testing of protocols in autonomous vehicles," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 823–828.

[51] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, "ICS protocol fuzzing: Coverage guided packet crack and generation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[52] Y. Zhai, R. Ma, Z. Zhang, S. Zhao, and Y. Yang, "MSNFuzz: Multi-criteria state-sensitive network protocols fuzzing," *Computers & Security*, p. 104621, 2025.

[53] X. Song, Y. Zeng, J. Wu, H. Li, C. Zuo, Q. Zhao, and S. Guo, "CSFuzzer: A grey-box fuzzer for network protocol using context-aware state feedback," *Computers & Security*, p. 104581, 2025.

[54] AFLnwe. A stateless coverage-guided fuzzer. https://github.com/thuanpv/aflnwe.

[55] D. Liu, V.-T. Pham, G. Ernst, T. Murray, and B. I. Rubinstein, "State selection algorithms and their impact on the performance of stateful network protocol fuzzing," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 720–730.