Automatically Fuzzing Routing Protocols

Martin Vivian
UCLouvain, Belgium
martin.vivian@uclouvain.be

Olivier Bonaventure
UCLouvain & WELRI, Belgium
olivier.bonaventure@uclouvain.be

***Abstract*—Routing protocols are key protocols on the public Internet and in enterprise networks. Routers exchange control messages to compute their routing tables. A bug in the processing of such messages could cause routers to crash or reboot, leading to wide scale disruptions. We propose a generic fuzzing approach that is able to automatically predict the structure of routing packets based on collected packet traces. We automatically extract header length, checksum and the Type-Length-Value structure of such protocols. We leverage this inferred structure to generate fuzzed routing messages and we apply this approach to three very different protocols: IS-IS, EIGRP and Babel. Our experiments reveal three major problems in an open-source implementation of EIGRP that have now been fixed by the maintainers and one problem in the IS-IS implementation.**

***Index Terms*—protocol fuzzing, routing protocols, TLV inference, packet structure inference, network protocol**

I. Introduction

Since the early days of the Internet, a wide range of standardized and proprietary protocols have been designed and implemented. The message-based protocols exchange independent messages, typically over UDP or directly over IP or link-layer protocols. These include application layer protocols such as RTP, DNS, DTLS, QUIC and control-plane protocols such as OSPF, IS-IS, Babel, RIP, RSVP, IGMP, ... The stream-based protocols exchange messages over a TCP connection. These include application layer protocols like HTTP, SMTP, POP, ... and control-plane protocols like BGP.

In this paper, we focus on the message-based protocols and leave the stream-based protocols for further work. A message is a discrete unit of communication whose boundaries and meaning are defined by a protocol. We focus on the control-plane protocols, because these protocols are crucial for the operation of enterprise and Internet Service Provider networks. If a router crashes because it received a malformed packet, this could have a huge impact on the global Internet. Such events appeared in the past [1]–[3]. For example, a BGP router will reset a BGP session with its peer if it receives a malformed packet. This unfortunately happens regularly [4]. In 2010, an experiment conducted by researchers with a new BGP attribute caused high-end BGP routers to reset their sessions with peers [3]. In 1990, the AT&T network fell due to a software failure [1]. The CVE database reports multiple vulnerabilities that have affected the parsing of OSPF (CVE-2024-20313, CVE-2022-20823, CVE-2017-3224, CVE-2004-1454), EIGRP (CVE-2002-2208, CVE-2005-4436) or IS-IS (CVE-2019-1910, CVE-2023-20169, CVE-2024-20312, CVE-2024-20406) packets.

Fuzzing is a widely used methodology for discovering vulnerabilities [5], introduced by Miller et al. [6]. It has been applied extensively to uncover packet parsing flaws in many protocols [7], [8]. Fuzzers are often classified by the level of visibility they have into the target: white-box fuzzers [9]–[11] have full source access, grey-box fuzzers [12], [13] rely on partial information, and black-box fuzzers [14], [15] observe only inputs and outputs.

To improve efficiency, many fuzzers use message templates to guide smarter mutations. Polymorph [16] and mqtt_fuzzing [17] are stateless fuzzers that parse messages using `tshark` or `scapy`, but their reliance on known protocol formats limits applicability to proprietary protocols. Other fuzzers learn protocol behaviors directly from network traces (e.g., [18], [15]), though they are typically evaluated on simple protocols and retain only a basic understanding of the system.

In this paper, we focus on fuzzing the packets that are exchanged by routing protocols over a network. Several approaches have been proposed to fuzz specific protocols [19], [20], [21]. We focus on a more generic problem and consider the following research question: **"By observing the packets exchanged by routers, is it possible to develop a generic fuzzer which can take into account the protocol structure without explicit knowledge of the protocol ?"** Our approach is generic as it can be applied to a range of protocols, including proprietary and non-documented ones.

At a high level, our approach combines structure inference with structure-aware fuzzing. After automatically identifying structured fields such as Type-Length-Value (TLV) elements from packet traces, we generate mutated packets while preserving the overall structure of the protocol. In particular, we prioritize mutations of the Value fields, while keeping Type and Length fields mostly consistent. To maximize the likelihood that mutated packets are processed by the target, we also adjust inferred header fields such as packet lengths and checksums. This enables our fuzzer to bypass early validation checks and explore deeper code paths in protocol implementations.

This paper is organized as follows. We first provide some background on routing protocols and the packets that they exchange in Section II. We then explain in Section III the challenges in developing such a generic fuzzer. Section IV contains our first contribution. We propose several algorithms that allow us to automatically infer the structure of routing

ISBN 978-3-903176-82-9 © 2026 IFIP

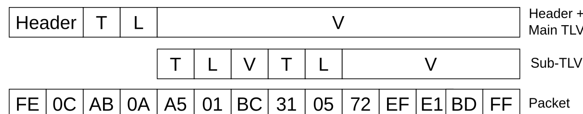

Fig. 1. Example of TLV and sub-TLVs

protocol messages. We start by locating the length field in the header, then detecting the location and type of checksum. Finally, our algorithm infers the different types of *Type, Length, Value* that are used by these routing protocols. We implement this inference technique in **PROtocol Structure Extractor** (**PROSE**). Section V describes our fuzzer, **Mutation & Stress Engine** (**MUSE**), which takes as input the information extracted by **PROSE** to create fuzzed packets. We evaluate the performance of **PROSE** with three different routing protocols (EIGRP, Babel and IS-IS) in Section VI. These experiments demonstrate that our generic algorithm works correctly with these protocols. In Section VII, we demonstrate that **MUSE** was able to detect three unknown vulnerabilities in the EIGRP implementation of FRRouting and one vulnerability in the IS-IS implementation. We compare **PROSE** and **MUSE** to related work in Section VIII.

II. Background

We first briefly describe in section II-A the messages exchanged by routing protocols and their structure. Section II-B describes how we collected the dataset used in this paper.

A. Messages exchanged by routing protocols

Packet-based routing protocols like OSPF, EIGRP, IS-IS or Babel have been designed with extensibility in mind. The messages that they exchange are composed of a header followed by a variable length field which is encoded as a sequence of Type, Length, Value (TLV) triplets. Figure 1 illustrates this with a typical packet. In addition, this figure shows the sub-TLVs, which are TLVs contained within the V field. In this example, we have one TLV with two sub-TLVs.

These protocols use different headers. OSPFv3 uses a 16 bytes header which contains a length field, a type field (to distinguish between Hello, Database Description and Linkstate packets), the identifications of the router, area and instance and a checksum. EIGRP [22] uses a 20 bytes header which contains a checksum, flags, sequence and acknowledgment numbers and AS number. IS-IS [23] uses an eight bytes header which contains mainly a protocol identifier, a lifetime, a version, a checksum and a PDU type. The Babel routing protocol is a more recent routing protocol [24]. It uses an 8 bytes header containing a magic number (42), a version field (2), a two bytes long length field and a nonce.

To protect the packet from transmission errors, these routing protocols use checksums. OSPFv3 and EIGRP use the Internet checksum [25] while IS-IS uses the Fletcher checksum [26].

These four routing protocols use TLV triplets to encode most of the information they exchange. EIGRP [22], [27] uses two bytes to encode the Type, 2 bytes to encode the length of the TLV, including the four bytes for the Type and Length information. IS-IS [23], [28], [29] and Babel [24] encode their Type and Length fields using one byte each. IS-IS also supports Sub-TLV, e.g. for the Extended Reachability TLVs.

Among the routing protocols standardized by the Internet Engineering Task Force, the Routing Information Protocol (RIP) [30] is a rare example of a protocol that does not use Type, Length, Value triplets. It uses a 32 bits header followed by a series of 20 bytes long routing series. However, given the well-known convergence problems of this basic distance vector protocol, it is rarely used nowadays.

B. Routing protocols dataset

To conduct the experiments described in the paper and enable other researchers to reproduce and expand them, we have collected a dataset of routing protocol packets. For this, we leverage the `containerlab` package [1]. This package is often used by network operators to create labs and validate network configurations before putting them in production. In this context, a “lab” refers to a fully emulated network environment composed of interconnected virtual routers used to reproduce realistic routing protocol interactions. Containerlab builds upon Linux containers. It supports about forty different network operating systems (NOS), both open-source and commercial ones. Each of these NOS is packaged inside a container or a virtual machine. Containerlab provides the tools that allow the creation of virtual topologies among these containers and manage them. We use the FRRouting [31] open-source package for our experiments. This is a stable and well-maintained implementation of various routing protocols.

We use Containerlab to create a five routers topology shown in Figure 2 and collect all the packets exchanged between `router1` and `router2`. For each studied routing protocol, we configure it to use as many extensions as possible in order to collect packets containing as many features as possible. We then use Wireshark to filter the trace and remove the packets exchanged by other protocols (e.g. `ARP`, `ICMP`, ...) than the studied ones. Unfortunately, EIGRP provides very few extensions, and these do not generate more than simple “Hello” messages. To obtain a more meaningful dataset, during our measurements, we ran a script that periodically changed the EIGRP router configuration weights, allowing us to capture a wider variety of messages.

III. Challenges in inferring packet structures

Internet and routing protocols use different types of packet structures. If we look at the TLV parts of a packet in more detail, the Length field does not always reflect the actual size of the Value, and in many protocols, including IS-IS, TLVs may be nested inside one another, which complicates boundary detection. Distinguishing the Type and Length fields can therefore be difficult, and frame headers or other protocol-layer metadata may further interfere with parsing. A single packet can contain multiple TLVs, sometimes different Types

[1] https://containerlab.dev/

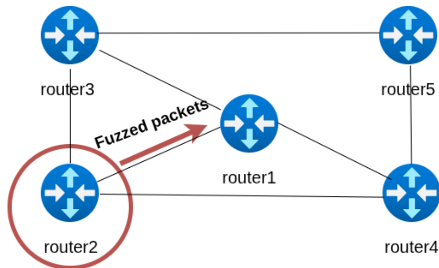

Fig. 2. Experiment Topology

and both the order and the count of those TLVs can vary from one protocol to another. On top of that, there is no fixed alignment. TLVs of varying sizes are often concatenated without padding, so offsets shift throughout the packet and make reliable parsing even harder. In the remainder of this paper, we adopt the following definitions and assumptions.

A. Definitions

The Type (T) field specifies the kind of information carried by this part of the message. The Length (L) field indicates the size of the Value (V) field, usually expressed in bytes. The Value (V) field contains the actual data corresponding to the specified type.

This TLV definition is generic and can vary between protocols. For example, EIGRP uses the L field to indicate the size of the entire TLV, not just the V field in contrast with IS-IS. Few protocols, such as Diameter [32] for backward compatibility with RADIUS [33] places part of the Length before the Type field and part after. However, very few protocols use such a complex encoding.

Given this diversity, we make several assumptions regarding the TLV structures that can be handled by our approach. These assumptions will be detailed in the following sections.

B. Assumptions

We assume that the studied protocol uses a well-formed TLV structure, where the T field is immediately followed by the L field, and then by the V field whose size matches the specified length. To cope with protocols such as IS-IS that use nested TLVs, we accept multiple TLVs within a single message and hierarchical TLVs. Hierarchical TLVs are TLVs inside a Value from a parent TLV. We further assume that TLVs are stored consecutively inside the packet, without any padding or alignment requirement. Each new TLV starts immediately after the previous one.

Concerning the Length field (L), we interpret it as a direct byte count, without any scaling factor or unit conversion. For example, a length of `0x10` is interpreted as exactly 16 bytes. We assume that the T and L portions have a fixed size in all packets of a given protocol.

We support two types of encodings for the Length field. In the value-only case, $L = |V|$. In the full TLV length case: $L = |T| + |L| + |V|$. Here, $|X|$ denotes the number of bytes required to encode field X. We consider that the TLVs with inconsistent or truncated lengths are invalid and skip them.

These assumptions rely on an explicit T–L–V structure, where each field is clearly separated in the packet. They do not apply to encodings such as CBOR where type and length information are not strictly separated and are partially encoded within the same byte.

IV. Inferring packet structure and TLVs

In this section, we describe how **PROtocol Structure Extractor** (**PROSE**) is able to automatically infer the structure of packets and the TLVs that they use from packet traces. **PROSE** provides the information required by our **Mutation & Stress Engine** (**MUSE**) to later fuzz packets. For this reason, **PROSE** prioritizes the correct identification of the T and L fields over any interpretation of the contents of the V field.

PROSE operates in three steps that are detailed in this section. It receives as input a network trace that only contains packets exchanged by the studied protocol. **PROSE** automatically returns the datalink, network layer and UDP headers if present in the trace to focus on the payloads. The first step, described in section IV-A, is to detect the location of the length field of the payload header. The second step, section IV-B, is to determine whether the message contains a checksum and if so provide its type and location. The third step, section IV-C, is to locate the beginning of the variable length part of the message, i.e. the beginning of the first TLV.

The computational overhead of **PROSE** remains limited in practice. The search space is tightly bounded, as the maximum size of the Type and Length fields is small (by default five bytes). In addition, routing protocol messages are typically short (a few tens to a few hundred bytes), and many invalid candidates are quickly discarded due to inconsistent length checks. In our experiments, the full inference process was completed in under 2 seconds on the collected traces.

A. Step 1: Locate the length field in the header

Some routing protocols such as Babel include a length field in their header. Identifying the header length field is challenging because it may overlap semantically with TLV length fields and can vary across protocols.

Since this field contains similar information as the Length part of TLVs, it could interfere with our TLV inference algorithm (section IV-C). We use BinaryInferno [34] to detect the presence and locate the position of a length field inside the packet header. If BinaryInferno has detected a length field, we provide it to **PROSE** to enable it to ignore this position and all earlier ones and reuse this parameter to find the checksum. This location is also provided as a parameter for our **Mutation & Stress Engine**.

B. Step 2: Inferring the presence of a checksum

Several approaches have been proposed for checksum inference, but they are less general. For example, [35] relies on more restrictive assumptions, such as the checksum being located at the end of the message, CRC-specific computation optimizations, and a computation window that always starts at the same position. There are also tools such as RevEng [36],

which is specialized exclusively in identifying CRCs using a predefined set of 113 algorithms, as well as Checksum Finder [37], which attempts to reconstruct a checksum implementation from a dataset. None of these approaches worked because our checksums are not CRCs, and despite multiple attempts, Checksum Finder never managed to reproduce the checksum implementations used in our protocols.

Inferring checksums is difficult because they introduce global dependencies across the packet and require identifying both the underlying algorithm and the exact computation region. Any mismatch typically leads to immediate rejection of the packet.

To infer the presence of a checksum, we must determine its position, the type of algorithm used to compute it and the parts of the packet that are used to compute it. Our detection process supports two standard checksums: the Fletcher-16 checksum [26] and the 16 bits Internet Checksum [25]. This design choice is motivated by the fact that these algorithms represent the most widely used checksum schemes in routing protocols. Other types of checksums such as Cyclical Redundancy Checks (CRC) [38] could be easily added if needed.

Our algorithm first estimates the largest possible computation region of the packet which serves as input for the checksum computation. This region is derived from the length inferred at the previous step (or the packet size if unavailable). It then determines the starting offset of this region and adjusts its size if necessary to match the actual checksum computation area. Finally, it identifies the offset of the checksum within this region.

Figure 3 illustrates the region used for checksum computation and the parameters we extract. The rectangles represent the bytes of the packet. The first vertical bar on the left marks the beginning of the computation region. The vertical bar on the far right indicates the supposed end of this region as inferred from the header length field. This boundary may be corrected, and the effective end of the region is shown by the vertical bar before it.

Within this region, the position of the checksum field, shown as a shaded segment, is determined relative to the start of the computation region.

The research parameters needed to recompute the checksum are first performed on one randomly selected packet. Our algorithm starts by testing all possible starting offsets for a computation region whose size is either the inferred length or the packet size. For each possible starting position, it tests every supported checksum algorithm as well as every possible offset for the checksum value within this region. If none of these combinations produce a value that matches the checksum present in the packet, the computation window is slid forward across the packet, and all algorithms and checksum offsets are tested again. Note that if the computation window initially covers the entire packet, this sliding step does not occur.

If no match is found after sliding the computation window over the entire packet, the computation window size is either decreased or increased by one byte, and the entire procedure is restarted from the beginning. Through this iterative process,

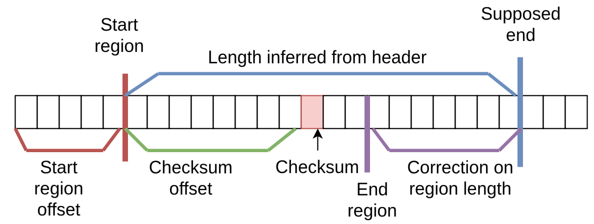

Fig. 3. Parameters inferred and used for checksum

the algorithm produces a set of candidate parameters. Each set of parameters includes a checksum algorithm, a computation region, an offset of the checksum within that window, and a correction applied to the window length to compute the good checksum.

These candidate parameters are then validated against other packets from the same trace. We only retain the parameters that consistently produce a correct checksum for every packet. Even after this validation step, several candidates may remain. In that case, we compute the entropy of the checksum field for each candidate position. The candidate with the highest entropy is selected as the most probable checksum configuration.

C. Step 3: Detecting where TLV structures begin

To describe our approach in detail, we illustrate each step using the three functions expressed in pseudo-code. As shown in Algorithm 1, the main procedure simply calls the `TL_FINDER` function with a single packet trace as input.

Before the analysis can begin, the algorithm requires an upper bound on the possible size of the Type and Length fields of a TLV structure. By default, this maximum size is set to 5 bytes in the variable `K`. This is the only parameter in the program, but generally, the Type and Length fields are not large. The algorithm then extracts all messages from the packet trace into a list, denoted `M` in algorithm 1. As explained earlier, we invoke the `BinaryInferno` tool to infer the position of the header's length field; this value is stored in variable `H`.

The objective of the `TL_FINDER` method is to identify the TLV parsing parameters that allow all messages in the trace to be interpreted as sequences of TLVs. For each candidate configuration (i.e., Type size, Length size, and starting offset), the algorithm attempts to parse every message as a chain of TLVs that exactly covers the packet. Among all configurations, the one that maximizes the number of distinct TL headers observed is selected. The resulting set of TL (the concatenation of Type and Length fields without the value) extracted under this configuration is stored in `best_S`, while its associated parameters such as the checksum configuration, the correct sizes of the Type and Length fields, and the inferred header size and the interpretation of the Length field (value-only or full TLV size) are stored in `best_P`. Finally, these elements are returned as a JSON object, together so that they can be directly used by **MUSE** for packet generation and fuzzing.

The first challenge is that we do not know in advance the size of the Type field or the size of the Length field. For this reason, our algorithm begins with two nested loops that test all possible combinations of field sizes, ranging from *1* to *K - 1* bytes for the Type field and from *1* to *K - t* bytes for the Length field where the variable `t` is the Type field length. After evaluating all combinations, the configuration that yields the best results is selected, we will describe later how we select the best configuration. This allows the algorithm to infer the actual size of both the Type field and the Length field. In Algorithm 1, the Length field size is stored in variable `l`, and the Type field size is stored in variable `t`

Inside these two loops, the algorithm must first determine the correct offset at which the TLV section begins in each packet. To achieve this, all possible starting positions are evaluated, and for each position, we check whether a valid TLV chain can be found across all messages. This is performed by calling the `offsetRates` function, detailed in Algorithm 2. This function requires the list of messages extracted from the trace, as well as the candidate sizes of the Type and Length fields.

To determine the correct offset, the `offsetRates` function first identifies the shortest message in the trace. This choice is deliberate: if TLVs are present in the packets, any valid TLV chain must also fit within the smallest message. Therefore, there is no benefit in testing starting positions that exceed its length, since a TLV cannot begin there.

The algorithm then evaluates every byte position from the beginning of the message up to the length of the shortest packet. For each candidate offset and for each message, the algorithm invokes the `loopOK` function detailed in Algorithm 3. This function checks whether a valid TLV structure could begin at the given position in the message. In other words, it determines whether the offset is plausible as the start of a TLV chain.

After testing all offsets across all messages, the function returns an array in which each entry contains the percentage of messages for which `loopOK` considered that offset to be a valid TLV start position.

To determine whether a given position corresponds to the start of a TLV, the `loopOK` function requires three parameters: the message to be tested, the candidate position (`pos` variable in Algorithm 3), and the potential sizes of the Type and Length fields.

The function `loopOk` begins by reading the value of the Length field. Since TLVs store the Type first, the Length field is located immediately after the Type. Thus, the function interprets the length at the position given plus the size of the Type field. If the extracted length is inconsistent. For example, if it exceeds the remaining size of the packet or is negative, then the candidate position cannot correspond to the start of a TLV.

Otherwise, the function continues this process iteratively (or recursively) along the message. A starting position is considered valid only if this iterative search reaches exactly the end of the packet. Any deviation, such as stepping beyond the packet boundaries, immediately invalidates the candidate offset. Importantly, this condition must be satisfied for every packet in the trace.

Thanks to the `offsetRates` function, `TL_FINDER` obtains an array of offsets stored in variable `R`, along with the percentage of messages for which a TLV was successfully detected at each position. We then retain only the offsets for which a TLV was found in 100% of the messages.

A subtle issue arises because some offsets may appear valid even though they correspond to the header length field rather than the beginning of the TLVs. When the header length directly encodes the total packet size, the function loopOK might incorrectly conclude that the TLV chain is valid simply because following this offset reaches the end of the packet. To prevent such false positives, the algorithm leverages the information provided by `BinaryInferno`, which identifies the exact location of the header length field. Any offset at or before this position is ignored, and the earliest offset strictly after the inferred header length field is selected as the starting position of the TLV section.

If no offset achieves a perfect 100% success rate, the algorithm concludes that at least one packet in the trace does not contain TLVs. In this situation, the TLV section is assumed to begin immediately after the header, and the header size is set to the length of the smallest packet in the trace.

D. Selection of good candidates

Once the starting index of the first TLV is determined, most of the work is complete. The algorithm then iterates through every packet and performs successive jumps according to the inferred TLV structure, collecting every Type and Length field encountered along the way. This yields the set of all top-level TLV types observed in the trace. This is done in the pseudo-code by the method `collectTL` called in Algorithm 1.

A subtle point remains, however. These jumps may cross over TLVs that are themselves nested inside the value field of another TLV. To handle such cases, we apply the algorithm described above, but only to the Value portion. Because of the diversity of value fields in the trace, we cannot simply search for a sub-TLV. For a given Value size, some messages may contain one sub-TLV, others two, and others none at all.

To reduce this variability for our algorithm, we create a set of messages associated with each T and L previously inferred. This grouping allows us to gather messages that share the same Type and Length, and therefore are likely to have a similar internal structure. We then apply our inference procedure on each of these sets to determine where sub-TLVs might begin.

However, in this case, this approach is considerably less robust than the previous one. For example, a trace from a network where IP addresses share similar byte patterns might contain a value that is misinterpreted as a length, causing the algorithm to incorrectly assume that the sub-TLV reaches the end of the message.

E. Automatic exploration of different hypotheses to find the Type and Length sizes

Finally, as explained before, for each possible TL size, the algorithm implicitly tests every feasible internal split between the Type and Length fields. It also evaluates both interpretations of the Length field: whether it encodes only the value size or the full TLV size. This step is not shown in the pseudocode, as it simply corresponds to restarting the algorithm with a different interpretation of the length.

For every possible size assignment of the Type and Length fields, where each field has a minimum size of 1 and their combined size does not exceed the predefined bound, the entire inference workflow is executed. The quality of each configuration is measured by the number of different Type and Length candidates it produces across all packets. This is illustrated at the end of Algorithm 1 by comparing the size of S with `best_s`. The configuration that produces the largest set of TL candidates is then selected.

Algorithm 1 TL_FINDER

```
1:  function TL_FINDER(packetsTrace)
2:      K ← 5
3:      M ← extractPayloads(packetsTrace)
4:      H ← BinaryInferno(packetsTrace).max
5:      best_S ← ∅
6:      best_P ← None
7:      for t = 1 to K − 1 do
8:          for ℓ = 1 to K − t do
9:              R ← OFFSETRATES(M, t, ℓ)
10:             V ← { o | R[o] = 100% }
11:             if V ≠ ∅ then
12:                 h ← smallest element of V not smaller than H
13:             else
14:                 h ← minMsgLen(M)
15:             end if
16:             S ← COLLECTTL(M, h, t, ℓ)
17:             if |S| > | best_S | then
18:                 best_S ← S
19:                 best_P ← (t, ℓ, h)
20:             end if
21:         end for
22:     end for
23:     return (best_S, best_P)
24: end function
```

Algorithm 2 offsetRates

```
1:  function OFFSETRATES(M, t, ℓ)
2:      L_max ← minMsgLen(M)
3:      rates ← array of size L_max
4:      for o = 0 to L_max − 1 do
5:          s ← 0
6:          for all msg ∈ M do
7:              if LOOPOK(msg, o, t, ℓ) then
8:                  s ← s + 1
9:              end if
10:         end for
11:         rates[o] ← 100% · s/|M|
12:     end for
13:     return rates
14: end function
```

Algorithm 3 loopOK

```
1:  function LOOPOK(msg, pos, t, ℓ)
2:      n ← |msg|
3:      while pos < n do
4:          if pos + t + ℓ > n then
5:              return false
6:          end if
7:          L ← getLength(msg, pos + t, ℓ)
8:          if L < 0 then
9:              return false
10:         end if
11:         next ← pos + t + ℓ + L
12:         if next > n then
13:             return false
14:         end if
15:         pos ← next
16:     end while
17:     return (pos = n)
18: end function
```

V. FUZZING WITH THE INFERRED PACKET STRUCTURE

Our **Mutation & Stress Engine** receives the following information from **PROSE**: (i) the location of the header length field, (ii) the location and type of the header checksum and (iii) the set of observed *Type, Length* pairs.

Based on this information, **MUSE** constructs fuzzed packets by randomly selecting a packet from the network trace. For efficiency, **MUSE** generates several random indices that could be tested within the packet. Then the indices are selected one after another. It is unlikely that an index will correspond exactly to the position of a TLV, so if no TLV is recognized at that position, we increment the index until a TLV is detected. When a TLV is recognized, there is a certain probability that this TLV will be fuzzed. Then, **MUSE** updates the header Length and checksums if they are used by this protocol.

Malformed TL headers, such as an invalid Type value or a L value that contradicts the expected size for that Type, are likely to trigger immediate message rejection by the target, preventing processing of the V field. Such early rejections reduce the effectiveness of fuzzing, as the payload (V) is never processed. Therefore, our approach focuses on accurately identifying and preserving TL, while mutating the V field to explore deeper code paths.

MUSE prioritizes fuzzing the values, and with a lower probability, fuzzing the types or length parts. To fuzz, **MUSE** randomly selects mutators. **MUSE** includes four types of mutators. Our *random mutator* randomly mutates one or more bytes in a given byte sequence; in our case, this could be the TLV value or the type. The *flip-bit mutator* selects a random byte from the given sequence and flips one bit. The *truncate mutator* shortens a given byte sequence by a random length. Finally, the *extend mutator* extends a given byte sequence with random bytes.

For the *extend* and *truncate mutators*, **MUSE** recomputes the length of the TLV, as well as any corresponding header length fields when necessary. This recomputation is performed 90% of the time, allowing the remaining 10% of the mutations

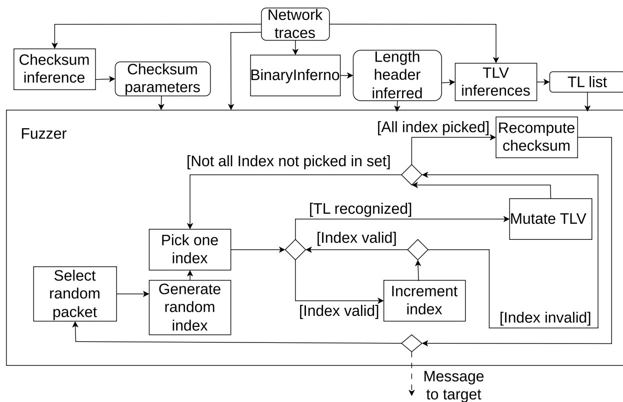

Fig. 4. Fuzzer workflow and interaction with other components

to produce incorrect length fields to also test packets with a malformed length.

Finally, after a packet has been modified, if a checksum was detected by **PROSE**, then **MUSE** recomputes the checksum before sending the packet to the target.

Figure 4 shows in detail how the components of our fuzzing process work, and how **MUSE** fits with the other components of the tool. We can see that the packet-length inference is the starting point. Then, TLV inference and checksum inference are performed independently. Once all these elements are available, the fuzzing stage can begin.

VI. Evaluation of **PROSE** on routing protocols

To evaluate the performance of **PROSE**, we consider three very different routing protocols: IS-IS, EIGRP and Babel. We collected a trace with 120 packets from each protocol using our Containerlab setup described earlier. We configured the virtual routers with all supported optional protocol features to maximize the number of TLV fields used. Knowing the structure of these protocols, we developed a parser that reads the trace in JSON format and using a set of keywords specific to the TLV of the protocols we are testing, the parser finds the TLV in the trace. We use the output of this parser as the ground truth and compare it to the results obtained by **PROSE**.

We use two metrics for our evaluation: (i) precision and (ii) recall. The precision measures the proportion of inferred TL sequences that are actually valid according to the ground-truth parser. A high precision indicates few false positives. The recall measures the proportion of valid TL sequences present in the traces that were correctly identified by **PROSE**. A high recall indicates strong coverage of the protocol's TL structures.

For IS-IS and Babel, the length field is correctly inferred by BinaryInferno, and for IS-IS and EIGRP we correctly infer their checksum. Moreover, BinaryInferno does not produce false positives. It does not infer a length field for EIGRP, which has none in its header, and similarly, our algorithm returns no checksum for Babel, as the protocol does not include one.

For TLV inference, our algorithm identifies all TLVs as long as sub-TLVs are not considered. It successfully detects the 16

	IS-IS	Babel	EIGRP
Precision	68.57	100	100
Recall	75	88.89	100

TABLE I
PROSE TLV inference results (%) per protocol

TLVs present in our IS-IS dataset, the 7 TLVs in the Babel dataset, and the 4 TLVs in the EIGRP dataset.

However, the results degrade when we extend the analysis to sub-TLVs. As we see in Table I, for Babel, the precision is not 100% because one sub-TLV could not be detected: this is a TLV with a length of 0. Accepting sub-TLVs with a length of `0` would cause our algorithm to interpret arbitrary offsets as sub-TLVs, and break our initial assumption that a TLV must follow the structure T followed by L followed by V. **PROSE** detected one sub-TLV, identified 6 TLVs and missed one sub-TLV. For IS-IS, the issue arises from the greater size of the packets: at some point, a byte value inside the frame may be interpreted as a length that runs past the end of the packet. Despite this, our approach still manages to correctly infer 8 sub-TLV on 16 sub-TLV. Finally, for EIGRP, there are no sub-TLVs, which is why it changes nothing.

PROSE achieved good overall performance on EIGRP, IS-IS and Babel, with these protocols showing high recall and acceptable precision.

VII. Fuzzing routing protocols

We use Containerlab [39] to simulate an interconnected router topology and FRRouting version 10.3.1 [31] to provide protocol implementations. All evaluated protocols rely on this FRRouting implementation.

A small five-router topology as described in Figure 2 was deployed on a laptop (Dell Latitude 5530, Intel i7-1255U, 16 GB RAM).

PROSE runs on the host and processes network traces to compute TL candidates. **MUSE** is deployed on `Router 2` (in Figure 2) and targets `Router 1`. Once TL are identified on the host, they are passed to **MUSE** running on `Router 2`, which generates and sends test packets to `Router 1`.

A. Coverage

To evaluate the performance of **MUSE**, we slightly modified FRRouting to connect it to GNU's source coverage analysis tool (GCOV). More precisely, we added a custom signal handler that forces a call to `__gcov_flush()` whenever FRRouting receives a specific signal. We then compared **MUSE** to a fuzzer that performs random fuzzing, which we call Random Fuzzer.

This fuzzer operates in the same way as our TLV-aware fuzzer, except that it does not know TLVs, header sizes, or checksum parameters. Like our TLV fuzzer, it selects a random frame from the dataset and applies random mutations to it, using the same mutation probabilities.

We report our coverage results when running **MUSE** with IS-IS, EIGRP and Babel by focusing on representative files

for each protocol that are directly involved in TLV processing files.

For IS-IS, we selected the file `isis_tlvs.c`, as it is responsible for processing TLVs within the protocol implementation. For EIGRP, the situation is more complex because TLVs are handled in multiple locations throughout the codebase. We therefore selected the `eigrp_hello.c` file, which processes Hello messages and is where our tool detected a bug related to TLV handling. For Babel, we selected the `message.c` file, which is the part of the program where TLVs are processed.

Coverage was measured after 100, 500, 1000, and 2000 packets with each experiment repeated twenty times, and the median value is reported in the Table II

For IS-IS, **MUSE** frequently triggered crashes (9/20 runs at 500 packets, 15/20 at 1000, and all runs at 2000). That's why we don't have data for IS-IS at 2000 packets in Table II.

Since our fuzzer stops the program after only two or three packets due to an incorrect assertion, it is difficult to obtain meaningful coverage measurements. Although the IS-IS process may crash after two or three malformed packets due to failed assertions, the crash is triggered by a delayed verification mechanism rather than immediately upon packet reception. As a result, multiple packets are processed before the verification occurs, allowing coverage information to accumulate despite the eventual termination.

For the same number of packets, we observe that **MUSE** achieves higher coverage in IS-IS. The difference in coverage between **MUSE** and the Random Fuzzer is explained by checksum recomputation and several check assertions, which enable access to a larger portion of the code.

For EIGRP, both the Random Fuzzer and the TLV-based fuzzer caused crashes after only 15–20 inputs. After fixing the discovered issues and validating the modifications with the FRRouting test suite. We measured the coverage present as reported in Table II. **MUSE** exercises significantly more TLV related code, including functions such as `eigrp_hello_receive`. This can be explained by the fact that with the random approach, upstream methods that are responsible for invoking TLV-processing functions are triggered less frequently. This is due to early checks, such as header validations, or to frequent structural errors in TLVs, which also occur within the TLV-processing methods themselves.

For Babel, **MUSE** also achieves higher coverage. For example, `parse_hello_subtlv` is invoked 175 times for 500 packets, compared to 108 with the random approach. By generating structurally valid TLVs, **MUSE** exercises sub-TLV handling code more frequently and reaches deeper parsing paths that are often blocked by early validation failures in purely random mutations

B. *Vulnerabilities discovered by* ***MUSE***

When **MUSE** fuzzed IS-IS, we detected a crash of the IS-IS process. Although it initially restarted, it crashed again after 3–4 seconds and was then unable to restart. Log and packet analysis suggest that a 213-byte packet was the root cause, though other inputs may also cause it. We are currently interacting with the developers to report the issue.

TABLE II
COVERAGE COMPARISON (%) BETWEEN **MUSE** AND RANDOM FUZZER

Packets	IS-IS		EIGRP		Babel	
	MUSE	Random Fuzzer	**MUSE**	Random Fuzzer	**MUSE**	Random Fuzzer
100	32.94	30.50	59.91	55.06	65.16	61.45
500	33.01	30.50	59.91	56.27	74.93	65.34
1000	35.04	32.73	59.91	56.68	76.70	71.97
2000	/	32.73	59.91	57.89	81.34	74.26

While fuzzing the EIGRP implementation in FRRouting, **MUSE** generated a malformed Hello packet containing a TLV with an indicated length of, for example, 16 bytes, followed by extra trailing data. This packet caused a parsing error. The EIGRP parser misinterprets these trailing bytes as additional TLVs. If the trailing bytes have a value of `0x00`, they are interpreted as a new TLV with a length of zero, resulting in an infinite loop since the parsing offset is never incremented.

We observed that a single malformed packet caused the EIGRP process to consume nearly 100% CPU and rendered the vtysh interface unresponsive, which implies that the network operator cannot anymore reach the router to debug the problem. Although FRRouting restarts the daemon via its watchdog mechanism, the issue immediately reoccurs.

To understand the root cause of the problem, we executed FRRouting within Valgrind. It detected uninitialized memory reads. We observed that after reading one TLV, the parser immediately reads the header of the next TLV directly from the packet buffer. If the previous TLV length field is incorrect, this can cause an inconsistent size, leading the parser to read from unallocated memory. We reported this problem to the maintainers of FRRouting[2]. They fixed it in FRRouting versions 10.3 and 10.4.

During fuzzing experiments, **MUSE** generated certain malformed EIGRP packets that caused a restart. Under normal circumstances, malformed EIGRP packets should be detected and safely discarded without affecting the stability of the routing daemon. However, in this case, the malformed input triggered an assertion failure in the `stream_getc()` function, leading to the unexpected termination and restart of the EIGRP process. We reported this problem to the maintainers[3], but it has not yet been fixed.

Additionally, **MUSE** generated malformed packets that triggered another assertion failure in the `masklen2ip()`, `stream_getc()` and `eigrp_fsm_event_nq_fcn()` functions, similarly causing the daemon to abort and restart.

VIII. RELATED WORK

BinaryInferno [34] allowed us to automatically infer the protocol's header size from the rest of the message. **PROSE** goes beyond BinaryInferno by successfully detecting most TLVs across the analyzed protocols.

Auto-ETLV [40] can only infer the TLV length field at bit and does not rely on predefined unit sizes. While robust to

[2]See https://github.com/FRRouting/frr/issues/19250
[3]See https://github.com/FRRouting/frr/issues/19503

varying granularity, it detects at most one TLV per packet, limiting its applicability to complex or nested TLV structures.

An earlier method infer T, L and V, but relies on restrictive assumptions including single byte length field, strict T–L–V ordering, byte-level analysis, no tolerance for malformed values, and prior knowledge of the number of types [41].

Nemesys [42] does not infer the semantic type of a field (e.g., whether it represents a length, a type, etc.). Instead, it focuses on identifying boundaries between fields. However, this boundary-based approach assumes fixed field positions, which is not compatible with TLV protocols where field sizes vary. We also test Nemetyl [43] which is an evolution of Nemesys but it looped forever on our Babel packet trace.

Netplier [44] does not recognize Length or Type field and assumes fixed fields but with TLVs, our routing packets can have multiple successive length fields. Moreover they need to know a field position to perform their clustering in contrast with our approach that uses a minimum size threshold. We also tested using Netzob [45] but we obtained an infinite loop with our IS-IS packet trace.

FRRouting includes built-in fuzzing targets, but they require source code modifications and are limited to specific daemons and internal entry points. Additionally, the fuzzing code is maintained in a separate branch that may lag behind the main codebase.

Fully automated fuzzers such as Pulsar [15] or Autofuzz [18] target mostly textual protocols and assume symmetric client–server exchanges, which do not reflect routing behavior.

Grey-box fuzzers like AFLnet [46] leverage coverage feedback but support only a small set of protocols and lack features needed for routing, such as automatic checksum recomputation. Black-box tools like Boofuzz [47] require a manual description of the message formats.

FuzzyCat [48] targets routing protocols, but focuses on configuration errors rather than implementation vulnerabilities. Its MitM approach [20] between two routers, avoiding state-machine modeling at the cost of generality. The authors still had to manually design a data model and identify key fields to fuzz, an effort that must be repeated for each protocol, which restricts the scalability of the approach.

We attempted to use existing fuzzers such as boofuzz and AFLNet as baselines. However, due to their inability to handle routing-specific constraints, they failed to generate valid test cases, preventing meaningful comparison.

IX. Conclusion

Routing protocols play a key role in the public Internet and in enterprise networks. Routers exchange messages to compute the routing tables that they use to forward packets. The failure of a router could bring down parts of a network. For the robustness of these networks, routers should never crash upon reception of a malformed or malicious message.

In this paper, we have proposed a pair of tools: **PROtocol Structure Extractor** (**PROSE**) and **Mutation & Stress Engine** (**MUSE**). **PROSE** can automatically extract the structure (header, checksum and TLV) of different routing protocols. We have applied it successfully to Babel, IS-IS and EIGRP. **MUSE** uses a packet trace and the output of **PROSE** to fuzz packets. We have used it to fuzz virtual routers running FRRouting and identified four bugs that caused this popular implementation to crash. Three of these bugs were in the EIGRP daemon and one in the IS-IS daemon.

Since **PROSE** and **MUSE** are generic, they can be used with proprietary protocols as well as the standardized ones that we used in this paper. For our further work, we will evaluate them on different protocols and will also explore how they can be extended to support protocols running over a TCP bytestream such as BGP.

MUSE and **PROSE** are written in Python. In our experimental setup, fuzzing throughput is primarily limited by the processing time of the target, rather than input generation speed. They will be made publicly available. We will also provide our Containerlab network configurations. The packet trace used in the experiment will also be available. In addition, we will provide the Python script used to parse TLVs and compare them with the output of **PROSE**. Finally, we will provide the bash scripts used to automatically run the experiments and collect the coverage.

Acknowledgment

This work was supported by the Walloon Region of Belgium under the conventions n°2110186 (Cyber Excellence).

References

[1] P. Neumann, “The crash of the at&t network in 1990,” https://telephoneworld.org/landline-telephone-history/the-crash-of-the-att-network-in-1990, year=1990.

[2] “A Brief History of the Internet’s Biggest BGP Incidents — kentik.com,” https://www.kentik.com/blog/a-brief-history-of-the-internets-biggest-bgp-incidents/, [Accessed 02-05-2026].

[3] E. Rominj, “RIPE NCC and Duke University BGP Experiment,” 2010, https://labs.ripe.net/author/erik/ripe-ncc-and-duke-university-bgp-experiment/.

[4] L. Wang, M. Saranu, J. M. Gottlieb, and D. Pei, “Understanding bgp session failures in a large isp,” in *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE, 2007, pp. 348–356.

[5] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[6] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[7] X. Zhang, C. Zhang, X. Li, Z. Du, B. Mao, Y. Li, Y. Zheng, Y. Li, L. Pan, Y. Liu *et al.*, “A survey of protocol fuzzing,” *ACM Computing Surveys*, vol. 57, no. 2, pp. 1–36, 2024.

[8] 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.

[9] P. Godefroid, M. Y. Levin, and D. Molnar, “Sage: whitebox fuzzing for security testing,” *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[10] U. Kargén and N. Shahmehri, “Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 782–792.

[11] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-fuzz: fuzzing by program transformation,” in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[12] M. Zalewski, “American fuzzy lop,” http://lcamtuf.coredump.cx/afl/.

[13] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

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

[15] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11*. Springer, 2015, pp. 330–347.

[16] S. H. Ramos, "Polymorph: A real-time network packet manipulation framework," *Exploit Database*, 2018.

[17] T. Ramsauer, "Black-box live protocol fuzzing," *Target*, vol. 2, pp. 1–2, 2021.

[18] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *Ijcsns*, vol. 10, no. 8, p. 239, 2010.

[19] Forescout, "GitHub - Forescout/bgp_boofuzzer — github.com," https://github.com/Forescout/bgp_boofuzzer, [Accessed 03-03-2025].

[20] C. Wen, Y. Liu, and S. Li, "A routing protocols fuzzing method based on man-in-the-middle," in *2022 2nd International Conference on Frontiers of Electronics, Information and Computation Technologies (ICFEICT)*. IEEE, 2022, pp. 491–496.

[21] W. Almuhtadi, W. Fenwick, L. Henley-Vachon, and P. Mitchell, "Assessing network infrastructure-as-code security using open source software analysis techniques applied to bgp/bird," in *2022 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2022, pp. 1–6.

[22] D. Savage, J. Ng, S. Moore, D. Slice, P. Paluch, and R. White, "Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP)," RFC 7868 (Informational), RFC Editor, Fremont, CA, USA, May 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7868.txt

[23] R. Callon, "Use of OSI IS-IS for routing in TCP/IP and dual environments," RFC 1195 (Proposed Standard), RFC Editor, Fremont, CA, USA, Dec. 1990, updated by RFCs 1349, 5302, 5304. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1195.txt

[24] J. Chroboczek and D. Schinazi, "The Babel Routing Protocol," RFC 8966 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jan. 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8966.txt

[25] R. Braden, D. Borman, and C. Partridge, "Computing the Internet checksum," RFC 1071 (Informational), RFC Editor, Fremont, CA, USA, Sep. 1988, updated by RFC 1141. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1071.txt

[26] J. Fletcher, "An arithmetic checksum for serial transmissions," *IEEE transactions on Communications*, vol. 30, no. 1, pp. 247–252, 1982.

[27] N. Shen (Ed.) and A. Zinin (Ed.), "Point-to-Point Operation over LAN in Link State Routing Protocols," RFC 5309 (Informational), RFC Editor, Fremont, CA, USA, Oct. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5309.txt

[28] M. Bagnulo, C. Paasch, F. Gont, O. Bonaventure, and C. Raiciu, "Analysis of Residual Threats and Possible Fixes for Multipath TCP (MPTCP)," RFC 7430 (Informational), RFC Editor, Fremont, CA, USA, Jul. 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7430.txt

[29] C. Hopps, "Routing IPv6 with IS-IS," RFC 5308 (Proposed Standard), RFC Editor, Fremont, CA, USA, Oct. 2008, updated by RFC 7775. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5308.txt

[30] G. Malkin, "RIP Version 2," RFC 2453 (Internet Standard), RFC Editor, Fremont, CA, USA, Nov. 1998, updated by RFC 4822. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2453.txt

[31] F. Project, "FRRouting — frrouting.org," https://frrouting.org/, [Accessed 29-08-2025].

[32] V. Fajardo (Ed.), J. Arkko, J. Loughney, and G. Zorn (Ed.), "Diameter Base Protocol," RFC 6733 (Proposed Standard), RFC Editor, Fremont, CA, USA, Oct. 2012, updated by RFCs 7075, 8553. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6733.txt

[33] C. Rigney, S. Willens, A. Rubens, and W. Simpson, "Remote Authentication Dial In User Service (RADIUS)," RFC 2865 (Draft Standard), RFC Editor, Fremont, CA, USA, Jun. 2000, updated by RFCs 2868, 3575, 5080, 6929, 8044, 9765. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2865.txt

[34] J. Chandler, A. Wick, and K. Fisher, "Binaryinferno: A semantic-driven approach to field inference for binary message formats." in *NDSS*, 2023.

[35] J. Pohl and A. Noack, "Automatic wireless protocol reverse engineering," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[36] "CRC RevEng: arbitrary-precision CRC calculator and algorithm finder — reveng.sourceforge.io," https://reveng.sourceforge.io/, [Accessed 04-02-2026].

[37] L. Labell, J. Chandler, and K. Fisher, "Automatic discovery and synthesis of checksum algorithms from binary data samples," in *Proceedings of the 15th Workshop on Programming Languages and Analysis for Security*, 2020, pp. 25–34.

[38] S. S. Gaitonde and T. V. Ramabadran, "A tutorial on crc computations," *IEEE Micro*, vol. 8, no. 4, pp. 62–75, 1988.

[39] R. Dodin, "containerlab — containerlab.dev," https://containerlab.dev/, [Accessed 29-08-2025].

[40] Z. Huang, K. Wu, S. Huang, Y. Zhou, and R. S. Giagone, "Automatic field extraction of extended tlv for binary protocol reverse engineering," in *2022 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2022, pp. 1–10.

[41] L. He, Q.-y. Wen, and Z. Zhang, "A tlv structure semantic constraints based method for reverse engineering protocol packet formats," *Journal of Networking Technology*, vol. 5, no. 1, p. 9, 2014.

[42] S. Kleber, H. Kopp, and F. Kargl, "{NEMESYS}: Network message syntax reverse engineering by analysis of the intrinsic structure of individual messages," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[43] S. Kleber and F. Kargl, "Refining network message segmentation with principal component analysis," in *2022 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2022, pp. 281–289.

[44] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "Netplier: Probabilistic network protocol reverse engineering from message traces." in *NDSS*, 2021.

[45] G. Bossert, F. Guihéry, G. Hiet *et al.*, "Netzob: un outil pour la rétro-conception de protocoles de communication," in *SSTIC 2012*, 2012, p. 43.

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

[47] J. Pereyda, "BooFuzz," http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, 2012, [Online; accessed 28-february-2025].

[48] J. Cai, G. Yang, J. Liu, and Y. Xie, "Fuzzycat: A framework for network configuration verification based on fuzzing," in *2023 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 2023, pp. 123–131.