

Automatic Field Extraction of Extended TLV for Binary Protocol Reverse
Engineering

by

Zewen Huang

B.Math., University of Waterloo, CA, 2019

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Zewen Huang, 2022
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by
photocopying or other means, without the permission of the author.

Supervisory Committee

Dr. Kui Wu, Supervisor
(Department of Computer Science)

Dr. Jianping Pan, Departmental Member
(Department of Computer Science)

ABSTRACT

Type Length Value (TLV) is one of the main structures commonly used in network protocols. A large number of proprietary protocols, whose specification is unknown to the public, run in the current Internet as well as domain-specific Internet of Things (IoT) applications. It is critical to infer the TLV fields within a packet because this information can help network administrators quickly identify abnormal traffic and potential attacks. Inferring TLV fields belongs to the general task of protocol reverse engineering and is particularly challenging for binary protocols, where the boundaries of TLV fields have many possible positions. Existing methods for reverse engineering binary protocols involve many parameters and only work for protocols strictly following the conventional TLV format. We extend the concept of TLV to accommodate a broader category of structural patterns in various binary protocols, such as TCP, IP, ModBus, and MQTT. We then design algorithms to *automatically* extract the extended-TLV fields from packets. Via a series of experiments over several protocols, we demonstrate that our algorithms can accurately and quickly identify the extended-TLV fields in all the tested protocols. Our approach can thus be deployed as a general method for automatically reverse engineering binary protocol format.

Contents

Supervisory Committee	ii
Abstract	iii
Contents	iv
List of Tables	vi
List of Figures	vii
Acknowledgements	viii
1 Introduction	1
1.1 Background on Protocol Reverse Engineering	1
1.2 Related Work	3
1.2.1 Network Trace-Based Approach	3
1.2.2 Execution Trace-Based Approach	3
1.3 The Problem to Be Solved	4
1.4 Challenges and Contributions	4
1.5 Thesis Organization	6
2 Extended TLV (ETLV) and Solution	7
2.1 Background and Problem Definition	7
2.1.1 What is TLV?	7
2.1.2 Extended TLV (ETLV)	8
2.2 Problem Analysis	11
2.2.1 The Property of $\mathcal{V}(\hat{L}), \mathcal{L}(\hat{L})$	11
2.2.2 The Unit Problem	12
2.2.3 The Property of $\mathcal{V}(L), \mathcal{L}(\hat{L})$ in Different Units	13
2.2.4 Constant Difference	13

2.3	Algorithm Design in Auto-ETLV	14
2.3.1	A High-Level Overview	14
2.3.2	Global Dictionary	16
2.3.3	Main Structure of <i>Auto-ETLV</i>	17
2.3.4	Generation Step	18
2.3.5	GD update	19
2.3.6	Verification Step	20
2.3.7	Time Complexity	21
2.4	Why Does Auto-ETLV Work?	22
3	Experiments	23
3.1	Evaluation Results	23
3.1.1	Accuracy	23
3.1.2	Dynamics of GD	26
3.1.3	The Impact of Parameters	27
4	Interactive ETLV Application	31
4.1	Control Panel	31
4.2	Display Panel	33
4.3	An Example Case Study	34
5	Discussion and Conclusion	37
5.1	Discussion	37
5.2	Nested ETLV	38
5.3	Conclusion	38
	Bibliography	39

List of Tables

Table 3.1 Location of the L Field	24
---	----

List of Figures

Figure 2.1 TLV structure	7
Figure 2.2 ETLV structure	9
Figure 2.3 Algorithm Flow	15
Figure 3.1 The format of TCP.	25
Figure 3.2 Size of global dictionary as we process more packets	26
Figure 3.3 Maximum Dcounter after processing each packet	27
Figure 3.4 Running time vs. the probability for executing the G step(sample rate), the filtering rate = 0.9.	28
Figure 3.5 Running time vs. the filtering rate, the probability for executing G step = $\frac{1}{1+index}$	29
Figure 4.1 Interactive tool for ETLV analysis.	32
Figure 4.2 Interactive tool for ETLV analysis. Layer parameter is set to tcp-app. We click the open button and choose mqtt.pcap.	35
Figure 4.3 Output for ETLV analysis for MQTT.	36

ACKNOWLEDGEMENTS

I would like to thank:

Supervisor, Dr. Kui Wu, for mentoring, support, encouragement, and patience.

Yang Zhou, Benny Huang, and Ronnie Salvador Giagone for their helpful discussion and feedback.

Dr. Jianping Pan and Dr. Issa Traore, for serving in my oral defence committee and giving helpful feedback.

Huawei Technologies Canada Co. Ltd. for the funding support of this research.

Thank you.
Zewen Huang

Chapter 1

Introduction

1.1 Background on Protocol Reverse Engineering

In recent decades, the Internet has evolved explosively, and there have been tremendous traffic flows running over the Internet each day. When people communicate in the real world, they need to speak the same language to understand each other. Language is nothing more than a set of sequences made of symbols organized in some particular structure. Similarly, in the context of computer networks, two hosts communicate with each other and agree on the same language, i.e., network protocol, which defines rules on how to interpret information carried in packets transmitted during the communication. As the network grows, many different protocols emerge. Some protocols, e.g., TCP/IP, are published as open standards, but many remain private or proprietary since a company develops them for a special purpose or their own products. In most cases, attackers or malware developers may use their secret protocols to perform attacks.

Nowadays, malware using private protocols has become a serious threat to Internet security. Some malware may compromise a large number of computers as bots and use a command-and-control (C&C) server to control the bots to perform distributed deny-of-service (DDoS) attacks. The communication between the C&C server and the bots may use a private protocol. If we can infer the private protocol's packet structure, intrusion detection with deep packet inspection would be much easier.

Protocol reverse engineering (PRE) is a technique for this purpose. Simply speaking, PRE refers to a collection of methods that extract protocol features without knowledge about protocol specification. A network protocol can be classified as text-

based, such as HTTP, where the packet header includes plaintext and is easy to analyze, and binary-based, such as TCP, where the packet header contains only binary bits. One essential task in PRE is identifying the private protocol’s field boundaries. While this task is relatively simple for text-based protocols, it is very challenging for binary protocols. Unfortunately, over 40% of traffic on the backbone of the Internet uses private protocols [16], among which a significant portion are binary protocols. This phenomenon has become more prominent with the recent advance of the Internet of Things (IoT), where many IoT systems are proprietary and use binary protocols unknown to the public. Overall, the lack of knowledge of underlying network protocols poses a significant barrier to accurately detecting abnormal traffic and intrusions.

Among previous works in automatic reverse engineering private protocols, the most famous one is the Project Informatics (PI) project [3]. PI borrowed the idea from bioinformatics and used a multiple sequence alignment (MSA) algorithm to discover the common pattern across the packets in network traffic. Since then, the concept of sequence alignment has played a fundamental role in PRE and has motivated a series of subsequent works. For instance, Cui et al. [7] designed Discoverer, a tool that tokenizes the sequence of packets and uses type-based sequence alignment and clustering algorithms to infer protocol message formats. Tao et al. [19] proposed PRE-bin, which uses the Bayes decision model to determine field boundaries in binary protocols.

All the above methods, however, have drawbacks when applied in PRE binary protocols. The MSA algorithm used in PI compares sequence *byte by byte* whereas fields of a binary protocol may only contain a few bits. Discoverer [7] defines a token as a sequence of consecutive *bytes* that are likely to belong to the same application-level field. Tokenization depends on printable characters, which usually do not appear in binary protocols. Indeed, the authors pointed out that “for binary fields, identifying field boundaries is very hard; so we instead simply declare a single binary byte to be a binary token in its own right.” PRE-Bin [19] tried to solve the problem of identifying field boundaries in binary protocols by using multiple sequence alignment and the Bayes decision model. PRE-BIN, however, introduces many parameters that need a manual tune-up, making PRE-BIN extremely hard to use, particularly when people have no prior knowledge of which parameters work empirically the best.

This thesis is thus motivated to address the above difficulties.

1.2 Related Work

The current PRE methods mainly fall into two categories, network trace-based approach and execution trace-based approach. The former takes traffic captured on the network as input, whereas the latter assumes the programs running the protocol are accessible and takes the program binary code as input. Depending on how we process the input and output, we can further divide them into active/passive or protocol format/grammar [10]. The active method purposely sends messages to the network to create traffic, whereas the passive method only captures the traffic on the network without creating traffic. On the other hand, the output of PRE could be either a protocol structure/format which specifies the protocol fields or a grammar that includes the rules for message exchanges.

1.2.1 Network Trace-Based Approach

The PI [3] project developed idea of Multi-Sequence Alignment (MSA) which composed of Needleman-Wunsch [14] and Smith Waterman [17] algorithms. Packets are classified, and protocol formats are generated by aligning similar packets. Nevertheless, the PI project mainly identifies similar packets with clustering, and it needs users to determine the protocol fields manually. One automatic method that produces protocol format, called Discoverer [7], breaks each packet into a sequence of tokens, then recursively classifies the tokens. In the end, different token types are interpreted as different protocol fields. Another similar method, called ReverX [1], keeps a set of predefined delimiter characters and uses it to break a message into fields. It constructs two automata to model the protocol fields and the protocol grammar. Both ReverX and Discoverer only work on byte-aligned protocols and require a predefined symbol set. Another automatic method, PRE-Bin [19], handles the non-byte-aligned binary protocols by combining multi-sequence alignment and the Bayes' decision model. PRE-Bin, however, requires users to configure up to eight parameters manually.

1.2.2 Execution Trace-Based Approach

Polyglot [5] produces a protocol format and analyzes the binary code using a shadowing algorithm for PRE that collects all necessary information when the protocol program is running. AutoFormat [13] extracts different protocol fields by examining

different execution contexts such as run-time call stack for every message. MACE [6], on the other hand, builds a finite state model that learns from execution loops to infer the protocol grammar.

In summary, the network trace-based approach takes network traffic as input, whereas the execution trace-based approach takes program code as input. In this thesis, we focus on network trace-based PRE because it is the most viable solution for many real-world systems where we cannot access the end hosts for various reasons. For instance, most smart camera manufacturers offer web-based surveillance services where the remote server is located in the cloud and controlled by the service provider. Unlike most network trace-based methods that rely on classification, we define a general structure for TLV inference and directly identify the fields based on the most-likely matches in network traffic data.

1.3 The Problem to Be Solved

In this thesis, we aim at tackling the problem of inferring field boundaries in an unknown binary protocol. We omit ASCII fields since they can be accurately identified with existing methods [3, 7, 19] and mainly focus on binary fields. To start with, it is possible and definitely helpful to learn some prior knowledge by studying the protocol fields that commonly appear in many known protocols. One such prior knowledge is a broadly-used data structure called Type-Length-Value (TLV), i.e., the Type (T) field, followed by the Length (L) field, then followed by the Value (V) field. We use TLV as prior knowledge and generalize it to *extended TLV* (ETLV), whose formal definition will be given in Chapter 2, to cover a large corpse of protocol header formats.

1.4 Challenges and Contributions

Generalizing the concept of TLV to cover a broader category of protocols and automatically identifying the (extended) TLV fields pose several challenges that cannot be solved with existing PRE techniques.

- Extended concept about L : In the traditional version, L is a length field measuring the size of a block called value (V) field that follows *right after* the L field. Many protocols do not conform to this traditional definition. For instance, in TCP header, the header length has 4 bits in the unit of 4 bytes (e.g., 1010_2

means the header length is $40(= 10 * 4)$ bytes. Note that in the rest of the thesis, a number with subscript 2 means the number is in binary format. To generalize TLV for TCP header analysis, L is still a length field. However, the V field can be any block of fields and may be located anywhere in the packet as long as its relative distance from L is fixed. In addition, we need to allow L to measure the size of the V field plus some offset since we do not know what and where the V field is.

- Binary protocols: Since we are dealing with a binary protocol, any bit could be the starting bit of a TLV field, and a field may only contain a few bits. Inferring boundaries in the unit of bits is extremely challenging and time-consuming.
- Unit problem: The unit of the L field is uncertain. Assume that the size of V field uses 32 bits and the L field measures the size of the V . The L field will be 1_2 if L is measured in the unit of 4 bytes (like TCP), and will be 100_2 if L is measured in the byte unit. Hence, a different unit will lead to a different length of the L field. Unfortunately, for a private binary protocol, we do not know in advance the unit used in the measure.

In this thesis, we present a solution to tackle all the above challenges. The key idea is based on the following fact: the relative distance between the L and the V fields is usually constant for a given protocol. By identifying this constant across all packets, we could spot the possible locations for extended-TLV fields. Although calculating this constant requires the value of the L field and any bit could be the first bit of the L field in a binary protocol, the length of the L field must be bounded. Due to this reason, we can deduce not only the value of the L field but also the unit it uses. Leveraging this idea, we make the following contributions:

- We formally define the concept of extended TLV, in which the L field could potentially measure anything within the packets. In addition, we define a more general packet structure where the extended TLV can be applied. Not only does it cover the traditional TLV, but it can also extract the L field even when there is a constant offset in the V field.
- We systematically analyze the efficiency of our algorithm, called *Auto-ETLV*, in terms of complexity, convergence rate, and accuracy. The algorithm contains two main operations, *Generation* and *Verification*. *Generation* generates and

records all sets of possible values for our interests, and *Verification* quickly verifies the percentage of packets for which the recorded values hold. The final output will be a set of possible values. We also analyze the converge rate, i.e., how quickly the set of possible values converges to a small number of values.

- We implement and test *Auto-ETLV* over several binary protocols, including IP, TCP, Modbus, and MQTT. Our results show that it is efficient in terms of running time. It can also achieve high accuracy by identifying the exact *L* field for Modbus and returning a number of possible locations for the *L* field for TCP.

The main content of this thesis research has been published in the following paper.

- Zewen Huang, Kui Wu, Shengqiang Huang, Yang Zhou, and Ronnie Salvador Giagone, “Automatic Field Extraction of Extended TLV for Binary Protocol Reverse Engineering,” The 31st International Conference on Computer Communications and Networks (ICCCN 2023), July, 2022.

1.5 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2 gives a formal definition of ETLV and discuss how the problem of protocol field identification could be solved within the context of ETLV. We present the detail of the algorithm, called Auto-ETLV, and its complexity analysis.

Chapter 3 shows the experimental results of Auto-ETLV on several popular protocols, e.g., TCP, MQTT, Modbus, and IP.

Chapter 4 presents an interactive application that implements Auto-ETLV and help users test Auto-ETLV with different parameters. We build a Graphic User Interface (GUI) to facilitates the protocol reverse engineering task.

Chapter 5 discusses the importance and limitation of ETLV structure. We also present the possible extension of our algorithm to detect nested ETLV.

Chapter 2

Extended TLV (ETLV) and Solution

In the chapter, we introduce our solution to the problem raised in the previous chapter and explain how our algorithm works to extract ETLV structure from a private binary protocol. Since ETLV is a structure extended from the TLV structure, we first introduce TLV and then formally define ETLV. We will discuss how the previous questions appear in the context of ETLV and how we solve those questions. We also analyze the running complexity of our algorithm.

2.1 Background and Problem Definition

2.1.1 What is TLV?

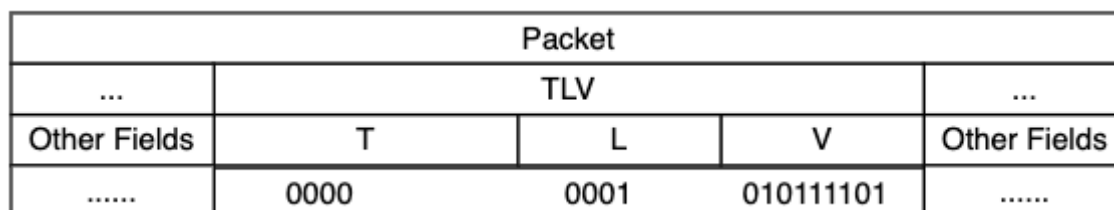


Figure 2.1: TLV structure

Type-length-value (TLV) is a broadly-used scheme for encoding information in communication protocols. It's also referred to as tag-length-value or key-length-value.

TLV-encoded packets include the type code, followed by the length of value, and then the value itself. This encoding scheme has been used in a large class of protocols such as SSL/TLS [9], RADIUS [8], and LLDP [2]. A canonical form of the TLV encoding scheme uses *consecutive* protocol fields as follows:

$$P = |\dots|T|L|V|\dots| \quad (2.1)$$

where P denotes a packet, T denotes the type code, L denotes the length field, V denotes the value field, and \dots represents the rest parts of the packet.

As network protocols evolve, TLV may appear in various forms. For example, the scheme may appear as LTV [12], which is the same as TLV except that the positions of L and T are swapped. As another example, the CoAP protocol [4] uses a TLV-like encoding scheme, where V is a token field, and L measures the length of the token. But V does not immediately follow the L field, and there are other fields between the L field and the V field. In this example, the L field and the V field are not consecutive blocks. They are separated with a fixed number of bits in between. If we relax the canonical definition of TLV, we can cover a much broader category of protocols into this “TLV” encoding scheme. For example, if we relax the requirement by assuming that the V field is as large as the packet header, TCP header can fit into the relaxed TLV encoding scheme since the L field measures the entire TCP header. Note that we do not know the boundary between TCP header and data in PRE, but we can still infer the V field if the trace includes diverse TCP packets, some of which may not include data, e.g., TCP SYN and FIN packets.

The above observation motivates us to extend the canonical TLV form, based on which we can build a powerful tool for reverse engineering a large group of unknown protocols.

2.1.2 Extended TLV (ETLV)

To help better understand our extended TLV scheme, we first introduce several functions.

- Length function $\mathcal{L}(x)$: It determines the number of bits used to represent binary string x , e.g., $\mathcal{L}(101_2) = 3$.
- Value function $\mathcal{V}(x)$: It determines the value of binary string x in decimal, e.g., $\mathcal{V}(101_2) = 5$.

- Binary function $\mathcal{B}(d)$: It converts decimal d to binary string, e.g., $\mathcal{B}(5) = 101$.

With the above functions, we formally define the extended TLV (ETLV) as follows.

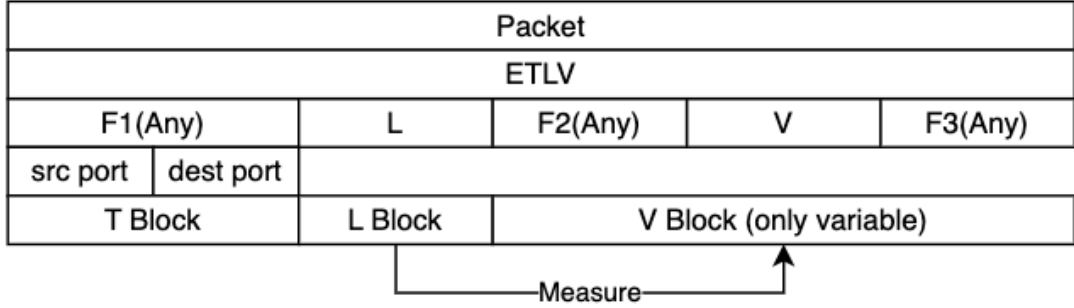


Figure 2.2: ETLV structure

Definition 1. *Extended TLV (ETLV)*: Assume that P is a packet in the format of $P = |F_1|L|F_2|V|F_3|$, where $\mathcal{L}(F_1)$, $\mathcal{L}(L)$, $\mathcal{L}(F_2)$, and $\mathcal{L}(F_3)$ are all constant. In addition, if P satisfies any of following conditions:

1. Case 1: $\mathcal{V}(L) = \mathcal{L}(V)$,
2. Case 2: $\mathcal{V}(L) = \mathcal{L}(P)$,
3. Case 3: $\mathcal{V}(L) = \mathcal{L}(V) + C$, where C is a constant (offset),

we say that the packet P follows the extended TLV (ETLV) structure.

In the ETLV packet structure, we introduce three blocks (a block is 0 or more consecutive protocol fields) F_1, F_2, F_3 , each of which may contain any number (including zero) of bits, for example F_1 may contain two protocol fields, source port and destination port as shown in figure 2.2. The F_1 block means that the L field locates at a fixed location relative to the start of the packet since the L field usually does not appear at the beginning of a packet. The F_2 block separates the L field and the V field so they can stay apart by a fixed distance. The F_3 block allows any extra fixed-size information at the end of the V field, since a protocol may not end with the V field. Note that we do not explicitly define a T field, implying that the T field could be in any of the F_1, F_2, F_3 blocks. In addition, we assume that F_1, F_2, F_3 , and

L all have fixed length, including zero. In particular, when the size of L is zero, i.e., $\mathcal{L}(L) = 0$, we have $\mathcal{L}(V) = 0$ since we do not have a L field. In this special case, the packet P would only contain F_1, F_2, F_3 , suggesting that P has a fixed size.

Next, we use examples to explain different cases in the ETLV structure.

- Examples of Case 1: This case is similar to the canonical TLV, where the L field tells the length of the V field. All protocols following the canonical TLV are covered by this case. For example, in ModBus and MQTT protocol [18] the value of the length field measures the size of remaining fields.
- Examples of Case 2: This case allows the L field to measure the size of the entire packet. One example is the IP protocol where the length field tells the size of the whole packet.
- Examples of Case 3: This case covers special situations where the L field may not measure the size of the V field exactly; instead, there is a fixed gap. For example, in CoAP, there are two extra fields between the token length field and the token field. On the other hand, if the V field contains some data, and the L field specifies the size of a buffer that the receiver should allocate, $\mathcal{L}(V)$ could be larger than the actual size of the V field. Some proprietary protocols fall in this category [11].

Remark 1. *We assume that a given protocol can only follow one of the above cases, but we do not know exactly which case the protocol follows. This assumption is reasonable since a protocol usually grants the unique meaning of an L field.*

Remark 2. *The definition of ETLV uses the \mathcal{V} function, which assumes the unit in bits. Yet, we DO NOT know in advance the unit of the L field used in an unknown protocol, so we cannot tell its value $\mathcal{V}(L)$ in terms of bits. For instance, the L field in the TCP header is measured in the unit of 4 bytes. Therefore, the three cases described above could all fail if we cannot automatically infer the right unit used in the protocol. We solve this problem in our later algorithms.*

One more thing to notice, we can dedicate the length field to L block, and every fields come before L block to T block, and every fields come out L block to V block, then we will have a blocked TLV structure as shown in figure 2.2. There are some difference between the blocked TLV and canonical TLV described in TLV section. Firstly, blocked TLV structure is at protocol level, which means we partition the

entire packet as TLV blocks whereas canonical TLV is at protocol fields level, which means the structure made of three protocol fields is only part of the protocol structure. Secondly, since L field and V field inside blocked TLV can stay apart from each other so that L block in blocked TLV is actually measuring the size of V block, which means it is the size of V field plus some offset.

With ETLV defined and explained, our goal in the rest of this chapter is to answer all the questions raised in previous chapter and to develop a tool, called, Auto-ETLV, that automatically extracts the L field/block in an unknown binary protocol following the ETLV packet structure

2.2 Problem Analysis

To automatically extract the information about the L and V fields from an unknown binary protocol, we need to tackle the following challenges. First, the boundaries of each field are unknown and may appear at any bits in a packet. Second, different protocols may use different units when calculating the value of the L field. Third, while we assume that a given protocol can only use one of the three cases in ETLV, we do not know whether or not the protocol actually follows, and even it does, we do not know which exact case of ETLV it uses.

To address the above challenges, we first analyze the properties of the \mathcal{V} and \mathcal{L} functions. Based on the analysis, we then solve the unit problem when calculating the value of the L field. Finally, we develop a method to find out the most likely ETLV case that the protocol follows.

2.2.1 The Property of $\mathcal{V}(\hat{L}), \mathcal{L}(\hat{L})$

Usually not all bits in the length field are used to represent the size of the value. For example, to represent a value 7, the L field of 5-bits would be 00111_2 . In other words, only the last three bits are used for calculation. We call the **effective length field** \hat{L} as the binary string of the L field excluding preceding 0s. Note that $\mathcal{V}(L) = \mathcal{V}(\hat{L})$.

Lemma 1. *If a protocol uses ETLV and the effective length field \hat{L} is in unit of bits, then any packet P satisfies the following constraint:*

$$\mathcal{L}(\hat{L}) < \mathcal{L}(\mathcal{B}(\mathcal{L}(P))) + 1, \quad (2.2)$$

where \mathcal{L} , \mathcal{V} , and \mathcal{B} are the functions defined in Section 2.1.2.

Proof. Since the protocol follows ETLV, we only need to verify Equation (2.2) in the three cases defined by ETLV.

1. Case 1: $\mathcal{V}(L) = \mathcal{L}(V) < 2\mathcal{L}(P)$.
2. Case 2: $\mathcal{V}(L) = \mathcal{L}(P) < 2\mathcal{L}(P)$.
3. Case 3: $\mathcal{V}(L) = \mathcal{L}(V) + C < \mathcal{L}(P) + \mathcal{L}(P) = 2\mathcal{L}(P)$. Note that C is a constant offset and should not be larger than $\mathcal{L}(P)$.

Therefore, we have $\mathcal{V}(L) < 2\mathcal{L}(P)$ in all cases. Thus, the number of bits that are needed to represent $\mathcal{V}(L) = \mathcal{V}(\hat{L})$ is at most the number of bits used to represent $2\mathcal{L}(P)$, or equivalently, $\mathcal{L}(\hat{L}) < \mathcal{L}(\mathcal{B}(\mathcal{L}(P))) + 1$. \square

Note that even if L is not measured in bits, Lemma 1 still holds since $\mathcal{V}(L)$ can only become smaller if measured in other units.

2.2.2 The Unit Problem

Based on Lemma 1, we know the upper bound of $\mathcal{L}(\hat{L})$. But we still do not know how to determine the actual unit of the L field. Without knowing its unit, we cannot determine $\mathcal{V}(L)$ and thus cannot tell if a packet falls into any of the three cases of ETLV.

We use two examples to illustrate the unit problem. If a field F uses 64 bits and the corresponding L field tells the length of this F field, then $L = 1000000_2$ if the unit of L is bits, $L = 100000_2$ if the unit is 2 bits, $L = 10000_2$ if the unit is 4 bits, $L = 1000_2$ if the unit is in bytes, and so on. If the field F uses 40 bits and the corresponding L field tells the length of this F field, then $L = 101000_2$ in bits, $L = 10100_2$ in 2 bits, $L = 1010_2$ in 4 bits, $L = 101_2$ in bytes, and so on. From the above examples, we can see that no matter what unit a binary string uses, only the number of trailing zero's matters. Everything before the trailing zeros is unchanged. We call the bits before the trailing zeros **core bits**. The core bits in the first example are 1_2 , and the core bits in the second example are 101_2 .

Therefore, given a L field binary string, we can shift its **core bits** to the left to get the length values in different units. The good news is that $\mathcal{L}(\hat{L})$ is bounded based on Lemma 1. Thus the number of possible shifts/units that we need to check is also

bounded. If any shift matches one of three cases across all packets, then we know this shift value implies the most-likely unit used by the L field, which is in 2^{shift} bits.

2.2.3 The Property of $\mathcal{V}(L), \mathcal{L}(\hat{L})$ in Different Units

We refer the *shift* mentioned above as *unit size*, e.g., unit size = 0 means that the \hat{L} field is measured in $2^0 = 1$ bit. Lemma 1 tells the bound of the length of the \hat{L} field in the unit of bits. Next, we show that a tighter bound if a different unit is used.

Lemma 2. *If a protocol uses ETLV, then the effective length field \hat{L} of packet P , satisfies the following constraint:*

$$\mathcal{L}(\hat{L}) < \mathcal{L}(\mathcal{B}(\mathcal{L}(P))) - \text{unit size} + 1, \quad (2.3)$$

no matter what unit \hat{L} uses.

Proof. No matter what the unit \hat{L} uses, if we shift \hat{L} to left with *unit size*, $\mathcal{V}(\hat{L} \ll \text{unit size})$ will return the value of \hat{L} in bits. In other words, $\mathcal{L}(\hat{L}) + \text{unit size} = \mathcal{L}(\hat{L} \ll \text{unit size})$ since both measures the same field in bits. For instance, if \hat{L} uses byte as the unit and $\hat{L} = 100_2$ (i.e., 4 bytes), then $\mathcal{V}(\hat{L} \ll \text{unit size}) = \mathcal{V}(10000_2) = 32$ (i.e., 32 bits). Since Lemma 1 is true when the unit is in bits, we have

$$\mathcal{L}(\hat{L}) + \text{unit size} = \mathcal{L}(\hat{L} \ll \text{unit size}) < \mathcal{L}(\mathcal{B}(\mathcal{L}(P))) + 1.$$

□

2.2.4 Constant Difference

From the previous analysis, given an L field in a binary string, we may have a small set of shifted values, and one of them will be the correct version of $\mathcal{V}(L)$ in terms of bits. We have the following lemma that can help us make the right decision.

Lemma 3. *Assume that a protocol uses ETLV and its packet $P = |F_1|L|F_2|V|F_3|$. Denote the blocks after the L field as $R = |F_2|V|F_3|$. Then, $\mathcal{L}(R) - \mathcal{V}(L)$ is a constant.*

Proof. We only need to verify the three cases defined in ETLV.

1. Case 1: Since $\mathcal{V}(L) = \mathcal{L}(V)$, we have

$$\begin{aligned}\mathcal{L}(R) - \mathcal{V}(L) &= \mathcal{L}(F_2VF_3) - \mathcal{L}(V) \\ &= \mathcal{L}(F_2) + \mathcal{L}(V) + \mathcal{L}(F_3) - \mathcal{L}(V) \\ &= \mathcal{L}(F_2) + \mathcal{L}(F_3)\end{aligned}$$

2. Case 2: Since $\mathcal{V}(L) = \mathcal{L}(P)$, we have

$$\begin{aligned}\mathcal{L}(R) - \mathcal{V}(L) &= \mathcal{L}(R) - \mathcal{L}(P) \\ &= (\mathcal{L}(P) - \mathcal{L}(F_1L)) - \mathcal{L}(P) \\ &= -\mathcal{L}(F_1) - \mathcal{L}(L)\end{aligned}$$

3. Case 3: Since $\mathcal{V}(L) = \mathcal{L}(V) + C$ where C is a constant, we have

$$\begin{aligned}\mathcal{L}(R) - \mathcal{V}(L) &= \mathcal{L}(R) - (\mathcal{L}(V) + C) \\ &= \mathcal{L}(F_2) + \mathcal{L}(V) + \mathcal{L}(F_3) - (\mathcal{L}(V) + C) \\ &= \mathcal{L}(F_2) + \mathcal{L}(F_3) + C\end{aligned}$$

We can see that no matter which case a packet follows, $\mathcal{L}(R) - \mathcal{V}(L)$ always remains constant. \square

Utilizing the above analytical results, we next develop the *Auto-ETLV* tool that automatically extracts the L block from a binary protocol if it follows the ETLV structure.

2.3 Algorithm Design in Auto-ETLV

2.3.1 A High-Level Overview

For an unknown protocol, the L field may appear anywhere in a packet, and as such, we have to loop through each bit of the entire packet. Based on previous analysis, for each bit, we only look at $\mathcal{L}(\mathcal{B}(\mathcal{L}(P))) + 1$ consecutive bits (called *check string* and denoted by cs), since $\mathcal{L}(L)$ is bounded by this number. Then we generate each suffix from cs . For each suffix, we can apply the left shift operation to get different possible values for different units. Then we use the shifted string to calculate

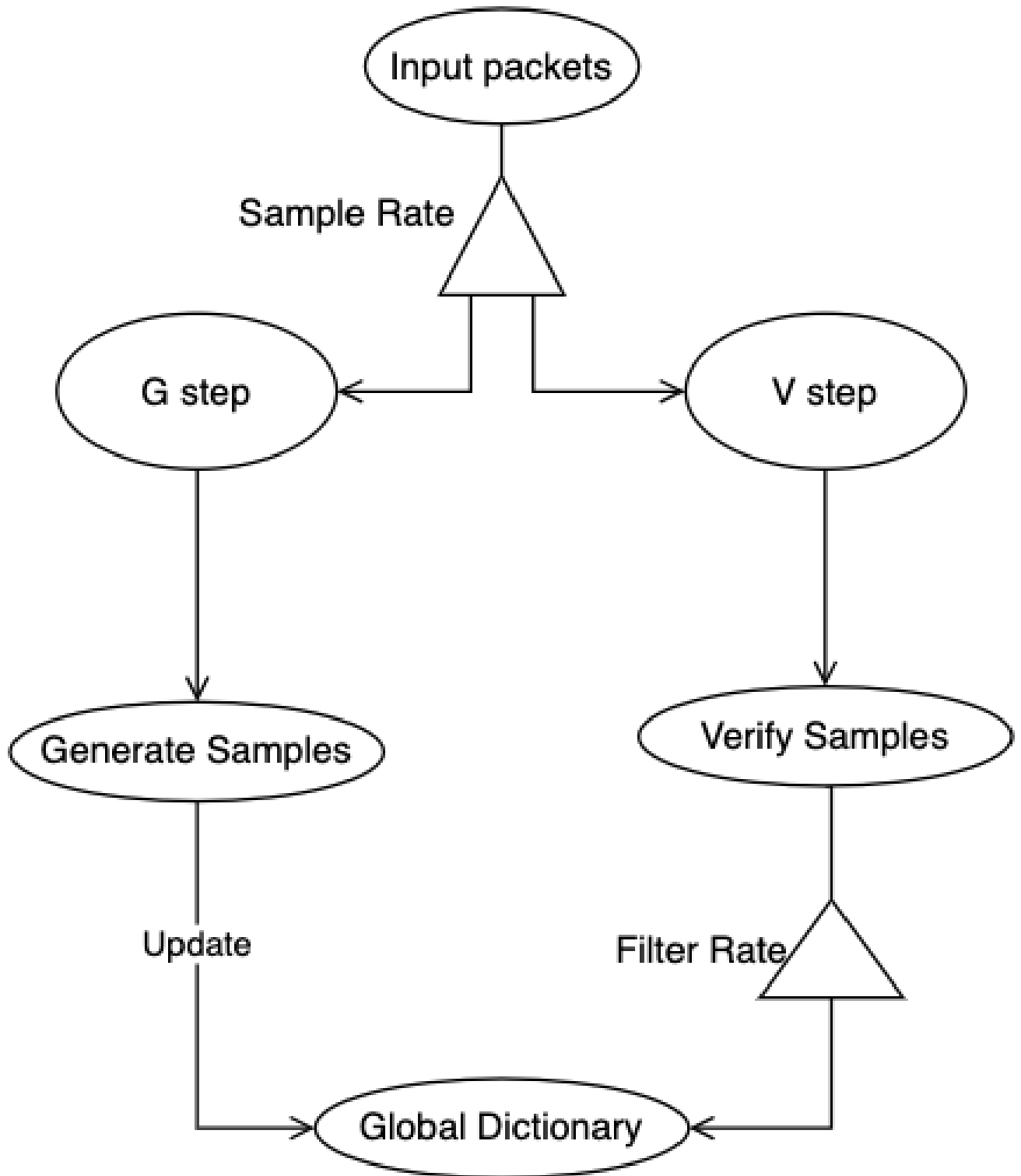


Figure 2.3: Algorithm Flow

$\mathcal{V}(\text{shifted string}) - \mathcal{L}(\text{remaining bits})$, and we call this value the *key difference*. If all packets have the same *key difference* at a certain bit position (*bpos*) and certain shift number. Then we can infer that this *bpos* is a possible location for the length field, with the unit in 2^{shift} bits.

While the above idea seems straightforward, it requires us to generate many *key differences* for each location. Denote $b = \mathcal{L}(P)$. We then need to loop through b bits. For each bit, we look $\log(b)$ bits *check string* and generate $\log(b)$ suffixes. For each suffix, we have $\log(b)$ shifts. Thus we have $O(b \log(b)^2)$ possible *key differences* for every packet. This is a huge number! Nevertheless, we will show that as the number of packets fed to the algorithm increases, this number will drop dramatically with a proper filtering mechanism to exclude the impossible bit positions.

Next, we introduce the algorithm details for extracting the length field from ETLV. As shown in Figure 2.3, the algorithm is mainly composed of two steps: the Generation (G) step and the Verification (V) step, and only one of them is executed for each packet. The G step is executed with small probability called sample rate since it takes most of time to generate all *key differences* at every bit position for the L field. The V step is executed with high probability since it only verifies what we have generated from the G step and filters out impossible bit positions which controlled by filter rate. The above generation-verification method can reduce the running time and the number of possible bit positions for the L field dramatically.

2.3.2 Global Dictionary

The core data structure we use is the global dictionary (GD). Each entry in the GD has a key field and a value field. The key field consists of $(bpos, shift)$, and the value field consists of $(index, Kcounter, a\ list\ of\ tuples)$, where each *tuple* is represented by $(key\ difference, l, r, Dcounter)$.

Recall that we loop through all bits in a packet, and for each bit position *bpos*, we have a *check string*, with which all suffixes are generated. For each suffix, we apply the left shift operation to generate the values of the binary string of this suffix in different units. Then we use each value of the shifted suffix to calculate the *key difference*. Note that for each suffix, they can be shifted different times, thus for each $(bpos, shift)$ combination, we will have a set of *key differences*. We need to store all the information for every $(bpos, shift)$ combination. Thus we use $(bpos, shift)$ as the key of the GD.

Algorithm 1: Main Structure

```

input : a set of packets
output: the possible locations for the  $L$  field
begin
  GD=;
  for  $index, p$  in  $enumerate(Packets)$  do
    if  $uniform(0, 1) < \frac{1}{index+1}$  then
       $\lfloor$  Generation( $p, index$ );
    else
       $\lfloor$  Verification( $p, index$ );
   $\lfloor$  return GD.maxDCounter();

```

The value field includes a list of *tuples*, each denoted by (*key difference*, l , r , *Dcounter*). Note that *key difference* is calculated by $\mathcal{V}(\text{shifted string}) - \mathcal{L}(\text{remaining bits})$, and l and r together denote the inferred interval (i.e., the left bit position and the right bit position, respectively) for the left boundary of the L field. We also include *index* to record the timestamp when this GD entry is created and *Kcounter* to record the number of times that the *list of tuples* has been updated. *Dcounter* is another counter recording the number of times that the same *key difference* value has been encountered.

2.3.3 Main Structure of *Auto-ETLV*

Algorithm 1 shows the main structure of our solution. It first creates a global dictionary to store all information for possible length field locations generated from the G step. In the main loop, the algorithm chooses one of the two steps for each packet with probability. As the number of packets increases, the probability that we choose the G step becomes smaller. It is worth noting two facts. First, the V step only verifies all possible locations generated from the G step, so the V step will not take any effect until we run the G step. Second, we use a dynamically adjusted probability for executing the G step, i.e., $\frac{1}{index+1}$ in Algorithm 1. While this method leads to good experimental results, users may use a different probability value. No matter which probability value to use, we need to ensure that the G step is executed at least several times because otherwise, there is nothing to verify in the V step.

In the end, the *Dcounter* reflects the likelihood that the represented bit location is the correct location of the L field. After processing all packets, we report the

Algorithm 2: Generation

```

input : p: a packet
         index:  $i$ -th packet
begin
  cl =  $\mathcal{L}(\mathcal{B}(\mathcal{L}(p)))+1$ ;
  for  $0 \leq bpos \leq \mathcal{L}(p)$  do
    cs = p[max(0,bpos-cl), bpos];
    r =  $\mathcal{L}(P)-bpos$ 
    suffix_set = a set of suffixes of cs
    diff_dic = {}
    for suffix in suffix_set do
      for  $0 \leq shift \leq cl - \mathcal{L}(suffix)$  do
        key = (bpos, shift)
        item = (#Of preceding zeros of suffix,
               $\mathcal{L}(suffix)$ ,
               $\mathcal{V}((suffix \ll shift) - r)$ ), 1)
        diff_dic[key].append(item)
    for key, val in diff_dic do
      GD.update(index, key, val, append = True )
  
```

location(s) that have the highest *Dcounter* value(s).

2.3.4 Generation Step

In the Generation step (Algorithm 2), we loop through all bits in a packet. For each bit position (*bpos*), we calculate all *key differences* for different possible shifts. Since the *L* field is bounded, the check string (cs) for generating possible differences is also bounded. If we assume that the current *bpos* is the bit right after the *L* field, then the possible *L* fields are all suffixes of cs. Furthermore, we need to infer the unit of the length field. The good news is that the *L* field is bounded; hence we can determine all possible values for all possible units by shifting the suffix to the left, bit by bit, for the length of cs times. After this, we have all possible values for possible units at this *bpos*. Accordingly, we can calculate and update the difference to the global dictionary.

Note that in the G step, we need to consider a list of (*key difference, l, r, Dcounter*). The fields *l, r* together denote the inferred range of possible length for the *L* field. Intuitively, since each suffix could be a potential *L* field, the length of the true *L*

Algorithm 3: GD.update

```

input : index:  $i$ -th packet
         key: (bpos, shift)
         val: a list of key difference
         append: bool variable

begin
  if  $key$  in  $GD$  then
    for  $v$  in  $val$  do
      for  $gv$  in  $GD[key][1]$  do
        if  $v.diff == gv.diff$  then
           $gv.l = \min(gv.l, v.l)$ 
           $gv.r = \max(gv.r, v.r)$ 
           $gv.Dcounter++$ 
        else if  $append == True$  then
           $GD[key][1].append(v)$ 
      if  $gv$  ever updated then
         $GD[key][0].Kcounter++$ 
  else
     $GD[key] = ((index, 1), val)$ 

```

field is at least as large as the length of the suffix. On the other hand, not all bits in the L field are usually used in a packet (e.g., the first several bits of the L field are 0's). Thus the length of the true L field is at most as large as the length of the suffix plus the preceding zeros. The *key difference* is calculated by $\mathcal{V}(\text{shifted string}) - \mathcal{L}(\text{remaining bits})$, and *Dcounter* counts how many times we have encountered the same *key difference* value. The timestamp records when this key was added to the global dictionary for the first time.

We also allow *bpos* to be as high as the length of the packet ($\mathcal{L}(p)$). This is because *bpos* is assumed to be the bit right after the L field, and we need to handle the (rare) case that a packet ends with the L field, i.e., no bit after the L field. Allowing *bpos* to be $\mathcal{L}(p)$ will not cause any problem, since we only process bits right before the *bpos*.

2.3.5 GD update

Algorithm 3 will be called in both the G and V steps. The only difference is that this function could increase the size of the list of *tuples* under the given key in the G step, but not in the V step. To update the GD, if the key does not exist in GD, we set

its first part of dictionary value as $(index, 1)$, where $index$ works as the timestamp indicating when this key is created and 1 is the current value of $Kcounter$. Note that we use packet index as the timestamp because they mean the same thing for packet trace analysis. If the key already exists in the GD, we first increase $Kcounter$ by 1. Then we check if the list of tuples already includes the same *key difference*. If yes, then we increase the corresponding $Dcounter$ by 1 and update l and r . Otherwise, we add a new *tuple* $((key\ difference, l, r, 1))$ to the list of *tuples*, where $Dcounter$ is set to 1.

To update l and r , imagine that we have two L fields from two packets. Let s_1 and s_2 be such two binary strings representing the two L fields. Then the length of the L field will be at least $\max(\mathcal{L}(s_1), \mathcal{L}(s_2))$, and at most $\min(\mathcal{L}(\text{preceding zeros} + s_1), \mathcal{L}(\text{preceding zeros} + s_2))$.

2.3.6 Verification Step

The verification step (Algorithm 4) is almost the same as the G step, except that we only verify what we have generated from the G step. Thus we only loop through the global dictionary. Since the key contains $bpos$ and $shift$, we can generate all values for the specific unit calculated from the $shift$ value. Then we can use the values to calculate the differences.

We use an additional condition to clear up GD at the end of the verification step. The $Kcounter$ tells the number of times we have encountered this key in the global dictionary. The $index$ records the first time that this key was added to GD. The last condition check in Algorithm 4 determines whether the number of encountered times is less than 90% of times since it appeared the first time. The value of 90% is empirical. The idea is that if this key is not encountered often, then this key should be dropped from GD. Note that if the L field matches the size of the V field in all packets, then the number of times that the key encountered for the correct L field passes the verification should be 100%. If any packet has the L field mismatching the V field, then the number of times the key encountered for the correct L field will not be 100%. Note that this cleanup step is to speed up future processing by reducing the size of GD.

Algorithm 4: Verification

```

input : p: a packet
         index:  $i$ -th packet
begin
  cl =  $\mathcal{L}(\mathcal{B}(\mathcal{L}(p)))+1$ ;
  for  $bpos, shift$  in  $GD.key$  do
    cs = p[max(0,bpos-cl), bpos];
    r =  $\mathcal{L}(P)$ -bpos
    suffix_set = a set of suffixes of cs
    diff_dic = {}
    for  $suffix$  in  $suffix\_set$  do
      if  $shift \leq cl - \mathcal{L}(suffix)$  then
        key = (bpos, shift)
        item = (#Of preceding zeros of suffix,
                $\mathcal{L}(suffix)$ ,
                $\mathcal{V}((suffix \ll shift) - r)$ , 1)
        diff_dic[key].append(item)
    for  $key, val$  in  $diff\_dic$  do
      GD.update(index, key, val, append = False )
      timestamp = GD[key].Index
      hit = GD[key].Kcounter
      if  $(index - timestamp)*0.9 > hit$  then
        GD.remove(key)
  
```

2.3.7 Time Complexity

We analyze the time complexity of processing each packet. The required storage space is essentially the size of the global dictionary. The space cost is negligible, considering that the number of possible combinations for the key field is not large for any given protocol.

Let b be the size of the largest packet, d be the number of entries (keys) in the global dictionary, and k be the largest number of tuples in the value fields of the global dictionary.

Complexity of the G Step

In the G step, we need to loop through all the bits, and for each bit, we need to generate all possible suffixes of cs . Since cs is bounded by $\log(b)+1$, we have $\log(b)+1$ suffixes and it takes $O(\log(b)^2)$ time to process them. For each suffix we can shift

$\log(b) + 1 - \mathcal{L}(\text{suffix}) = O(\log(b))$ times to get different values in different units, that are used to calculate the *key differences*. So it will take $O(\log(b)^2)$ time.

For updating the l value, it requires the knowledge about preceding zeros at current $bpos$. However, this value can be pre-computed by simply looping through all bits once, which takes $O(b)$ time.

For the *GD.update* function, we need to compare the list of *key differences* generated from the current packet and the list of *key differences* in the GD. Since the maximum size of the former list is $O(\log(b)^2)$, and the maximum size of the latter list is $O(k)$, the comparison takes $O(k \log(b)^2)$ time.

In total, the G step takes $O(bk \log(b)^2)$ time.

Complexity of the V Step

In the V step, we need to loop through all entries in the global dictionary. Since d is the number of entries in the global dictionary, we can follow the similar analysis as in the G step to conclude that the V step takes $O(dk \log(b)^2)$ time.

Note that the V step will be executed more frequently than the G step. In addition, since it can quickly reduce the size of GD due to the filtering mechanism, the value of d is expected to be small.

2.4 Why Does Auto-ETLV Work?

The success of Auto-ETLV relies on two core lemmas. Lemma 2 bounds the size and unit of the length field to eliminate unnecessary computation. Lemma 3 describes the relationship between a block of binary sequence and the length of remaining bits. Auto-ETLV actually looks for such relationship in packets with additional bound constraints, and the verification counter measures how frequent this relationship holds across all packets. The filter rate controls the intense level of the relationship stored in the global dictionary so that every record whose verification counter isn't higher than the filter rate will be removed. From that point of view, the length field is just a special block of binary sequence that satisfies all the constraints and has the strongest relationship.

Chapter 3

Experiments

In this chapter, we use *Auto-ETLV* to analyze the packet structure of several protocols, pretending that we do not know their specification in advance. We compare the results of *Auto-ETLV* and the ground truth (i.e., protocol specification) to demonstrate the efficiency and effectiveness of our solution.

3.1 Evaluation Results

We test *Auto-ETLV* on several well-known protocols: ModBus, IP, TCP and MQTT. We pretend that we know nothing about these protocols. For each protocol, we collect 2000 packets, which are used as the input to *Auto-ETLV*.

3.1.1 Accuracy

Table 3.1 shows the locations for the L field of a protocol proposed by *Auto-ETLV* and the ground truth. The proposed location is labeled by $[(x_1, x_2), x_3](x_4)$, where (x_1, x_2) denotes the left boundary of the L field (i.e., the left boundary should fall in the range from x_1 -th bit to x_2 -th bit), x_3 denotes the right boundary of the L field, and x_4 is the unit of the L field. The ground truth is denoted by $[y_1, y_2](y_3)$, where y_1 and y_2 are the left boundary and the right boundary of the L field, respectively, and y_3 is the unit of the L field.

From the table, we see that *Auto-ETLV* can produce the correct locations for correct boundaries and units for IP, ModBus, and MQTT. It also produces a small interval containing the correct left boundary of the L field. Note that identifying the exact left boundary of the L field is difficult if no packet in the input data uses

Table 3.1: Location of the L Field

Protocol	Proposed Locations (unit)	True Location (unit)
TCP	[(96, 96), 99] (32 bits) [(96, 96), 100] (16 bits) [(96, 96), 101] (8 bits) [(96, 96), 102] (4 bits) [(96, 96), 103] (2 bits) [(96, 96), 104] (1 bit)	[96, 99] (32 bits)
IP	[(21,14),31] (8 bits)	[16,31] (8 bits)
ModBus	[(16, 40), 47] (8 bits)	[32, 47] (8 bits)
MQTT	[(4, 9), 15] (8 bits)	[8, 15] (8 bits)

The L field is represented by [the range of the left boundary, the location of the right boundary].

the maximum length. In this case, the prefix of the L field will be filled with zeros, and we do not have enough information to figure out the exact location of the left boundary.

TCP Header/(ETLV)							???
					Measure →	Variable Size	Variable Size
0,,,16	16,,,32	32,,,64	64,,,96	96,,,100	100,,,106	106,,,
source port	dest port	seq num	ack num	Length	reserved	Data(optional)
				1001	000000		
				1001		4 bytes	
				1001	0	2 bytes	
				1001	00	1 bytes	
				1001	000	4 bits	
				1001	0000	2 bits	
				1001	00000	1 bits	
				all same values			

Figure 3.1: The format of TCP.

It is interesting that *Auto-ETLV* returns six possible locations for the right boundary of the L field in TCP. This is not an error but instead discloses the complex TCP header structure. This result is because TCP has six reserved bits right after the L field, which are set to zeros in all the TCP packets of our input data. In this case, any of 6 bits can be reasonably interpreted as a valid right boundary of the L field, where different right boundaries correspond to different units, i.e., in Figure 3.1 1001_2 corresponds to 4 bytes, 10010_2 corresponds to 2 bytes, 100100_2 corresponds to 1 byte, and 1001000_2 corresponds to 4 bits. In addition, the interval of the left boundary (96, 96) indicates the exact location of the left boundary of the L field.

Note that the TCP header certainly follows ETLV structure since the length field measures the entire header, but TCP does not. There is an extra variable-sized data block that comes after the TCP header. Since it is not a fixed-length block, the key difference will not be computed correctly even if the current bpos points to the correct length field location. However, there is a portion (roughly 30%) of TCP packets that do not come with data (data block with size 0) in the input packets. Thus the Dcounter will be correctly updated with 30% of times. This is high enough to be recognized as a meaningful output as long as we do not filter out this information (controlled by the filtering rate discussed later). From the TCP example, we can see that even if only a portion of packets (e.g., only TCP packets without data block)

follow the ETLV structure, our method can still correctly identify the L field.

Auto-ETLV has two parameters to set: the first one is the probability of calling the G step, and the second one is the proportion of the total number of packets that determines when we should drop a key from the global dictionary. The default value for the first is set dynamically to $\frac{1}{1+packet\ index}$, and the default value for the second is set to 0.9. Before we discuss the impacts of these two parameters, we show the dynamic changes of the global dictionary and the maximum $Dcounter$ in the global dictionary.

3.1.2 Dynamics of GD

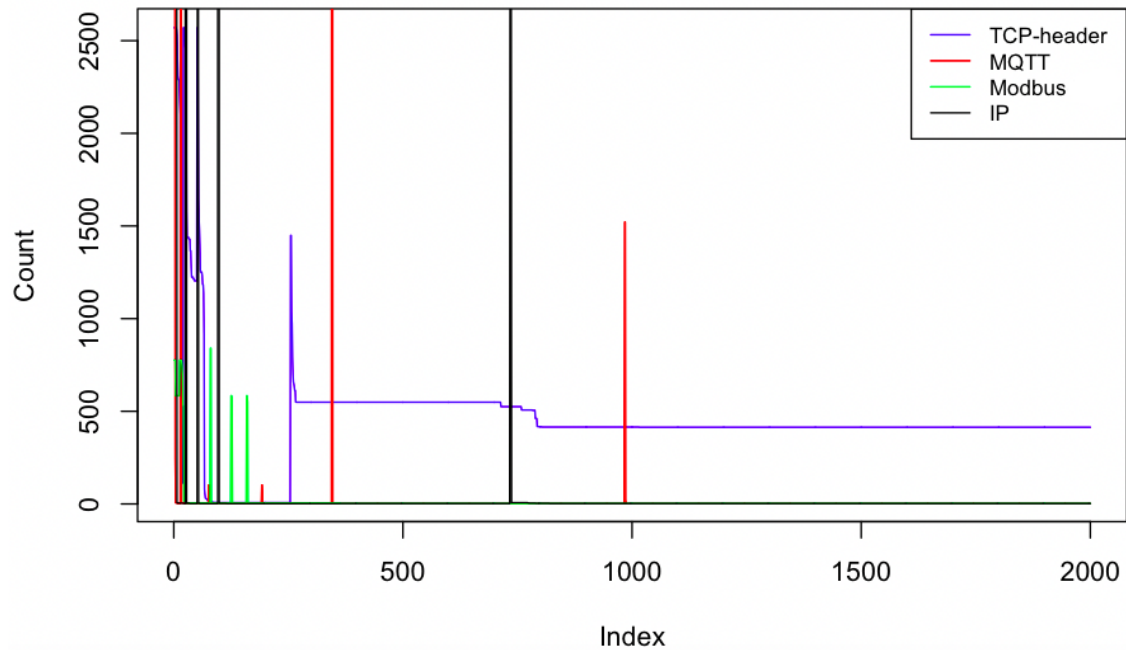


Figure 3.2: Size of global dictionary as we process more packets

Recall that we have a filtering mechanism that reduces the size of the GD in the V step. Figure 3.2 shows the changes in the size of the GD vs. the total number of parsed packets. Every spike in the GD size indicates execution of the G step, since the G step is the only step that generates *key difference* information at different *bpos*. We can also see that the GD size drops shortly after every G step.

Although we use 2000 packets for each protocol, it may be unnecessary to use so

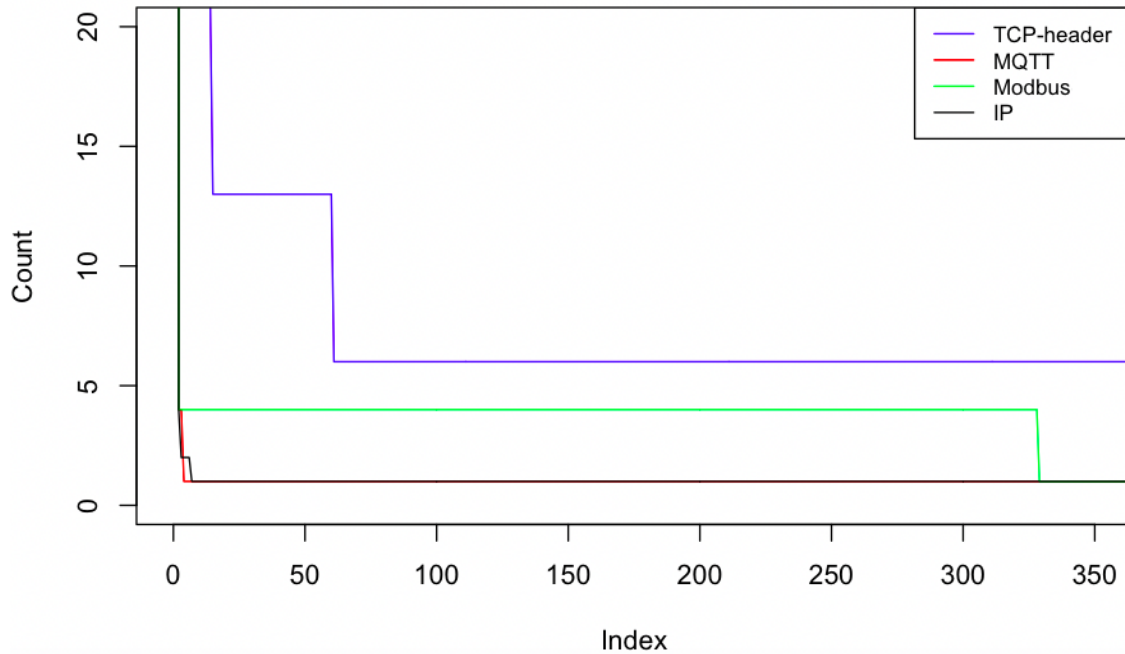


Figure 3.3: Maximum Dcounter after processing each packet

many packets to get a stable answer. In fact, some protocols only take less than 60 packets to get the final result. Figure 3.3 shows the size of *key difference* information with the highest Dcounter value for each protocol as the number of parsed packets increases. For MQTT, *Auto-ETLV* only takes 4 packets to reduce 11212 possible *key difference* to the only possible solution, and after that this answer remains unchanged. Of course, the threshold for the number of required packets depends on the diversity of packets, e.g., different packet sizes. Our experimental results suggest that, in general, tens or hundreds of packets should be enough to obtain a stable answer for most protocols.

3.1.3 The Impact of Parameters

Auto-ETLV uses two parameters: the probability for executing the G step and the filtering rate. The first parameter determines the probability that the G step is executed for a packet. The second parameter specifies a proportion value, and if a dictionary key is updated less frequently than this value since this key was created, we drop the key from the GD. To assess the impact of these two parameters on the average

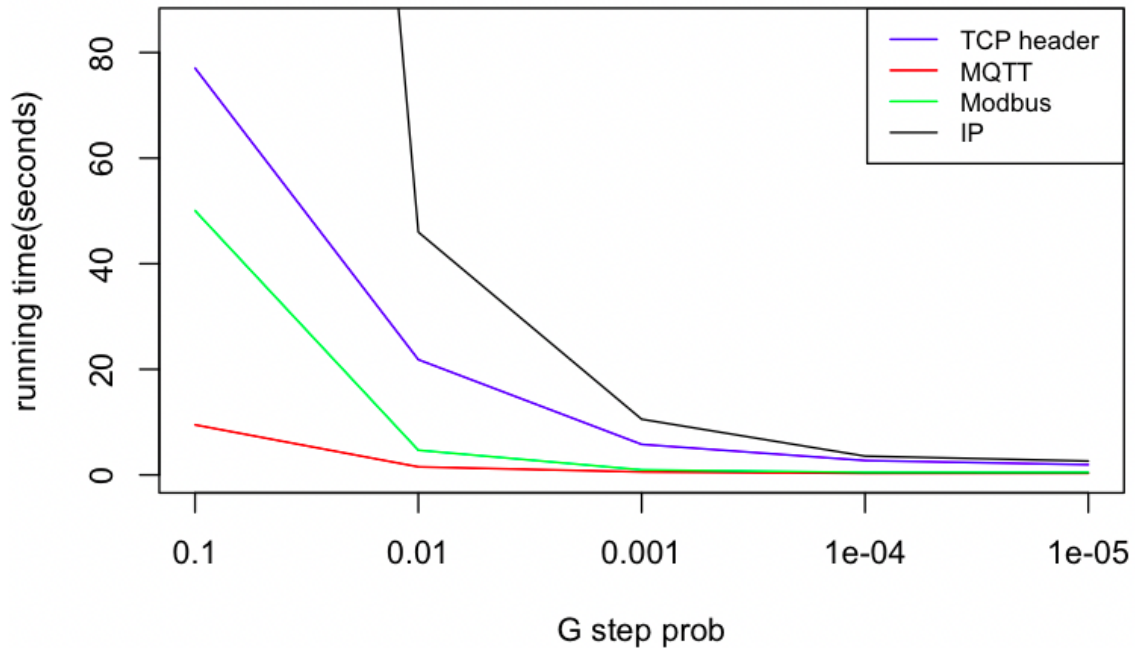


Figure 3.4: Running time vs. the probability for executing the G step(sample rate), the filtering rate = 0.9.

running time, we use 1000 packets for each protocol and set the filtering rate to 0.9 when we test the running time of *Auto-ETLV* with different G-step probabilities. In addition, we hold the G-step probability to $\frac{1}{1+packet\ index}$ when we test the running time for different filtering rates. To get the average running time, we run 10 experiments for each parameter configuration on a laptop computer (MacOS, 2.2GHz Dual-Core Intel Core i7, 8GB 1600MHz DDR3). The results are shown in Figure 3.4 and Figure 3.5. From the figures, we can see that *Auto-ETLV* runs very fast and returns a converged result in less than 2 minutes.

Note that the above two parameters have no impact on the correctness of the algorithm. They only serve as tuning knobs to speed up the algorithm. A natural question is: why should we need to use the G-step probability instead of just running the G step once? We made this decision for precaution. The inputs to the algorithm may contain some corrupted packets where the L field in the packets does not correctly measure the size of its value field. We call these packets *unexpected packets*. If we happen to run the G step on an unexpected packet, then the global dictionary will not contain the correct *key difference* information, and all verification will fail to update the correct *key difference*. To prevent this potential problem, we need to run the G

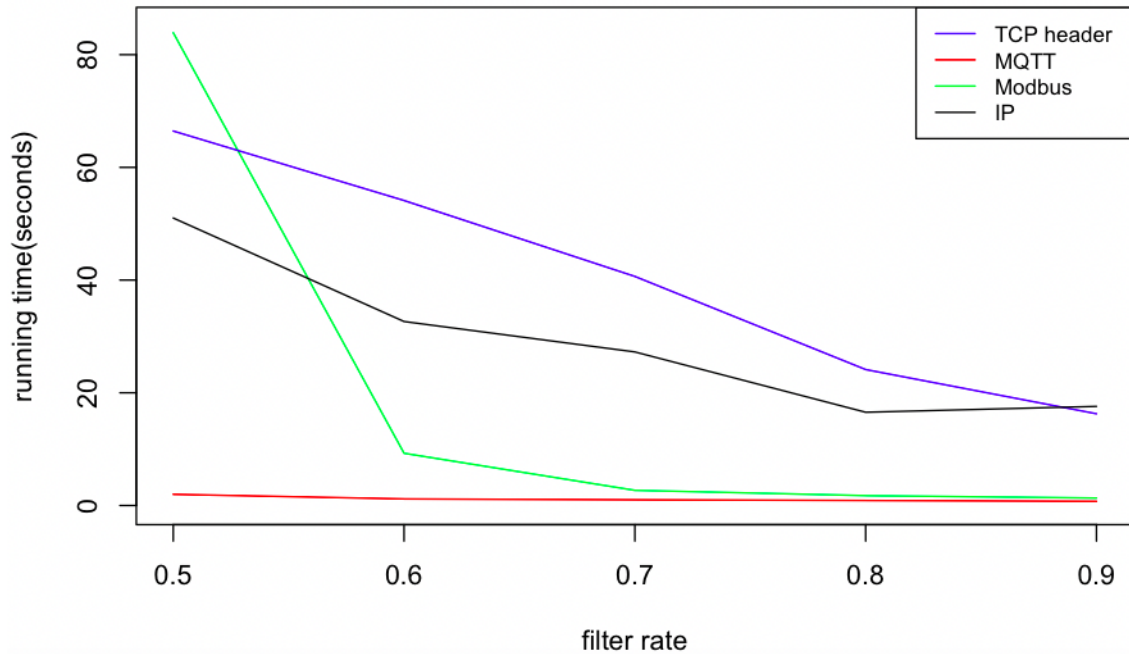


Figure 3.5: Running time vs. the filtering rate, the probability for executing G step $= \frac{1}{1+index}$.

step more than once, and we adopt a probability value that changes depending on the proportion of unexpected packets in the inputs. If the value is set to 1, there is no optimization (w.r.t. running speed), and every packet will use the G step. The result will be accurate but with a much longer running time. On the other hand, if the value is set to 0, then the G step will be executed only once, and the rest packets are processed with the V step. *Auto-ETLV* will run very fast, but the result might be incorrect¹.

Regarding the filtering rate, we can set this value to any value rather than 1. When the filtering rate is set to 1, we must drop any key if it is not updated in any packet. This might slow down the algorithm in case we drop a key that needs to be added back to the GD again at a later time. Like the G -step probability, the filtering rate depends on the proportion of unexpected packets in the inputs. When this value is set to 0, the size of GD will never be reduced, and in this case, the V step and the G step will have the same running time.

In summary, if the user can provide a good input where the proportion of unex-

¹When the packet processed by the G step is corrupted, *Auto-ETLV* will fail to return right answer.

pected packets is very small, they can choose a large filtering rate and a small G-step probability to greatly speed up the running time.

Chapter 4

Interactive ETLV Application

A good tool for protocol reverse engineering should support iterative analysis. That is, based on existing analytical results, people can make some reasonable guesses on the possible field boundaries, and the PRE tool should have a way to accept users' feedback and performs further analysis. After several iterations, we may find more accurate results.

Due to the above reason, we implement an interactive desktop application, as shown in Figure 4.1, for analyzing ETLV structure. The application supports the user interface to allow people easily to perform trial-and-error analysis on the ETLV structure.

4.1 Control Panel

Open & Run & Cancel: The Open button enables users to select a pcap file as input to the algorithm, where the pcap file records packets captured by traffic monitoring tools such as Wireshark. Once the file is selected, we can hit the Run button to start reverse engineering the packets using the Auto-ETLV algorithm. We can hit the Cancel button anytime to stop the current analysis, and open a new file or set parameters to different values before starting a new analysis.

layer: This parameter tells pcap file where to cut the packets before feeding into Auto-ETLV algorithm. This is an important input pre-processing step since the pcap file contains all information starting from data link layer. Having this additional lower layer information make the task of analyzing ETLV structure difficult. We need to select the layer that contains exactly the unknown protocol

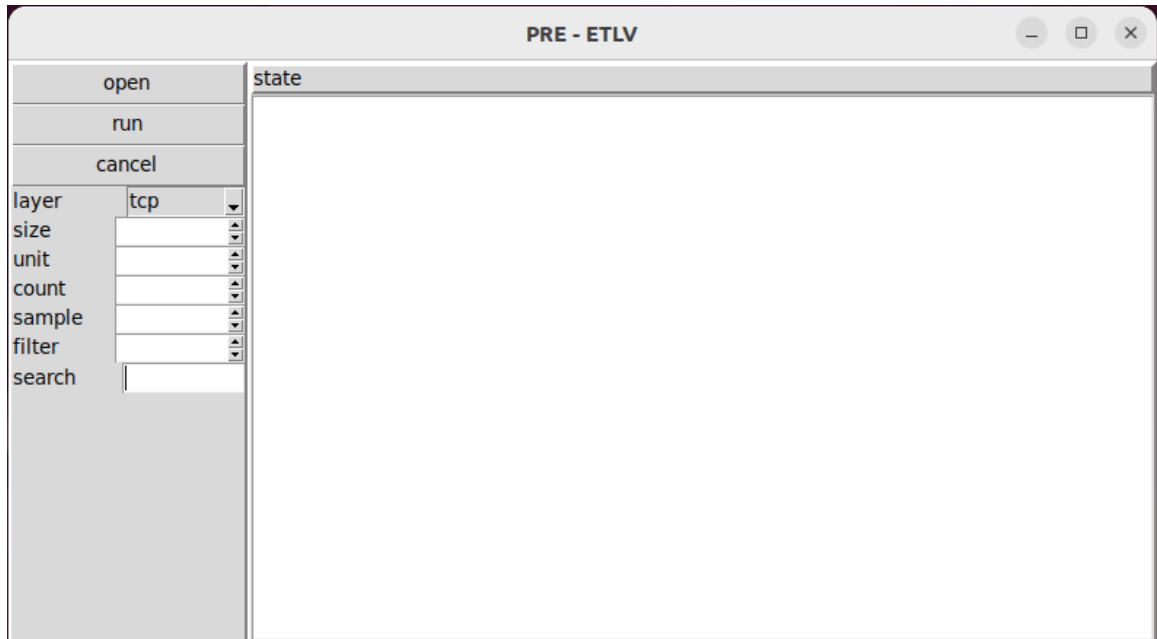


Figure 4.1: Interactive tool for ETLV analysis.

that we want to process. After this pre-processing step, our tool will cut the input packets and only leave the layer containing the unknown protocol that we want to analyze.

size & unit: These two parameters are useful if we have some knowledge about the value of the size and unit of the length field of the unknown protocol. We can set value to size or unit before running the algorithm, so that the application will only look at length field information for the specified size or unit. If we leave the unit or size values unset, the application will output all possible values for the unit or size.

Clearly setting these parameters can greatly speed up the analysis since we ignore all other possibilities. On the other hand, if we guess wrong values, the Auto-ETLV may just return empty result. As a suggestion, we can always run Auto-ETLV with all values unset, the output will tell us what possible values for unit and size. Based on this initial output results, we can make “educated” guess and only test the guessed values in the next-round analysis.

Count: This parameter is used to inform Auto-ETLV the number of packets it needs to process. This parameter is useful when we have a very large pcap file, e.g.,

a file containing several million packets. In many cases, we can find out the right ETLV structure without need to process all the packets. For instance, our results in Chapter 4 suggest that Auto-ETLV usually is able to find the ETLV results with about one thousand packet.

Sample & Filter: The Sample parameter controls the sample rate, i.e., the probability a packet is processed by G step. The Filter parameter controls the filter rate, i.e., the threshold beyond which we remove information from the global dictionary. As explained in Chapter 2, these two parameters do not affect the output of the algorithm if all input packets following the ETLV structure, but impact the performance dramatically. In addition, these two parameters provide robustness in the presence of bad inputs in which only a few packets follow the ETLV structure. How to set these values depends on how confident we believe the input packets following the ETLV structures. Also note that there is a trade off between robustness and speed performance. If we leave the values unset for these parameters, the application will use default values, i.e., the sample rate is set to $\frac{1}{1+index}$, where the meaning of *index* is explained in Chapter 2, and the filter rate is set to 0.9.

Search: This parameter is used to find whether or not some pattern exists in the output. For example, after processing the given number of packets, we can check whether or not certain unit-size pair exists in the output.

4.2 Display Panel

The right side window of Figure 4.1 displays the information about the output of the Auto-ETLV algorithm. We use the right side window of Figure 4.3 that shows a detailed running case to explain the meaning.

Running: indicates whether the programming is running.

layer: indicates which layer of protocol we are processing.

unit & size: indicates the unit and size that we are using. They will be empty string if we enumerate all possible unit and size pairs.

count: indicates the total number of packets we want to analyze.

max/min_seq_len: tells the length of longest/shortest packets.

max_cl: tells the upper bound of the check string we need to work with. This value bounds the unit plus size as proved by Lemma 2.

Output: is a list of records, where each record is in the form of

$$(bpos, size, unit) : [\text{range of length field}] \text{ unit, size } v \text{ off range}$$

where “bpos” is current bit position, which suggests that a possible length field ends at (bpos-1)-th bits. The “size” and “unit” are self-explanatory. The “range of length field” tells us where the length field starts and ends. “v” is Vcounter that counts how many times this record has been verified in Auto-ETLV. “off” is the key difference. The last “range” measures the range of the size of the V field that has been seen if the current information record was treated as the true length field.

4.3 An Example Case Study

This section shows how to reverse engineer MQTT packets using this interactive application. Let’s pretend that we do not know the structure of MQTT. We only know that it’s an application layer protocol. If we open the captured packets with Wireshark, we can find that the MQTT is running over the TCP protocol. Therefore we first set the “layer” as “tcp-app” , which means we only keep the application layer and remove all protocols below the application layer. Then we hit the open button, and a window like that in Figure 4.2 will be popped up. We then select the “mqtt.pcap” file as the input packets.

The next step is to set the count parameter. Because the pcap file contains over 300 thousand packets, it may take a long time to process all the packets, which may be unnecessary. We choose 100 packets here. We can also set other parameters if we have some knowledge about them. Nevertheless, we suggest not setting them for the first run.

Once we hit the run button, the program will start to process all the input packets. The output is displayed at the right window like Figure 4.3. We can see that there is only one output shown in the display panel. This output suggests that the length field ends at 16-th bit with unit $2^3 = 8$ bits (i.e. byte), and its size is 7 bits. $v = 100$

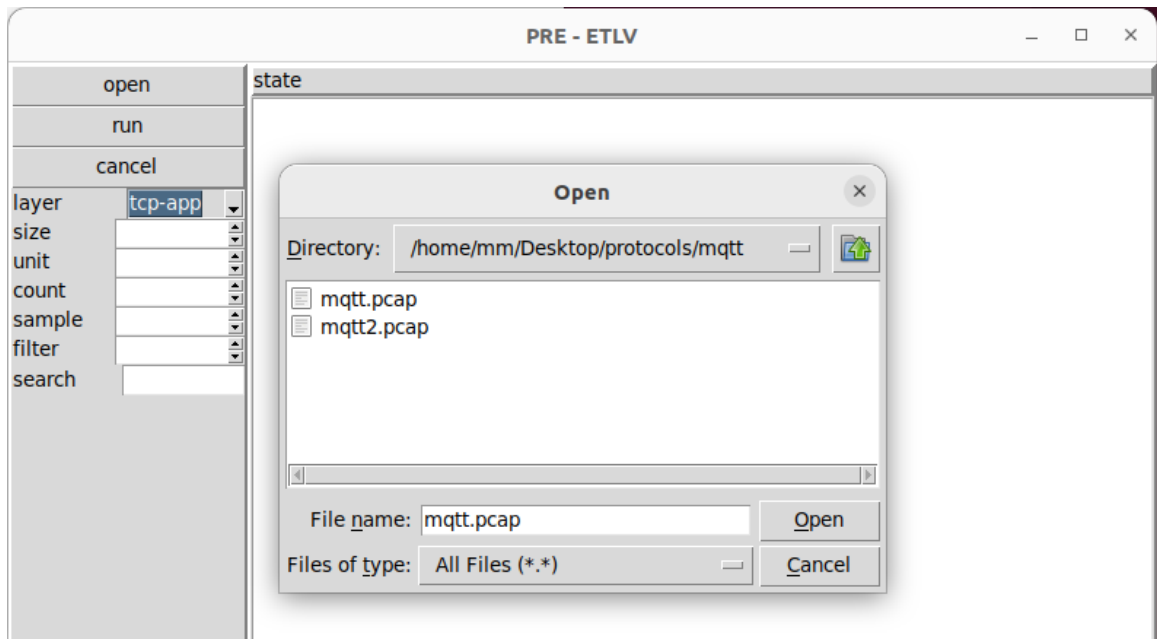


Figure 4.2: Interactive tool for ETLV analysis. Layer parameter is set to tcp-app. We click the open button and choose mqtt.pcap.

means that it passed all verification, so we are 100 percent sure that this is the length field. “off=0” means that the value field is 0 bits away from the length field. The above output matches exactly the MQTT specification.

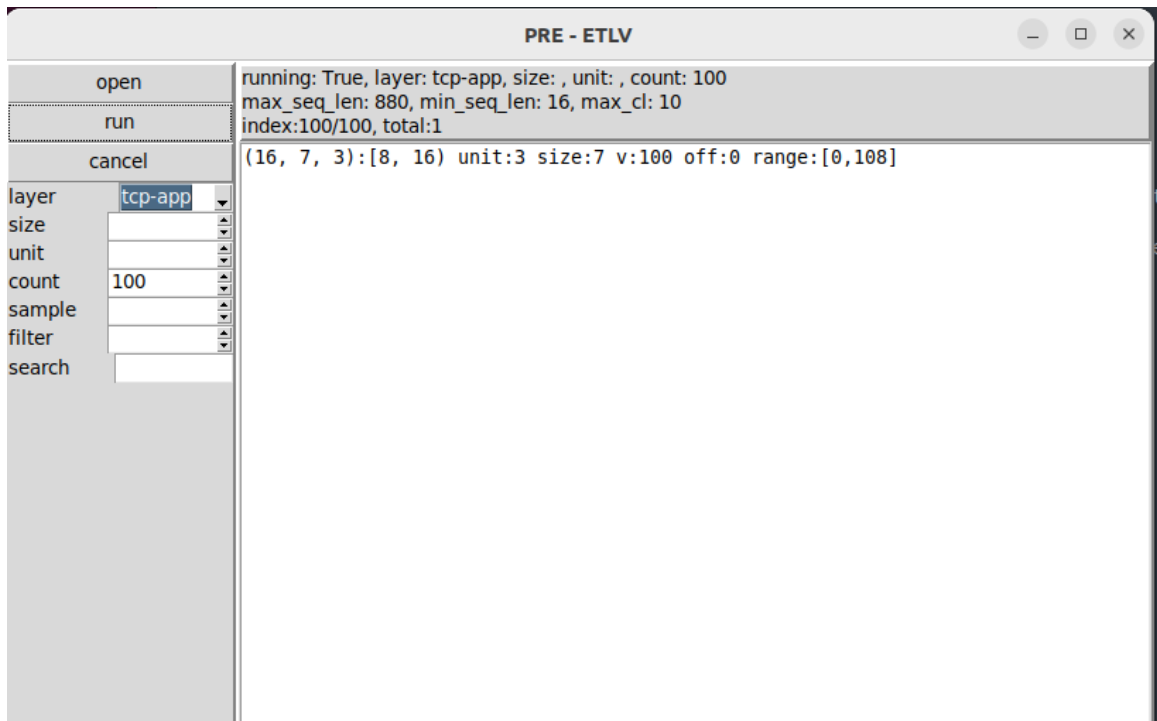


Figure 4.3: Output for ETLV analysis for MQTT.

Chapter 5

Discussion and Conclusion

5.1 Discussion

This section explains why ETLV is important and how it can be used in practice. We also discuss the limitation of our work.

As we have illustrated in Section 2.1, ETLV is a pretty general data format that covers the structure of many protocols. The concept of ETLV does not require the TLV fields to be consecutive in a data packet and assumes the default unit of length calculation may vary. Such generality is critical for protocol reverse engineering because we do not know the actual locations of the TLV fields or the unit that the protocol uses to calculate the length. With the identified LV fields, we can leverage this information for many practical applications.

One possible application is for security tests of unknown protocols. Network protocol fuzzing [15] is a technique that sends random data to the protocol port and checks whether or not the protocol responds. With some fields identified with our method, we may effectively generate packets that are likely to follow the legitimate format of the protocol, resulting in more targeted testing of the protocol. Another possible application is to enhance existing clustering-based PRE tools. PRE tools such as those in PI [3] cluster packets based on their similarity. With the help of our solution, the identified fields can be used as a significant feature to validate whether or not the clustering results are reasonable.

While the concept of ETLV and our algorithms are important and useful, they are only one of the building blocks in the arsenal for PRE. Our method alone cannot fulfill the complex task of PRE because real-world protocols usually involve much

rich information that ETLV may not model. In many cases, PRE requires expert domain knowledge and human intervention. Our future work will integrate the method presented in this paper into some open-source PRE tools.

5.2 Nested ETLV

Once we have identified the ETLV structure, the packet can then be partitioned into three blocks (i.e., the T, L, V blocks). It is possible that the V block also follows some ETLV structure. This is called nested ETLV structure, which may be found in some protocols. In this case, we will have the structure like ETLETLV. The next question is how we can identify the V field in ETLETLV?

We can extend our Auto-ETLV to identify a nested ETLV. We first apply Auto-ETLV and identify the leftmost ETLV structure. Once we find the length field, we break packets into TLV blocks, then apply Auto-ETLV on the V block. Auto-ETLV is able to find out if there is an ETLV inside. By repeatedly applying Auto-ETLV on the (new) V block, we can detect the nested ETLV structure.

5.3 Conclusion

In this thesis, we extended the concept of TLV to cover the packet structure of a large category of protocols. Based on the extended TLV (ETLV), we proposed an algorithm called Auto-ETLV to extract the length field from unknown binary protocols. Auto-ETLV not only identifies the left and right boundaries of the length field but also infers the unit that the length field uses. We test Auto-ETLV on several binary protocols, such as TCP, MQTT, and ModBus. The experimental results demonstrate that Auto-ETLV can quickly identify the length field for all the protocols under test. The returned answers are correct in the sense that the answers not only include the ground truth but also cover all possible protocol specifications consistent with the parsed packets.

In the future, we plan to apply Auto-ETLV in protocol fuzzing to detect potential security problems in private protocols.

Bibliography

- [1] Joao Antunes, Nuno Neves, and Paulo Verissimo. Reverse engineering of protocols from network traces. In *2011 18th Working Conference on Reverse Engineering*, pages 169–178. IEEE, 2011.
- [2] Vahida Z Attar and Piyush Chandwadkar. Network discovery protocol lldp and lldp-med. *International Journal of Computer Applications*, 1(9):93–97, 2010.
- [3] Marshall A Beddoe. Network protocol analysis using bioinformatics algorithms. *Toorcon*, 2004.
- [4] Carsten Bormann, Angelo P Castellani, and Zach Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.
- [5] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329, 2007.
- [6] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. {MACE}:{Model-inference-Assisted} concolic exploration for protocol and vulnerability discovery. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
- [7] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14, 2007.
- [8] Joseph Davies. Radius protocol security and best practices. *Microsoft Corporation*, 2002.

- [9] Joshua Davies. *Implementing SSL/TLS using cryptography and PKI*. John Wiley and Sons, 2011.
- [10] Julien Duchene, Colas Le Guernic, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. State of the art of network protocol reverse engineering tools. *Journal of Computer Virology and Hacking Techniques*, 14(1):53–68, 2018.
- [11] Jack Halon. Reverse engineering network protocols.
- [12] Aleksandar Kuktin. Binary protocol design: Tlv, ltv, or else? online discussion in 2014, accessed in Sept. 2021.
- [13] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, volume 8, pages 1–15. Citeseer, 2008.
- [14] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.
- [15] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a grey-box fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.
- [16] Stanislav Shalunov. ‘internet netflow statistics – internet2 netflow organization’. accessed July 2021.
- [17] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.
- [18] Dipa Soni and Ashwin Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*, volume 20, 2017.
- [19] Siyu Tao, Hongyi Yu, and Qing Li. Bit-oriented format extraction approach for automatic binary protocol reverse engineering. *Iet Communications*, 10(6):709–716, 2016.