Inspector: A Semantics-Driven Approach to Automatic Protocol Reverse Engineering

Yige Chen[1,2], Tianning Zang[1,2], Yongzheng Zhang[1,2], Yuan Zhou[3], Peng Yang[3(✉)], and Yipeng Wang[4]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] National Computer Network Emergency Response Technical Team/Coordination Center of China, Beijing, China
yp@cert.org.cn
[4] Beijing University of Technology, Beijing, China

Abstract. The automatic protocol reverse engineering for undocumented network protocols is important for many fundamental network security applications, such as intrusion prevention and detection. With the growing prevalence of binary protocols in the network communication to organize data in a terse format and ensure data integrity, the proven reverse approaches for ordinary text protocols face severe challenges in the compatibility. In this paper, we propose Inspector, an automatic protocol reverse engineering approach that exploits semantic fields to infer message formats from binary network traces. Inspector reasonably infers two semantic fields based on the binary content analysis of protocol messages to support clustering messages and message format inference. We evaluate the effectiveness of Inspector on two binary cryptographic protocols (TLS and SSH) and a binary unencrypted protocol MQTT by measuring the accuracy of message clustering and comparing the inferred message formats with the ground truths on a traffic dataset captured from a campus. Our experimental results show that Inspector accurately cluster messages with 100% cluster precision and 100% message recall for TLS, 90% cluster precision and 99.6% message recall for SSH, 100% cluster precision and 92.7% message recall for MQTT. Based on the accurate message clusters, Inspector can correctly infer the format of the messages in the cluster.

Keywords: Protocol reverse engineering · Protocol format inference · Binary protocol · Semantic fields

1 Introduction

This paper concerns protocol reverse engineering for binary protocols based on network traces, which is to automatically infer the application-level specification

H. Gao and X. Wang (Eds.): CollaborateCom 2021, LNICST 406, pp. 348–367, 2021.
https://doi.org/10.1007/978-3-030-92635-9_21

of the binary protocol by only accessing the passively collected network traces. The protocol specification stipulates all formats of the legal messages sent by the communicating parties. Consequently, the protocol specification is critical in many network security applications, such as intrusion prevention and detection [21] and unknown application protocol parsing [23]. Taking the botnet as an example [14], when we know the specifications of the malware communication protocol, we can easily recognize the malicious traffic and intercept these traffic to protect vulnerable hosts from being compromised. With the growing attention to network security, the mainstream applications gradually adopt binary cryptographic protocols to ensure data security and ensure data integrity in the network transmission [6]. The Internet of Thing (IoT) devices need a lightweight messaging protocol to minimize bandwidth and power usage in the network communication. Unlike traditional text protocols, such as HTTP, SMTP, and FTP, it is much more difficult for the network administrator to analyze the network traces of binary protocols because the binary byte sequences are designed to be understood by a machine but not a human being.

The classic network-trace-based approaches generally follow the routine of first clustering messages and then inferring the protocol specification based on the clustering results [3,9,23]. Because of the cascade inference routine, the accuracy of the inference results heavily relies on the precision and coverage of the message clustering results. These approaches are incompatible with binary protocol traffic because the binary message structure greatly increases the difficulty of aligning variable position fields and adversely affects the accuracy of message clustering. Thus, a feasible idea to overcome the obstacles of binary protocol reverse engineering is to perceive the binary message structure before conducting message clustering and protocol specification inference.

In this paper, we propose a semantics-driven approach named Inspector to conduct automatic protocol specification inference from network traces. Taking full advantage of the general design methodology of the binary protocol, we reasonably infer two semantic fields, namely the length field and message type field, before conducting the messages clustering and the subsequent protocol specification inference. Based on the inference of the semantic fields, we can accurately cluster messages of the same message format and then directly infer the specification of message formats. Similar to the Discoverer [9], the message format output by the Inspector is a sequence of specification tokens that specify both the semantics and properties of the message fields. To evaluate the effectiveness and accuracy of the Inspector, we collect a dataset of two widely used binary cryptographic protocols, namely Transport Layer Security (TLS) [11] and Secure Shell (SSH) [25], and a lightweight messaging protocol Message Queuing Telemetry Transport (MQTT) [22]. The ground truth specifications of these protocols are public in the Request for Comments (RFC) [12] and OASIS standards [17]. We deploy the Inspector on the collected dataset to infer the specifications without knowing the specification documents in advance. The experimental results show that the Inspector can properly deal with the semantic field inference and message clustering, and then perform accurate protocol specification inference.

We briefly summarize our major contributions as follows:

- We propose a semantic-driven approach named Inspector to implement the specification inference for binary protocols. The Inspector recursively extracts two kinds of semantic fields of the message and then cluster messages based on the semantic fields until each cluster contains messages of the same format.
- We design a hierarchical length field model and a corresponding algorithm to search length field pattern candidates and select the correct one for the messages. We also design a message type field inference algorithm to search for message type fields to support the clustering of messages with the same length field pattern.
- We implement the Inspector and conduct evaluation experiments over three binary protocols. The experimental results show that the Inspector can accurately identify the semantic fields of the messages and partition messages into clusters with 100% cluster precision and 100% message recall for TLS, 90% cluster precision and 99.6% message recall for SSH, 100% cluster precision and 92.7% message recall for MQTT. The Inspector also works well in the format inference and outputs results that are consistent with the ground truths.

The rest of this paper is organized as follows. Section 2 summarizes the related work of the protocol reverse engineering. Section 3 introduces the inference of the length fields and message type fields and then describes the details of the message clustering and format inference. Section 4 introduces the experimental dataset and presents the evaluation details of the Inspector. Section 5 concludes this paper.

2 Related Work

A communication protocol can be regarded as an agreement on how to properly exchange information between communication entities [15]. The specifications of communication protocols are similar to natural languages, including vocabulary and grammar. The vocabulary consists of a set of words or byte values that can compose valid messages. To clearly communicate with each other, the communication entities need to negotiate the semantics of the words or the meaning of the byte values in advance. The grammar specifies the detailed rules of the information exchange procedure. In a communication process, the sender should follow the procedural rules to enclose the raw information into a formatted message. When the receiver observes the formatted message, he can restore the information from the message through inverse operations. The protocol reverse engineering focuses on both the vocabulary and grammar of the protocol specification.

Existing automatic protocol reverse methods can be divided into two families: methods based on execution traces [4,5,8,10] and methods based on network traces [1,3,9,23,27]. The methods based on execution traces include Polyglot [5], Tupni [10], Prospex [8], and Dispatcher [4]. These methods conduct protocol reverse engineering by directly debugging the protocol binary or monitoring how the protocol binary generates and parses messages. Because these methods can

dissect the real binary program, the accuracy of the inference results is generally high. However, we must obtain the corresponding protocol binary programs in advance to obtain the execution traces. Besides, some private protocol developers may deliberately use obfuscation techniques to thwart protocol specification inferences such as binary packing, inlining unnecessary instructions, and replacing calls with indirect jumps [16].

In contrast, the methods based on network traces only take the network traces of the target protocol as input and the obtaining of network traces is more straightforward. When the protocol binary programs are available, we can control the binary programs to automatically generate the target traces and then directly capture the traces. When the binary programs are unavailable, we can collect the network traces by passively monitoring the network gateway through port mirroring switches [26] and then sift out the target traces based on the IP address or port number.

Existing methods based on network traces include Discoverer [9], ProDecoder [23], Netzob [3], ProWord [27], and FieldHunter [1]. The basic inference framework of these methods is first clustering messages based on their group characteristics and then performing specification inferences within each cluster. Cui *et al.* first propose Discoverer [9] to automatically infer the complete protocol format from network traces. Discoverer first tokenizes messages into a sequence of text and binary tokens and then conducts initial clustering messages based on the token pattern. After the initial clustering, Discoverer recursively identifies the format distinguisher (FD) tokens in the messages through a heuristic method and divides existing clusters into subclusters based on the values of FD tokens until each cluster only contains messages of one format. Finally, Discoverer directly infers the protocol specification by making comparisons between the token sequences within each cluster. Wang *et al.* propose ProDecoder [23] to support the specification inference of asynchronous protocols. ProDecoder first uses n-gram sequences to represents messages and exploits Latent Dirichlet Allocation (LDA) [2] to select the keywords in the messages. Then, ProDecoder uses Information Bottleneck (IB) [20] as the clustering metric to partition messages into clusters based on their keywords. Based on the clustering results, ProDecoder can find the invariant fields among the messages of each cluster and output the message formats in the form of regular expressions. To provide more accurate inference results, Bossert *et al.* propose Netzob [3] to pre-cluster messages with the same program activity to support the subsequent clustering. Then, Netzob introduces contextual and semantic information as a key parameter to cluster messages and partition the fields in the message. Based on the intermediate inference results, Netzob uses an extended sequence alignment algorithm to align the partitioned fields in different messages to extract the message format. Unlike previous works, the ProWord [27] proposed by Zhang *et al.* formulates the protocol reverse engineering as an information retrieval problem and focuses on extracting protocol feature words from network traces. They first modify the voting expert algorithm [7] to extract feature word candidates and then select the desired feature words based on the frequency, location, and length of the word

candidates. The recent format inference work FieldHunter [1] focus on identifying field boundaries and inferring specific field types for both binary and textual protocols. The FieldHunter successfully infers several semantic fields for binary and textual protocols. However, the FieldHunter is hard to precisely infer the length fields for binary protocols since the message data structure is unknown.

3 System Design

In this section, we first present an overview of the Inspector. Then, we describe the phases of the Inspector in detail, including length field inference, message type field inference, and format inference.

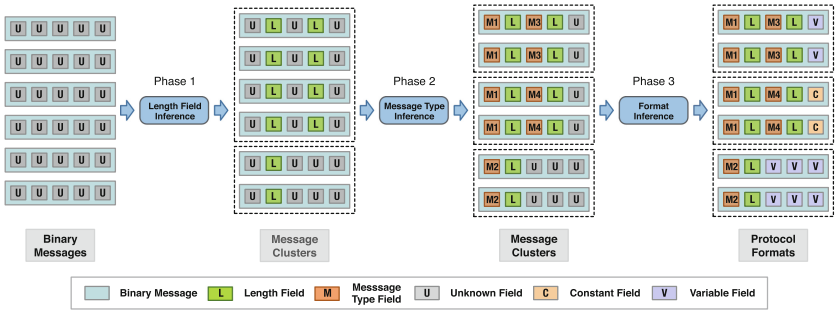

Fig. 1. System architecture of the inspector

3.1 Overview

The system architecture of the Inspector is shown in Fig. 1. The first phase of the Inspector is length field inference, which takes the binary messages as input and searches for the length field pattern candidates based on the proposed hierarchical length field model. After getting all the length field pattern candidates that meet the model requirements, we selected the correct length field pattern based on a reasonable assumption that the correct length field pattern should be more frequently appeared than the wrong ones in the messages and can be well explained. The length field pattern contains both the position information of the length fields and the hierarchical information of the message, so we can cluster messages with the same length field pattern.

The second phase is message type field inference that intends to find the message type fields of the message. Since we have discovered the hierarchy of the messages, we can perform targeted lookups for the message type fields around the diverging positions of the message hierarchy. The main intuition in identifying the message type field is that the number of unique message types should be limited to an appropriate range, and the sizes of the same message type message are relatively close. After inferring the message length field, we can perform finer message clustering so that each cluster only contains messages of one message format.

The third phase format inference focuses on the token properties inference. The output of the message format inference is a sequence of format tokens, including length field, message type field, constant field, and variable field. Since the semantic-driven approach can provide precise and pure clustering results, we can directly infer the token property sequence of the message cluster through a simple and effective sequence alignment method.

3.2 Length Field Inference

The network traces between communication parties can be viewed as consecutive dialogue messages. However, dialogue messages should be divided into independent data packets to meet the transmission protocol of the Internet. Therefore, after collecting network traces from the network link, we first need to recover the original dialogue messages in the same way as the communication parties. For the packets transmitted by Transmission Control Protocol (TCP), we reassemble them into the original message to ensure the integrity [19] and take it as system input. The datagram transmitted by User Datagram Protocol (UDP) is naturally integrated [18], so we directly extract the payloads and regard them as individual messages.

Algorithm 1. Length Field Inference Algorithm

Input: Message Dataset $\boldsymbol{MD}$; Bit-Widths $\boldsymbol{BW}$; Orientations $\boldsymbol{O}$; Endianness $\boldsymbol{E}$; Size of the Suffix Field $\boldsymbol{S}$; Max Field Number $\boldsymbol{NF}$; Selection Threshold $\boldsymbol{ST}$; Minimum Number of the Messages in the Cluster $\boldsymbol{NC}$.
Output: Message Clusters $\boldsymbol{MCS}$.
Message $\mathbf{M}(mb, mps, c, p)$: mb = message bytes, mps = message parts to be inferred, c = length field pattern candidates, p = selected length field pattern.
LengthFieldPattern $\mathbf{LFP}(ofs,\ w,\ ori,\ e,\ sub)$: ofs = field offset, w = bit-width, ori = orientation, e = endian, sub = sub length fields.
function CandidateSearch($mp, n=1$)
    $C = \varnothing$
    if $n > \boldsymbol{NF}$ **then**
        return C
    end if
    for ofs from 0 to Len(mp) **do**
        for all (w, ori, e) in $(\boldsymbol{BW} \times \boldsymbol{O} \times \boldsymbol{E})$ **do**
            $entity \leftarrow$ extract the field bytes by ofs, w
            $value \leftarrow$ parse the value of $entity$ by ori, e
            $leftlen \leftarrow$ Len(mp) $- value - w - ofs$
            $lengthfield \leftarrow \mathbf{LFP}(ofs, w, ori, e, \varnothing)$
            if $n = 1$ **and** $leftlen$ in $\boldsymbol{S}$ **then**
                add [$lengthfield$, ('suffix', $left$)] to C
            end if
            if $left = 0$ **then**
                add [$lengthfield$] to C

```
            else if left > 0 then
                extpart ← Cut out the extra parts
                lf ← CandidateSearch(extpart, n + 1)
                merge lengthfield + lf to C
            end if
        end for
    end for
    return C
end function
function CandidateSelect(mc, ST, NC)
    while mc contains message m with nonempty m.c do
        C ← extract all m.c for all m in mc
        lfps ← select the element that exists in most messages from C
        num ← count the messages containing lfps
        prop ← num/LEN(mc)
        if prop ≥ ST and num ≥ NC then
            for all messages m where m.c contains lfps do
                m.lf ← lfps or add lfps into related sub length field
                m.c ← ∅
                update new m.mps based on the new m.lf
            end for
        end if
        remove lfps from all m.c
    end while
end function
procedure LengthFieldInfer(MD)
    initmc = ∅
    for all message in MD do
        add M(message, [message], ∅, ∅) to initmc
    end for
    MCS = [initmc]
    while MCS contain m with nonempty m.mps do
        for all mc in MCS do
            for all m containing nonempty m.mps in mc do
                m.c ← CandidateSearch(m.mp)
            end for
            CandidateSelect(mc)
            Split mc into subclusters based on the length field pattern.
        end for
    end while
    return MCS
end procedure
```

The primary purpose of the length fields in the message is to specify the boundary of the variable-length payload, so the communication parties can accurately extract the relevant payload from sequential binary messages without

explicit delimiters. Thus, when we find the length fields in the message, we can extract the payload just like the communication parties and then perform further format analysis for each message part. There are two main difficulties in seeking the length fields. a) The first difficulty is to deal with the hierarchy of the length fields. From a vertical perspective, only the length fields in the top-level can be directly related to the size of the message, while other length fields are only related to the size of the relevant payload. From a horizontal perspective, there could be multiple parallel length fields in the message, which weakens the correlation between the value of the length field and the size of the message. When we perform length field inference for a message, the only reliable length information is the total size of the message. Without a prior definition of the length field structure, we cannot infer most of the length fields. b) The other difficulty is to deal with the various interpretations of the length field. The length fields in different protocols may have their preferred specifications, such as in terms of position, endianness, and bit-width. To be specific, the length field could be located before or within the relevant payload, the endianness could be little-endian or big-endian, and the bit-width could be 8, 16, 32, or 64 bits. Even in one binary message, the protocol designers may use different bit-width length fields to compress the message size. An intuitive method to deal with the undetermined interpretation of the length field is to conduct a grid search that requires costly computing resources. Therefore, we need a search strategy that can hit the correct interpretation of the length field but only consumes moderate computing resources.

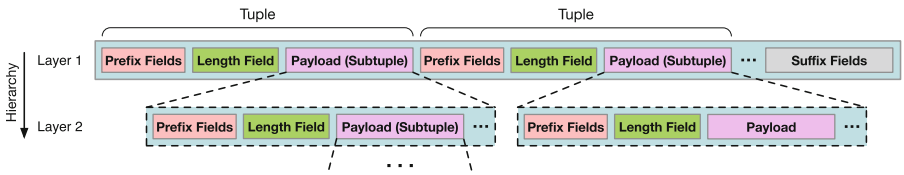

Fig. 2. Hierarchical length field model

In order to solve these two problems, we propose a hierarchical length field model to perform the length field search in a certain routine and develop a corresponding algorithm to implement the length field inference. Figure 2 shows the structure of the hierarchical length field model. In this model, the basic structural units of the message are tuples that arranged in sequence or nested recursively. An ordinary tuple contains prefix fields, a length field, and a payload. The prefix fields typically provide meta information of the message, such as message type, protocol version, and session ID. The value of the length field determines the size of the subsequent payload. The contents of the payload could be sequential bytes or a subtuple that should be parsed in the lower layer. The suffix fields are optional with respect to the protocol format, which may store the message authentication code of the preceding byte sequences. Based on the hierarchical

length field model, we develop an algorithm for the search and selection of length field pattern candidates. During the length field inference process, the algorithm is executed sequentially from the top layer to the bottom layer because the inference operation in the lower layer depends on the determination of the payload in the upper layer.

Algorithm 1 shows the details of the length field inference algorithm, which contains a procedure LengthFieldInfer and two functions CandidateSearch and CandidateSelect. We first define the Message and LengthFieldPattern of the inference algorithm. A Message object contains message bytes mb, the message parts to be inferred mps, the length field pattern candidates c, and the selected length field pattern p. A LengthFieldPattern object contains the field offset ofs, bit-width w, orientation ori, endian e, and the length field pattern in the lower layers sub (lines 3–4). The procedure LengthFieldInfer first initializes the message dataset MD to the initial message clusters MCS for the management of the intermediate clustering results. Since the initial Messages have no length field pattern, we consider these messages sharing the same empty length field pattern (lines 47–51). In the process of the hierarchical length field inference, each Message cluster should only contain all messages of the current length field pattern. Then, for all message parts in the message cluster mc, we call the function CandidateSearch to search the length field pattern candidates fitting the hierarchical length field model in the current layer (lines 52–56). The function CandidateSelect intends to pick out the correct length field from numerous candidates based on the relevance between the length field pattern candidates and the messages in a cluster, so we only call it after performing the function CandidateSearch for all Messages in the cluster (line 57). Since the CandidateSelect may produce new length field patterns and the new message parts in the next layer, we should split the current cluster into subclusters to maintain the format purity of each cluster and then iteratively execute the process above until no more new message parts are generated (line 58).

The function CandidateSearch makes use of the characteristics of the length field to perform the search task in an effective strategy to significantly reduce the search space of the length field. Specifically, we clarify the algorithm parameters in advance, including the bit-width of the length field $\boldsymbol{BW}$, the length field orientation $\boldsymbol{O}$, the endianness $\boldsymbol{E}$, the size of the suffix field $\boldsymbol{S}$, and the maximum field number $\boldsymbol{NF}$. We first limit the maximum number of length fields in each layer to $\boldsymbol{NF}$ (lines 7–9). Then, we slide the offset ofs on the message part to determine the length field position and traverse the values of the w, ori, e in the Cartesian product $\boldsymbol{BW} \times \boldsymbol{O} \times \boldsymbol{E}$ (line 10–11). Based on the above parameters, we can extract the potential length field entity and parse the value of the selected field entity (lines 12–13). If the value of the selected field exactly matches the size of the left part (with or without a suffix field), we consider this field a candidate (lines 16–20). When the value of the field is smaller than the size of the left message part, we iteratively call the function CandidateSearch to analyze the extra message part and splice the current message field with the recursive results (lines 21–25). The output of the CandidateSearch function is a set of length field

pattern candidates that can be applied to the input message part, so most of them are potentially illegal to other message parts and must be filtered out.

The function CandidateSelect in the algorithm processes the numerous length field pattern candidates and selects the correct one from them. The basic principle for selecting the length field pattern is that the correct one is more robust to the messages of the cluster. For the input Message cluster mc, we first extract all length field candidates and select the candidate $lfps$ that exists in most messages (line 32–33). Then, we count the messages containing $lfps$ in the cluster mc and calculate the proportion of the covered messages to the cluster mc (line 34–35). The number of messages in the cluster num should be greater than or equal to the preset minimum number of the message in the cluster $\boldsymbol{NC}$ and the proportion of the covered messages $prop$ should be greater than the preset selection threshold $\boldsymbol{ST}$ (line 36). We may find several different length field pattern candidates meet the selection but cover the same messages because their parsed values of the length fields are all the same, so we simply choose the one with the largest bit-width to maximize robustness. After selecting the length field pattern, we apply the pattern to all covered messages and exclude these messages from subsequent selections (lines 37–39). We also extract new message parts in the next layer since we have obtained the length field pattern in the current layer (line 40). Whether a length field pattern candidate passes the selection or not, we should remove the candidate from all messages before the next selection. Otherwise, these selected candidates would be repeatedly counted and influence the subsequent selections (line 43). Through the collaboration of the procedure LengthFieldInfer and the functions CandidateSearch and CandidateSelect, we can infer the length fields from collected traffic trace without knowing the protocol specification in advance.

3.3 Message Type Field Inference

Similar to the length field, the message type field is also a semantic field that plays a key role in the protocol format inference. The message type field usually indicates the specific format type of the message when exists, so that the communicating parties can reach a consensus on the structure of the relevant payload. Therefore, when we find the message type field in the message, we can further cluster the messages with the same format type. Figure 3 shows an example of the message type fields in the TLS messages. According to the RFC document [11], the byte field at offset 05 is a top-level message type field, which is marked with a light blue background in the figure. In this example, the message type field uses continuous values from 14 to 22 (Hex) while other fields take fixed or random values. This phenomenon only appears in the cluster where the observed messages belong to the same superior format and the message type field is used to distinguish different inferior formats. The message type field typically locates around the layer break position of the message hierarchy since the communicating parties need to know the specification before parsing the transmitted payload. Fortunately, we can exploit the hierarchy of the length field pattern to perform preliminary clustering and make the messages in each cluster shares the

Offset	00	01	02	03	04	05	06	07	08	09	10	11	···
Msg 1	00	00	05	34	05	**14**	05	7D	FF	59	13	54	···
Msg 2	00	00	00	0C	0A	**15**	00	00	00	00	00	00	···
Msg 3	00	00	00	2C	06	**1E**	00	00	00	20	D7	48	···
Msg 4	00	00	01	04	09	**1F**	00	00	00	68	00	00	···
Msg 5	00	00	01	0C	06	**20**	00	00	01	00	78	1F	···
Msg 6	00	00	03	3C	08	**21**	00	00	01	17	00	00	···
Msg 7	00	00	00	14	06	**22**	00	00	04	00	00	00	···

Fig. 3. An example of the message type field in the TLS messages

same length field pattern. We limit the search range of message type field to the unknown fields in each layer to avoid the meaningless searches.

Based on two reasonable intuitions that the message type field typically takes a few distinct values to distinguish different formats and the length of messages in one format tends to be similar, we use the following four indicators to infer the message type field.

- The number of used values *num*, which measures the unique number of the value appeared in the collected dataset. We estimate this indicator of each unknown field by simply counting the number of unique values.
- The mean of message size variances *var*, which measures the average message size uniformity of the assumed message type field.

We propose a message type field inference algorithm using these two indicators, as shown in Algorithm 2. The main idea of the algorithm is to limit the values of the first indicator to a predefined value range to make preliminary filtering, and then use the second indicator to select the message type field. Specifically, the procedure MessageTypeInfer first traverses the byte offset *ofs* from the prefix field start position ***Start*** to the prefix field end position ***End*** to locate the field of the messages in the given message cluster ***MC*** (line 7). For the field at the byte offset *ofs*, we calculate the number of used values *num* and the mean of message size variances *var* for all the messages in the cluster (lines 8–18). We sift out the fields that meet the requirement of the indicator *num* and consider the one with smallest *var* as the message type field (lines 19–22). Once the message type field in the message cluster is determined, we can perform a more elaborate message clustering so that each cluster contains all messages of one format.

Algorithm 2. Message Type Field Inference Algorithm

Input: Message Cluster $\boldsymbol{MC}$; Prefix Field Start Position $\boldsymbol{Start}$; Prefix Field End Position $\boldsymbol{End}$; Valid Number of Message Types $\boldsymbol{NT}$.
Output: Message Type Field $\boldsymbol{Msgtype}$.
MessageType MT(ofs, num, var) : ofs = offset, num = the number of unique values, var = the average of message length variances.
procedure MessageTypeInfer($\boldsymbol{MC}$)
    $Msgtype$ = None
    $minvar = +\infty$
    for ofs from $\boldsymbol{Start}$ to $\boldsymbol{End}$ **do**
        $fieldbytes \leftarrow [m[ofs]$ for m in $\boldsymbol{MC}]$
        $byteset \leftarrow$ Set($fieldbytes$)
        $num \leftarrow$ Num($byteset$)
        $varsum = 0$
        for $byte$ in $byteset$ **do**
            $lens \leftarrow$ the sizes of messages m, where $m[ofs] = byte$
            $varsum$ += Var($lens$)
        end for
        $var \leftarrow varsum/num$
        if $num \in NT$ **and** $var < minvar$ **then**
            $Msgtype \leftarrow \mathrm{MT}(ofs, num, var)$
            $minvar \leftarrow var$
        end if
    end for
    return $Msgtype$
end procedure

3.4 Protocol Format Inference

After obtaining the length fields and the message type fields, we can exploit these fields to cluster messages with the same format and then perform protocol format inference for other fields. Similar to Discoverer, we specify the inferred format as a sequence of token specifications and use token properties to describe field specifications [9]. The token properties describe two field characteristics: binary/text and constant/variable. The first token property focus on the encoding of the field. Although we focus on the format inference of the binary messages, we may still encounter text segments in the messages. A binary field can be parsed through hexadecimal conversion, while a text field is parsed through character decoding. The second token property concerns whether a field is fixed to a certain value in different messages. The inference of this property depends largely on the adequacy of the datasets. When the data source is deficient, we can only observe a few values of a variable token and may misjudge it as a constant field. To avoid this kind of mistake, we need to collect network traces from different

sources to enrich the value diversity of variable tokens and improve the accuracy of the token property.

The detailed process of the token attribute inference is as follows. For each unknown token in the message, we collect all the values that have appeared in the cluster's messages. When the token takes the same value across in all messages, we consider it as a binary constant token. When multiple consecutive tokens take variable values within the ASCII coding range in all messages, we consider them as text variable tokens. All tokens outside the above cases will be considered as binary variable tokens.

4 Evaluation

In this section, we first introduce the dataset used in the experiments. Then, we define the evaluation metrics in detail and set the tunable parameters of two inference algorithms. Finally, we present the experimental results and the corresponding analyses.

4.1 Datasets

We evaluate the Inspector on three widely used binary protocols, namely TLS, SSH, and MQTT. The Transport Layer Security (TLS) [11] is a binary application protocol that primarily aims to provide privacy and data integrity between network communicating parties. The Secure Shell (SSH) [25] is a binary application protocol that provides a secure communicating channel for the client and server. Therefore, except for the channel establishment traffic and control fields in the messages, the real payload transmitted by these two protocols is strictly encrypted. The Message Queuing Telemetry Transport (MQTT) is a lightweight unencrypted message protocol that designed for the IoT communications. Since we aim to reverse the format specification of these protocols, we can focus on the message format structure and ignore the content inside the encrypted payload. The ground truth specification details of these three protocols are available in the online RFC documents [12] and OASIS standards [17].

We collect a bunch of protocol traffic traces from a campus network gateway. The summary of the dataset is shown in Table 1. During the trace collection, we try to collect the traffic from different client and server pairs to enrich the diversity of the protocol formats. Some particular messages of the binary protocols may be text messages. For example, the version announcement message of the SSH is a pure text message. Since many previous works have studied format reverse engineering for the text-oriented protocols, we filter out the text messages for individual processes. A simple and effective method to separate the text messages from binary messages is to check the existence of consecutive ASCII printable characters. When a message begins with these kind of characters, we treat this message as a text message.

4.2 Evaluation Metrics

The evaluation metrics are designed to measure the effectiveness of the message clustering and the consistency between the inferred formats and the ground truths. The previous works have proposed many evaluation metrics to evaluate clustering accuracy. Some metrics have been frequently used, such as correctness [3,9,27] and conciseness [3,9], while others are specific to their experiments [23,24]. In our evaluation experiments, we combine the correctness and conciseness to a comprehensive metric named format precision and adopt the message coverage in Discoverer [9] as the message recall.

Table 1. Summary of datasets used in the evaluation

Protocol	Source	Size(B)	# Packets	# Messages	# Formats
TLS	Campus	27,601K	324250	44316	12
SSH	Campus	5,177K	23900	12988	10
MQTT	Campus	3,791K	33164	33879	21

Table 2. Summary of tunable parameters

#	Parameter	Var	Value
1–1	Bit-Widths (Bits)	BW	8/16/32
1–2	Orientation	O	Before/Within
1–3	Endianness	E	Little/big
1–4	Size of the suffix field (Bytes)	S	0/16/32/64
1–5	Maximum field num	NF	5
1–6	Selection threshold	ST	30%, 10%
1–7	Min. Num. of messages in the cluster	NC	10
2–1	Valid Number of message types	NT	$[2, 30]$

– Format Precision: The correctness and conciseness respectively focus on the number of true formats contained in a cluster and the number of clusters required to describe a true format. Since the Inspector can cluster messages in high accuracy, we only consider completely pure and precise clusterings to be the correct inference cases. So, we measure the proportion of inferred message clusters that only contains all messages of one true format.
– Message Recall: We measure the proportion of messages covered by the interred formats in the dataset as the message recall. Since the format precision can reflect the ability to cover the true formats, we use this metric to evaluate the ability of the inferred protocol formats to recall messages from the dataset.

In order to automatically obtain the ground truth format of the message, we leverage Wireshark [13], a free and open-source network traffic analyzer, to get rid of the tedious manual format annotation works. For the correct message clusters, we further verify the correctness of the semantic inference results in the format by comparing it with the field sequence outputted from Wireshark [13], including the length fields, message type fields, constant fields, and dynamic fields.

4.3 Tunable Parameters

As shown in Table 2, the Inspector has a few tunable parameters. (1–1) The bit-width $\boldsymbol{BW}$ determines the size of the length field. We choose three most commonly used bit-widths for the Inspector. (1–2) We consider length fields that are contiguous before or at the beginning of the relevant payload as valid length field positions. Therefore, the orientation $\boldsymbol{O}$ can significantly restrict the search space of the length field in the message. (1–3) For the endianness $\boldsymbol{E}$, we consider both little-endian and big-endian in the first level length field search. When searching for length fields at other levels, we directly use the previous orientation and endianness to reduce the search space. (1–4) We determine the size of the suffix field S according to the general conventions of the protocol format. (1–5) The maximum field num NF limits the number of the length fields in the same layer to avoid meaningless searches. We set NF to 5 based on the experimental experience. (1–6) The selection threshold ST controls the minimum proportion of applicative messages for the selected length field pattern. We start the selection from the most general length field pattern, so we assign the first selection threshold to 30% and the subsequent selection thresholds to 5% to cope with the selection order. (1–7) The minimum number of messages in the cluster NC guarantees the minimum number of the target messages of each format. We set NC to 10 to exclude the cluster containing too few messages for the subsequent inference. (2–1) The number of message type NT should be within a proper interval. We set the interval to $[2, 50]$ based on the experimental experience.

4.4 Experimental Results

Table 3. Summary of message clustering results

Protocol	# Inferred formats	Precision	# Covered messages	Coverage
TLS	12/12	100.0%	44316/44316	100.0%
SSH	9/10	90.0%	12972/12988	99.9%
MQTT	21/21	100.0%	31413/33879	92.7%

The presentation of experimental results includes the message clustering and format inference. Except for one format of the protocol SSH, all message clustering results correspond exactly to the ground truths. Table 3 shows the summary of message clustering results. For the protocol TLS, 12 message clusters correspond to all 12 true formats in 100% accuracy and cover 100% messages of the dataset. For the protocol SSH, the Inspector finds 9 message clusters that correspond to 9 true formats in 90% accuracy and cover 99.6% messages of the dataset. For the protocol MQTT, the Inspector finds 21 message clusters that correspond to 21 true formats in 100% accuracy and cover 92.7% messages of the dataset. We manually check the excluded SSH messages and find that these messages are fully encrypted and have no segment for field inferences. These experimental results show that the Inspector can perform very well on the message clustering and can automatically filter out unformatted messages from the clustering.

Table 4 shows the detail of message clustering results. The length field pattern (ofs, w, ori, e, sub) has been introduced in Sect. 3.2. For message type (ofs, v, sub) in the table, ofs means the offset from the current tuple beginning, n means the value of the message type, sub means the message type in the lower layer. All the messages in the cluster have the same message type and their message type is consistent with the official documents. We also find that most of the inferred message types are closely related to the establishment of encrypted communication channels.

The Inspector handles the hierarchical formats as expected and successfully finds the message type in the lower level, such as the 5th–12th TLS formats in Table 4. Besides, the Inspector can still infer the correct length field pattern when the messages contain a suffix field. Notice that the Inspector puts 753 text messages into the 1st SSH format, which is the version announce message format of SSH, such as, 'SSH-2.0-OpenSSH_7.7'. For the text messages in the collected dataset, we can directly identify and filter them before conducting our length field inference. We also noticed that for all the inferred protocols, the orientation is "after" and the endianness is "big". We guess that this orientation is suitable for the design of a hierarchical protocol format, and this endian is convenient for the direct reading and format uniform of the network traffic.

The quality of format inference results greatly depend on the accuracy of the message clustering. Based on our message clustering results, the format inference results are very close to the ground truths. To better understand the results of message format inferences, we present a format inference example of TLS, SSH, and MQTT, as shown in Table 5, Table 7, and Table 6, respectively. For L(x, y) in the table, L means length field byte, x means the minimum legal value, y means the maximum legal value. For M(x), M means message type field byte, x means the message type value. For C(x), C means constant byte, x means the constant value. For V(x, y) and nV(x, y), V means variable byte, nV means an indefinite number of consecutive variable bytes, x and y have the same meaning as in L(x, y). For nT(x, y), nV means an indefinite number of consecutive text bytes, x and y have the same meaning as in L(x, y). The example TLS message is "Client

Table 4. Message clustering results in detail

Protocol	#	Inferred length field pattern	Inferred message type	True message type	# Message
TLS	1	(3, 2, 'after', 'big')	(0, 20)	Change Cipher Spec	5719
TLS	2	(3, 2, 'after', 'big')	(0, 21)	Alert	2487
TLS	3	(3, 2, 'after', 'big')	(0, 22)	Handshake	5714
TLS	4	(3, 2, 'after', 'big')	(0, 23)	Application Data	16459
TLS	5	(3, 2, 'after', 'big', [(2, 2, 'after', 'big')])	(0, 22, [(0, 1)])	Client Hello	2965
TLS	6	(3, 2, 'after', 'big', [(2, 2, 'after', 'big')])	(0, 22, [(0, 2)])	Server Hello	2877
TLS	7	(3, 2, 'after', 'big', [(2, 2, 'after', 'big')])	(0, 22, [(0, 4)])	New Session Ticket	1094
TLS	8	(3, 2, 'after', 'big', [(2, 2, 'after', 'big')])	(0, 22, [(0, 11)])	Certificate	1639
TLS	9	(3, 2, 'after', 'big', [(2, 2, 'after', 'big')])	(0, 22, [(0, 12)])	Server Key Exchange	1689
TLS	10	(3, 2, 'after', 'big', [(2, 2, 'after', 'big')])	(0, 22, [(0, 14)])	Server Hello Done	1691
TLS	11	(3, 2, 'after', 'big', [(2, 2, 'after', 'big')])	(0, 22, [(0, 16)])	Client Key Exchange	1722
TLS	12	(3, 2, 'after', 'big', [(2, 2, 'after', 'big')])	(0, 22, [(0, 22)])	Certificate Status	264
SSH	1	None (Text Message)	None	Version Announce	3168
SSH	2	(0, 4, 'after', 'big')	(5, 20)	Key Exchange Init	3154
SSH	3	(0, 4, 'after', 'big')	(5, 21)	New Keys	2295
SSH	4	(0, 4, 'after', 'big')	(5, 30)	DH[a] Key Exchange Init	1575
SSH	5	(0, 4, 'after', 'big')	(5, 31)	DH Key Exchange Reply	1518
SSH	6	(0, 4, 'after', 'big')	(5, 32)	DH GEX[b] Init	50
SSH	7	(0, 4, 'after', 'big')	(5, 33)	DH GEX Reply	29
SSH	8	(0, 4, 'after', 'big')	(5, 34)	DH GEX Request	49
SSH	9	(0, 4, 'after', 'big'), ('suffix', 32)	None	Encrypted Payload	1134
MQTT	1	(1, 1, 'after', 'big', 0)	(0, 16)	Connect	244
MQTT	2	(1, 1, 'after', 'big', 0)	(0, 32)	ConnAck	310
MQTT	3-9	(1, 1, 'after', 'big', 0)	(0, 48 / 49 / 50 / 51 / 52 / 53 / 56)	Publish (DUP[c] / QoS[d])	10402
MQTT	10	(1, 1, 'after', 'big', 0)	(0, 64)	PubAck	2748
MQTT	11	(1, 1, 'after', 'big', 0)	(0, 80)	PubRec	364
MQTT	12	(1, 1, 'after', 'big', 0)	(0, 98)	PubRel	334
MQTT	13	(1, 1, 'after', 'big', 0)	(0, 112)	PubComp	371
MQTT	14	(1, 1, 'after', 'big', 0)	(0, 130)	Subscribe	305
MQTT	15	(1, 1, 'after', 'big', 0)	(0, 144)	SubAck	355
MQTT	16	(1, 1, 'after', 'big', 0)	(0, 162)	Unsubscribe	6
MQTT	17	(1, 1, 'after', 'big', 0)	(0, 176)	UnsubAck	2
MQTT	18	(1, 1, 'after', 'big', 0)	(0, 192)	PingReq	8178
MQTT	19	(1, 1, 'after', 'big', 0)	(0, 208)	PingResp	7715
MQTT	20	(1, 1, 'after', 'big', 0)	(0, 224)	Disconnect	65
MQTT	21	(1, 1, 'after', 'big', 0)	(0, 240)	Auth	14

[a] Diffie-Hellman [b] Group Exchange SHA-1 [c] Duplicate Delivery [d] Quality of Service

Table 5. The format of TLS client key exchange

Token	Ground truth	Token	Ground truth	Token	Ground truth	Token	Ground truth
M (22)	Message type	L (0, 255)	Message length	C (0)	Pubkey length	nV (0, 255)	Pubkey
C (3)	Version	L (0, 255)	Message length	L (0, 255)	Pubkey length		
C (3)	Version	M (16)	Handshake type	L (0, 255)	Pubkey length		

Table 6. The format of MQTT subscribe request

Token	Ground truth	Token	Ground truth	Token	Ground truth	Token	Ground truth
C (130)	Header flags	V (0, 255)	Message identifier	C (0)	Property length	nT (0, 127)	Property
L (0, 255)	Message length	V (0, 255)	Message identifier	V (0, 255)	Property length		

Table 7. The format of SSH DH GEX request

Token	Ground truth	Token	Ground truth	Token	Ground truth	Token	Ground truth
L (0, 255)	Message length	C (0)	DH GEX Min	V (0, 255)	DH GEX num of bits	V (0, 255)	Padding
L (0, 255)	Message length	C (0)	DH GEX Min	C (0)	DH GEX num of bits	V (0, 255)	Padding
L (0, 255)	Message length	C (4)	DH GEX Min	C (0)	DH GEX max	V (0, 255)	Padding
L (0, 255)	Message length	C (0)	DH GEX Min	C (0)	DH GEX max	V (0, 255)	Padding
C (6)	Padding length	C (0)	DH GEX Num of Bits	C (32)	DH GEX max	V (0, 255)	Padding
M (34)	Message code	C (0)	DH GEX Num of Bits	C (0)	DH GEX max	V (0, 255)	Padding

Key Exchange", which sends a public key to the server for key negotiation. The example SSH message is "Diffie-Hellman (DH) Group Exchange SHA-1 (GEX) Request", which specifies the parameters for the subsequent Diffie-Hellman key exchange. The example MQTT message is "Subscribe Request", which is sent from the Client to the Server to register one or more topic subscriptions. We can see that the inferred formats can properly reflect the token properties and semantic fields of the true formats. Notice that the boundary of some fields is imprecise, such as the tokens of the "Pubkey Length" in TLS are "C(0) L(0, 255) L(0, 255)" and the tokens of the "DH GEX Num of Bits" in SSH are "C(0) C(0) V(0, 255) C(0)". This does not mean that our inference results are not applicable, but that we restrict the value range of the field based on the appeared values of the dynamic field in the network. Since the integration of the length field pattern and message type values are specific to the message format, the inferred formats can be clearly distinguished from each other, which guarantees the reliability of the inferred formats when applied to network security applications.

5 Conclusions

In this paper, we propose a protocol reverse engineering approach Inspector, which makes use of the semantic fields for the message clustering and format inference. In order to find the semantic fields in the messages, we propose a hierarchical length field model to search length field pattern candidates and develop two algorithms to infer the length fields and message type fields. The experimental results demonstrate the accurate message clustering and excellent format inference results. However, the Inspector has the limitation that the format inference results are not detailed enough. In future work, we plan to study a more elaborate approach to enrich the properties and semantic fields of the format inference results.

Acknowledgment. This work is supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (No.XDC02030100), the National Key Research and Development Program of China (Grant No.2018YFB0804704), and the National Natural Science Foundation of China (Grant No.U1736218).

References

1. Bermudez, I., Tongaonkar, A., Iliofotou, M., Mellia, M., Munafo, M.M.: Automatic protocol field inference for deeper protocol understanding. In: 2015 IFIP Networking Conference (IFIP Networking), pp. 1–9. IEEE (2015)
2. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003)
3. Bossert, G., Guihéry, F., Hiet, G.: Towards automated protocol reverse engineering using semantic information. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, pp. 51–62 (2014)
4. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 621–634. ACM (2009)
5. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 317–329. ACM (2007)
6. Chen, Y., Zang, T., Zhang, Y., Zhouz, Y., Wang, Y.: Rethinking encrypted traffic classification: a multi-attribute associated fingerprint approach. In: 2019 IEEE 27th International Conference on Network Protocols (ICNP), pp. 1–11. IEEE (2019)
7. Cohen, P., Adams, N.: An algorithm for segmenting categorical time series into meaningful episodes. In: Hoffmann, F., Hand, D.J., Adams, N., Fisher, D., Guimaraes, G. (eds.) IDA 2001. LNCS, vol. 2189, pp. 198–207. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44816-0_20
8. Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: protocol specification extraction. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 110–125. IEEE (2009)
9. Cui, W., Kannan, J., Wang, H.J.: Discoverer: automatic protocol reverse engineering from network traces. In: USENIX Security Symposium, pp. 1–14 (2007)
10. Cui, W., Peinado, M., Chen, K., Wang, H.J., Irun-Briz, L.: Tupni: automatic reverse engineering of input formats. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 391–402 (2008)
11. Dierks, T., Rescorla, E.: The transport layer security (TLS) protocol version 1.2 (2008)
12. Force, T.I.E.T.: Rfc index. https://tools.ietf.org/rfc/index
13. Foundation, W.: Wireshark - go deep. (2020). https://www.wireshark.org/
14. Gu, G., Perdisci, R., Zhang, J., Lee, W.: Botminer: clustering analysis of network traffic for protocol-and structure-independent botnet detection. In: Proceedings of the 17th USENIX Security Symposium, pp. 139–154. USENIX (2008)
15. Holzmann, G.J., Lieberman, W.S.: Design and Validation of Computer Protocols, vol. 512. Prentice hall Englewood Cliffs, Englewood Cliffs (1991)
16. Juan, C., Noah, M.J., Stephen, M., Dawn, S.: Binary code extraction and interface identification for security applications. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2010. The Internet Society (2010)
17. OASIS: Standards archive - oasis open. https://www.oasis-open.org/standards/
18. Postel, J., et al.: User datagram protocol (1980)
19. Postel, J., et al.: Transmission control protocol (1981)
20. Slonim, N., Tishby, N.: Agglomerative information bottleneck. In: Advances in Neural Information Processing Systems, pp. 617–623 (2000)

21. Sommer, R., Paxson, V.: Outside the closed world: On using machine learning for network intrusion detection. In: 2010 IEEE Symposium on Security and Privacy, pp. 305–316. IEEE (2010)
22. Standard, O.: Mqtt version 5.0. Retrieved 22 Jun 2020 (2019)
23. Wang, Y., et al.: A semantics aware approach to automated reverse engineering unknown protocols. In: 2012 IEEE 20th International Conference on Network Protocols (ICNP), pp. 1–10. IEEE (2012)
24. Wang, Y., Zhang, Z., Yao, D.D., Qu, B., Guo, L.: Inferring protocol state machine from network traces: a probabilistic approach. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 1–18. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21554-4_1
25. Ylonen, T., Lonvick, C., et al.: The secure shell (SSH) protocol architecture (2006)
26. Zhang, J., Moore, A.: Traffic trace artifacts due to monitoring via port mirroring. In: 2007 Workshop on End-to-End Monitoring Techniques and Services, pp. 1–8. IEEE (2007)
27. Zhang, Z., Zhang, Z., Lee, P.P., Liu, Y., Xie, G.: Proword: an unsupervised approach to protocol feature word extraction. In: 2014 IEEE International Conference on Computer Communications (Infocom), pp. 1393–1401. IEEE (2014)