IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 4, 2019, accepted March 12, 2019, date of publication March 15, 2019, date of current version April 2, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2905353

Protocol Specification Extraction Based on Contiguous Sequential Pattern Algorithm

YOUNG-HOON GOO, KYU-SEOK SHIM, MIN-SEOB LEE, AND MYUNG-SUP KIM
Department of Computer and Information Science, Korea University, Sejong 30019, South Korea

Corresponding author: Myung-Sup Kim (tmskim@korea.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) grant through the Korean Government (MSIT), (Development of Blockchain Transaction Monitoring and Analysis Technology) under Grant 2018-0-00539, and in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) through the Ministry of Education under Grant NRF-2018R1D1A1B07045742.

ABSTRACT As the amount of Internet traffic increases due to newly emerging applications and their malicious behaviors, the amount of traffic that must be analyzed is rapidly increasing. Many protocols that occur under these situations are unknown and undocumented. For efficient network management and security, a deep understanding of these protocols is required. Although many protocols reverse engineering methods have been introduced in the literature, there is still no single standardized method to completely extract a protocol specification, and each of the existing methods has some limitations. In this paper, we propose a novel protocol reverse engineering method to extract an intuitive and clear protocol specification. The proposed method extracts field formats, message formats, and flow formats as protocol syntax by using a contiguous sequential pattern algorithm three times hierarchically and defining four types of the field formats. Moreover, the proposed methods can extracts protocol semantics and a protocol finite state machine. The proposed method sufficiently compresses input messages into a small number of message formats in order to easily identify the intuitive structure of an unknown protocol. We implemented our method in a prototype system and evaluated the method to infer message formats of HTTP (a text protocol) and DNS (a binary protocol). The experimental results show that the proposed method infers HTTP with 100% correctness and 99% coverage. For DNS, the proposed method achieves 100% correctness and coverage.

INDEX TERMS Contiguous sequential pattern algorithm, network security, protocol reverse engineering.

I. INTRODUCTION

A. MOTIVATION

With the development of information technology (IT), the use of Internet traffic has become more common and diverse, and the traffic volume of Internet applications and malicious behaviors using the network have rapidly increased. Although many technologies for efficient network management, such as software defined networking (SDN) and risk management in cloud computation [1]–[3], are being studied, these can only be applied to the identified traffic. They are not even applicable for unidentified traffic, but many protocols that occur under these circumstances are unknown and undocumented protocols. According to the report by Sophos, IT managers cannot identify 45% of their organization's network traffic that belongs to unknown protocols [4]. These protocols include proprietary protocols such as Skype protocol, industrial communication protocols used in an industrial control system (ICS) environment, or customized protocols used in various types of attacks [5]. For efficient network management and security, protocol reverse engineering, the act of extracting an unknown protocol specification, is very important. Securing an unknown protocol specification in the field of network monitoring implies that information on unknown traffic occurring in the target network can be acquired; thus, it is possible to classify the traffic generated by these unknown protocols to identify the network usage status, establish a network expansion plan, and control the bandwidth for specific protocols [6]. In the field of network security, it can be helpful in analyzing network vulnerabilities or providing useful information to firewalls and intrusion detection and prevention systems (IDSes/IPSes) for detecting and blocking previously unknown attacks. In particular, many prior studies of protocol reverse engineering have focused on

The associate editor coordinating the review of this manuscript and approving it for publication was Peter Langendorfer.

analyzing the command and control (C&C) protocols used in botnets. Protocol reverse engineering can also be used for legacy software integration, intelligent DPI, penetration testing, and building an application-aware fuzzer system [7].

B. PROBLEM STATEMENT

Many protocol reverse engineering methods have been introduced in the literature, but there is still no single standardized method to extract a protocol specification completely, and each of the existing methods has some limitations. Traditional protocol reverse engineering methods are painstaking and laborious tasks, so they are time-consuming and prone to errors. To address these problems, many automatic protocol reverse engineering methods have been proposed, but they also have limitations in extracting well-trimmed message formats.

First, many of the previous works cannot extract all of the syntax, semantics, and finite state machine (FSM) that are related to the three major elements of protocol. Second, some existing methods cannot extract intuitive message formats. They extract too many message formats, so they keep the network analysts waiting. As a result, it is hard to intuitively grasp the structure of the target unknown protocol. Third, some existing methods extract unclear message formats. When they extract a message format, they merely use a list of static fields that have fixed lengths and static values, and then they fill the blank parts between the static fields with gap data that can have many values and lengths, resulting in an incomplete message format.

C. CONTIRIBUTION

In this paper, we propose a novel protocol reverse engineering method to extract a protocol specification using a contiguous sequential pattern (CSP) algorithm. We aim to achieve several important goals.

- **Fully Automation.** The proposed method does not require any manual intervention.
- **Abundant Specification.** The proposed method extracts protocol syntax, semantics, and FSM. These outputs reflected all of the major element of protocol.
- **Intuitive Specification.** The proposed method sufficiently compresses input messages into a small number of message formats. As a consequence, the network analysts can certainly identify the intuitive structure of an unknown protocol.
- **Clear Specification.** We defined four types of field format. The proposed method extracts message formats that are fully filled with these four types without any blank part in the message formats. Beside, we defined three types of formats which are field format, message format, and flow format for deeper understanding of an unknown protocol, and the proposed method extracts them as protocol syntax using CSP algorithm hierarchically.

We implemented our method in a prototype system and evaluated the method to infer message formats of HTTP (a text protocol) and DNS (a binary protocol).

The remainder of this paper is organized as follows. Section II describes the related works, and Section III explains the CSP algorithm developed by us. Section IV explains the overall design of the proposed protocol specification extraction method in detail. In Section V, we discuss the experimental results proving the superiority of the proposed method. Finally, Section VI presents the concluding remarks and future work.

II. RELATED WORKS

In this section, we describe approaches for protocol reverse engineering with categorization, and we describe the existing limitations of previous works.

A. PROTOCOL REVERSE ENGINEERING APPROACHES

The traditional approach for protocol reverse engineering methods is manual; thus, it is incredibly labor-intensive and error-prone. Two typical examples are the SAMBA project [8] and the Pidgin project [9] for interoperability. The SAMBA project took 12 years to generate a Microsoft server message block (SMB) protocol specification. The Pidgin project, a multi-platform integrated messenger client, requires patching to support the target protocol whenever the target protocol is changed, and it takes months. In today's high-speed network environment, these manual protocol reverse engineering methods cannot cope with the emergence and renewal of rapidly increasing applications or the evolution of various highly intelligent attacks.

In order to address these problems, automatic protocol reverse engineering has been proposed over the past decade, and has become a hot topic in the research field of network management. Automatic protocol reverse engineering can be categorized into two types: application-based and network trace-based methods.

The application-based method uses source code or execution traces, which are files logging how the program binary that implements the protocol processes messages. Dispatcher [10] and Liu *et al.* [11] infers protocol syntax based on the application-based method using dynamic taint analysis. These are effective methods to infer the protocol specification, but the program binary or its source code is hardly available in real-world situations. For example, the program binary of a malicious botnet C&C server is likely to exist in an external network, not in the target network, and the malicious server makes use of obfuscation interference techniques to protect itself from being detected.

By contrast, the network trace-based method is more realistic and among the most widely-used method, as it solely analyzes network traces captured by monitoring network packets of the target protocol without access to the program binary. Hence, in this paper, we focus on the network trace-based method. These methods mainly fall in 1 of 3 approaches: natural language processing, bioinformatics, and data mining.

ProDecoder [12], PRISMA [13], and Li *et al.* [14] are based on algorithms originating from natural language processing to identify protocol keywords by finding tokens that

appear together frequently in messages. However, because binary protocols typically pack data more densely and do not use delimiters to distinguish protocol fields, these methods are not suitable to infer a binary protocol specification.

PI [15], ScriptGen [16], Discoverer [17], and Netzob [18] use sequence alignment techniques based on bioinformatics to determine the similarity of messages and cluster them. Then, they separate messages into the fields by identifying the common parts among messages in the same cluster. The amount of data has a significant impact on the quality of protocol specification, but the multiple sequence alignment has exponential complexity because the sequence alignment algorithm always uses only two messages as input at a time [19].

As for the methods using data mining techniques, they are relatively recently proposed methods. AutoReEngine [20] consecutively uses the Apriori algorithm to extract protocol keywords and message formats. Wang *et al*. [21] and Ji *et al*. [22] first use Aho-Corasick algorithm to extract protocol keywords, and then extract the message formats using a frequent pattern (FP)-growth algorithm. These data mining techniques can possibly use all messages as input at a time, contrary to sequence alignment, but they also have a computational cost for candidate selection and yielding support values. Further, it is crucial to know how to optimize the result to make the result an intuitive and clear specification. This will be mentioned in the next part.

FieldHunter [23] and Ladi *et al*. [24] are state of the art approaches. Both methods can extracts more specific types of semantics than other previous works. They determine the boundaries of the fields in each message type by inferencing the pre-defined semantics types. However, they do not consider FSM and may extract too many message formats.

B. EXISTING LIMITATIONS

The fundamental goal of protocol reverse engineering is to extract as much information as possible related to the three major elements of protocol. These elements of protocol are syntax, semantics, and timing, and they define what, how, and when to communicate, respectively. Syntax refers to the format of data, including the order in which data is represented. Semantics indicates the meaning of each region of data. Timing has two characteristics, indicating when to transmit data, i.e., the order in which they are transmitted, and how fast to transmit. For results of protocol reverse engineering, timing can be represented as an FSM, i.e., a finite state automaton that expresses the occurrence order, occurrence probability, and direction of message types through analyzing the behavior of messages. However, as shown in Table 1, most previous methods only extract some of the syntax, semantics, and FSM of the protocol as a result.

Other existing limitations include the extraction of unintuitive message formats or unclear message formats. In the former case, although the number of inferred message formats must be compressed enough to represent the unknown target protocol for an effective understanding, many of previous methods extract too many message formats; given 3500 HTTP messages, the number of extracted message formats from Netzob [18] is 2500 when the similarity threshold is set to 50%. Therefore, it is difficult for network analysts to grasp the overall structure of the target protocol. Moreover, these methods tend to excessively subdivide the message format into too many fields, as shown in Fig. 1. Fig. 1 is a notable example of an overly fragmented HTTP request message format.

In the latter case, the inferred message format should be separated by fields in succession, but many of the previous methods extract message formats that are composed of only static fields, and they refer to the blank part between two static fields as the gap (some previous works call this the dynamic field). Hence, network analysts receive unclear message formats that have many blank parts. Fig. 2 shows this problem by exemplifying the HTTP protocol.

In this paper, we propose a novel protocol reverse engineering method to address above-mentioned limitations. The proposed method extracts syntax, semantics, and FSM of the target protocol, and it can extract intuitive message formats by using a CSP algorithm hierarchically. Moreover, it can extract clear message formats by extracting additional fields

TABLE 1. Outputs of previous methods.

Methods	Output		
	Syntax	Semantics	FSM
Taint Analysis			
Dispatcher [10]	✓	✓ (weak)	
Liu *et al*. [11]	✓		
Natural language processing			
ProDecoder [12]	✓		
PRISMA [13]	✓		✓
Li *et al*. [14]	✓		
Bioinformatics			
PI [15]	✓		
ScriptGen [16]			✓
Discoverer [17]	✓	✓ (weak)	
Netzob [18]	✓	✓ (weak)	✓ (manual)
Data mining			
AutoReEngine [20]	✓		✓
Wang *et al*. [21]	✓		
Ji *et al*. [22]	✓		
State of the art			
FieldHunter [23]	✓	✓	
Ladi *et al*. [24]	✓	✓	

Field 0	Field 1	Field 2	Field 3	Field 4	Field 5	...	Field 24	Field 25	Field 26	...	Field 45
GE	T|20|/	app/newmail/editor/	|20|HTTP/1.1|0d||0a|Host:|20|	adimg	.nate.com|0d||0a|Connection:|20|keep-alive|0d||0a|	...	ml+xml,a p	p	: /	...	...
		app/newmail/elapsed		cyad		...				...	...
POS		...		...		...			lication	...	...
		Total # : 141		Total # : 8		...				...	...

FIGURE 1. Example of an overly fragmented message format.

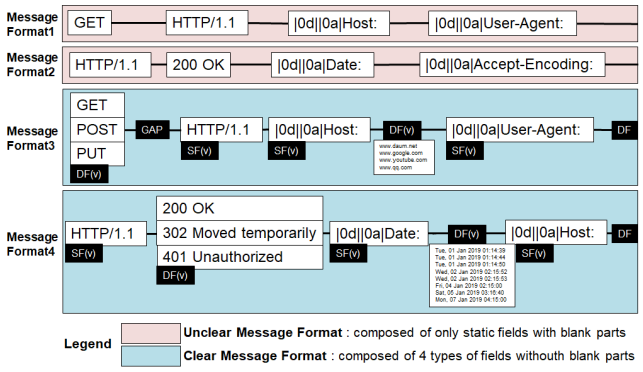

FIGURE 2. Examples of unclear message formats and clear message formats.

using statistical techniques for blank parts in initial message formats.

III. CONTIGUOUS SEQUENTIAL PATTERN (CSP) ALGORITHM

In this section, we first explain the basic concept of the CSP algorithm we developed, and then show how to apply CSP algorithm hierarchically for protocol specification extraction.

A. BASIC CONCEPT OF CSP ALGORITHM

Data mining is the process of navigating and analyzing meaningful patterns or rules in a vast amount of data in a database. The knowledge that one can gain from data mining includes association rules, classification rules, sequential patterns, clustering rules, generalization rules, and similarity search. One of the association rule mining algorithms, the Apriori algorithm [25], is historically significant because it is simple and easy to learn. It can reduce the number of itemsets to be generated in each pass by reducing the number of candidate itemsets; thus, it has spawned many algorithms. These Apriori-like algorithms use the Apriori principle, i.e., any subset of a frequent itemset must be frequent, and also any superset of an infrequent itemset must be infrequent. AprioriTID [26], a variation of the Apriori algorithm, extracts the same output as the Apriori algorithm, but reduces the computational complexity for yielding support of itemsets.

Sequential pattern mining is a similar process to association rule mining, but the main difference is that the purpose of association rule mining is to extract frequently occurring concurrent itemsets, whereas the purpose of sequential pattern mining is to extract frequently occurring time-series patterns. Sequential pattern mining algorithms using the Apriori principle include AprioriAll, AprioriSome [27], and Generalized Sequential Pattern (GSP) [28]. GSP is an improved version of AprioriAll that allows users to adjust gap constraints, sliding time window constraints, and constraints on whether to allow pattern discovery in different levels of a taxonomy, so it is possible to extract patterns suitable for a user's own interests.

Among these data mining techniques, the technique that we need for inferring the specification of the unknown protocol is sequential pattern mining. In particular, among the sequential pattern mining algorithms, we need an algorithm to find contiguous patterns that do not allow gaps in a pattern. The reason is that a protocol message is a contiguous sequence of fields, and a field is a contiguous sequence of bytes. In addition, by mining the contiguous sequential patterns in messages that occur in chronological order, it is possible to infer the order in which the messages occur.

We developed an algorithm suitable for extracting a protocol specification, called CSP, whose goal is to find frequent contiguous common subsequences. This algorithm is based on the Apriori principle. It is a modified GSP algorithm with gap constraints, whose minimum and maximum gaps are zero to avoid gaps in patterns to be found. Moreover, it has improved the computational complexity of the counting support by using the concept of AprioriTID, and it has improved the computational complexity of the candidate generation process by utilizing hash. In other words, the CSP algorithm is an integrated version of the advantages of AprioriAll, GSP, AprioriTID, and AprioriHash [29], [30]. The proposed method uses CSP to extract field formats, message formats, and flow formats.

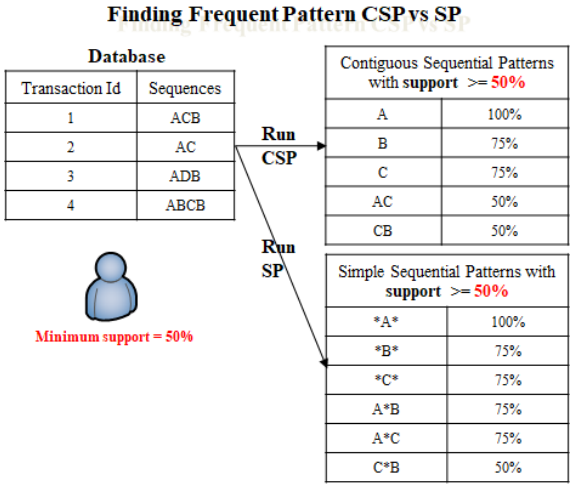

FIGURE 3. Output of the contiguous sequential pattern (CSP) algorithm and a simple sequential pattern mining algorithm.

A threshold, named the minimum support, is very significant for this algorithm. The support for a target subsequence is defined as the ratio of sequences containing the target subsequence to the total sequences. This algorithm can obtain frequent contiguous patterns by extracting subsequences whose support is higher than the user-defined minimum support. Fig. 3 illustrates the concept of this algorithm by comparing the final output of CSP with a simple sequential pattern mining algorithm. The asterisk in Fig. 3 indicates a gap.

This algorithm uses a bottom-up approach. First, it generates length-1 candidate subsequences and determines frequent subsequences by calculating the support of each candidate. Subsequently, it expands this process by extending

subsequences to larger and larger subsequences as long as no more frequent patterns are extracted.

$$SequenceSet = \{S_1, S_2, \ldots, S_n\} \tag{1}$$

$$S_i = \left\{Sequence_{ID}, \langle I_1 I_2 \ldots I_m \rangle\right\} \tag{2}$$

Equation (1) shows a set of sequences that is an input database for the CSP algorithm. In Equation (2), S is an element of a *SequenceSet* that consists of a sequence identifier and contiguous items. As will be explained later, the units of sequence and item depend on the type of format to be extracted. The sequence identifier is used to utilize the mechanisms of AprioriTID that help it improve the performance of support calculation and candidate generation, by reducing the time to read the database.

Algorithm 1 Contiguous Sequential Pattern Algorithm

Input: *SequenceSet, Min_Supp*
Output: *SubSequenceSet*
01: ***foreach*** *sequence S in SequenceSet* ***do***
02: ***foreach*** *item i in sequence* ***do***
03: $L_1 = L_1 \cup i$;
04: ***end***
05: ***end***
06: $k = 2$;
07: ***while*** $L_{k-1} \neq \Phi$ ***do***
08: ***foreach*** *candidate c in* L_{k-1} ***do***
09: *supp* = ***calSupport****(c, SequenceSet)*;
10: ***if*** *supp* < *Min_Supp* ***then***
11: $L_{k-1} = L_{k-1} - c$;
12: ***end***
13: ***end***
14: L_{k-1} = ***extractCandidate***(L_{k-1});
15: k++;
16: ***end***
17: $SubSequenceSet = \cup_k L_k$;
18: ***deleteSubset*** *(SubSequenceSet)*;
19: ***return*** *SubSequenceSet*;

Algorithm 1 is the pseudo-algorithm of CSP. First, it extracts length-1 subsequences from all the sequences and stores them in the length-1 subsequence set, L_1 (Alg.1, Lines 1–5). From the length-1 subsequences, it extracts all length-k candidate subsequences by increasing the length until no newer subsequences or candidates are to be extracted (Alg.1, Lines 6–16). This iteration process consists of two parts. At the first part, it eliminates candidates that do not satisfy the minimum support threshold after obtaining the support value from the calculation (Alg.1, Lines 8–13). At the second part, it extracts length-k candidates by using length-(k–1) candidates according to the Apriori strategy (Alg.1, Line 14). As a final step, a relation of inclusion between subsequences is checked; if the relation is found, the included subsequences are deleted (Alg.1, Line 18).

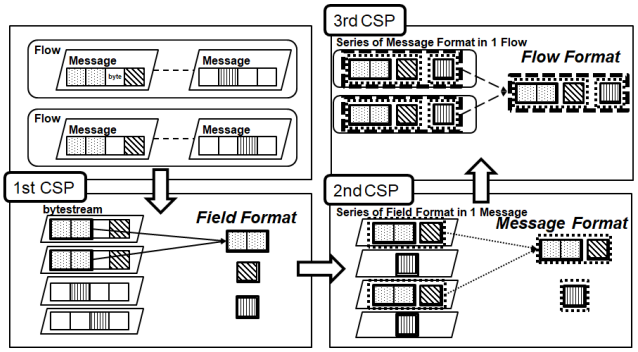

FIGURE 4. Process of extracting protocol syntax using hierarchical CSP algorithm.

B. HIERARCHICAL CSP ALGORITHM

The proposed method extracts field formats, message formats, and flow formats as protocol syntax by using the CSP algorithm three times hierarchically, as shown in Fig. 4. The only differences in each level are the units of input transaction, i.e., the sequence, length-1 item unit constituting the sequence, and support unit. Table 2 shows these differences in each level.

$$FlowSequenceSet = \{F_1, F_2, \ldots, F_n\}, \quad \text{n} = \text{number of flows in the target protocol} \tag{3}$$

$$F_i = \{Flow_{ID}, \langle P_1, P_2, \ldots, P_n \rangle\}, \quad \text{n} = \text{number of packets in } F_i \tag{4}$$

$$P_i = \langle B_1, B_2, \ldots, B_n \rangle, \text{ where } 0\text{x}00 \leq B_i \leq 0\text{x}FF, \quad \text{n} = \text{number of bytes in } P_i \tag{5}$$

Fig. 5 shows the flow chart of hierarchical CSP. All the following equations to be described are based on (1) and (2) mentioned above, except of the equations related to output such as *FieldFormatSet*, *MessageFormatSet*, and *FlowFormatSet*. First, it generates *FlowSequenceSet*, as shown in (3), from network traces of the target protocol. A flow is a bidirectional set of packets having the same 5-tuple: source and destination IP address, source and destination port number, and transport layer protocol. In Equation (4), F_i is an element of *FlowSequenceSet*, and refers to a flow. In Equation (5), P_i refers to a packet, and B is each one byte of the payload. Next, as will be explained in Section 4, the process conducts a message assemble stage to assemble packets of each flow into a message unit.

$$FlowSequenceSet' = \left\{F'_1, F'_2, \ldots, F'_n\right\}, \quad \text{n} = \text{number of flows in the target protocol} \tag{6}$$

$$F'_i = \{Flow_{ID}, \langle M_1, M_2, \ldots, M_n \rangle\}, \quad \text{n} = \text{number of messages in } F'_i \tag{7}$$

$$M_i = \left\{Message_{ID}, \langle B_1, B_2, \ldots, B_n \rangle, fromFlow_{ID}\right\}, \quad \text{n} = \text{number of bytes in } M_i, \tag{8}$$

TABLE 2. Differences in each level of the hierarchical CSP algorithm.

Level	Output	Database	Item unit	Support unit
1st CSP	Field formats	*FlowSequenceSet*	One byte	The number of req.[res.] messages containing the candidate / the number of total req.[res.] messages
2nd CSP	Message formats	*MessageSequenceSet*	Field format	The number of req.[res.] messages containing the candidate / the number of total req.[res.] messages
3rd CSP	Flow formats	*MessageSequenceSet`*	Message format	The number of flows containing the candidate / the number of total flows

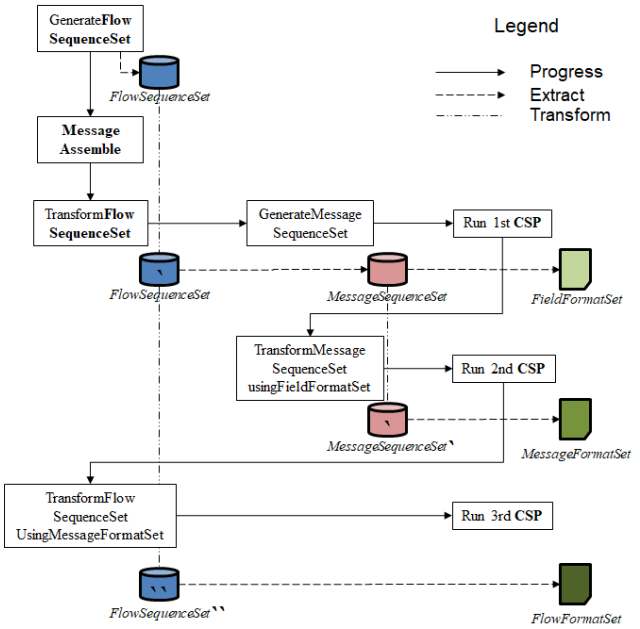

FIGURE 5. Flowchart of hierarchical CSP algorithm.

Then, all of F_i changes to F_i' of (7) that have a series of messages, so *FlowSequenceSet* changes to *FlowSequenceSet*′, as in (6). In Equation (8), M_i refers to a message and $fromFlow_{ID}$ refers to the $Flow_{ID}$ to which the message belongs.

$$MessageSequenceSet = \{M_1, M_2, \ldots, M_n\}, \quad n = \text{number of messages in target protocol} \tag{9}$$

$$FieldFormatSet = \{FieldF_1, FieldF_2, \ldots, FieldF_n\}, \quad n = \text{number of field formats of target protocol} \tag{10}$$

$$FieldF_i = \left\{ \begin{matrix} FieldFormat_{ID}, \langle B_1, B_2, \ldots, B_n \rangle, \\ FromMessage_{ID}Set \end{matrix} \right\}, \quad n = \text{number of bytes of the field format} \tag{11}$$

The algorithm generates *MessageSequenceSet*, as shown in (9), having all messages of the target protocol, and runs the first CSP on *MessageSequenceSet*. Then, it extracts a set of field formats as shown in (10). Equation (11) is a field format, and it has $FromMessage_{ID}Set$, the set of $Message_{ID}$s of messages that contain the field format. Support for the field format can be calculated as the ratio of the size of $FromMessage_{ID}Set$ to the number of total messages. A field format extracted by the first CSP is a contiguous common bytestream, and the type of field is a static field that has static value and length. The process of extracting dynamic fields will be described in the next section.

$$MessageSequenceSet' = \{M_1', M_2', \ldots, M_n'\}, \quad n = \text{number of messages in target protocol} \tag{12}$$

$$M_i' = \left\{ \begin{matrix} Message_{ID}, \langle FieldF_1 FieldF_2 \ldots FieldF_n \rangle, \\ FromFlow_{ID} \end{matrix} \right\}, \quad n = \text{number of field formats in the message} \tag{13}$$

$$MessageFormatSet = \{MsgF_1, MsgF_2, \ldots, MsgF_n\}, \quad n = \text{number of message formats of target protocol} \tag{14}$$

$$MsgF_i = \left\{ \begin{matrix} MessageFormat_{ID}, \langle FieldF_1 FieldF_2 \ldots \\ FieldF_n \rangle, FromMessage_{ID}Set \end{matrix} \right\}, \quad n = \text{number of field formats in the message format} \tag{15}$$

Now, we can easily transform M_i to M_i' as shown in (13) by using *FieldFormatSet*, because each field format has $FromMessage_{ID}Set$. Then, *MessageSequenceSet* changes to *MessageSequenceSet*′, like in (12). It runs the second CSP on *MessageSequenceSet*′ to extract a set of message formats, as shown in (14). Equation (15) is a message format. Support for the message format can be calculated as the ratio of the size of $FromMessage_{ID}Set$ to the number of total messages, as with support for the field format. A message format extracted by the second CSP is a contiguous series of field formats. This message format is the skeleton of the final message format that will be described in the next section.

$$FlowSequenceSet'' = \{F_1'', F_2'', \ldots, F_n''\}, \quad n = \text{number of flows in target protocol} \tag{16}$$

$$F_i'' = \{Flow_{ID}, \langle MsgF_1, MsgF_2, \ldots, MsgF_n \rangle\}, \quad n = \text{number of message formats in the flow} \tag{17}$$

TABLE 3. Four types of field format.

Field Format Type	Value Attribute	Length Attribute	Remarks	
SF(v)	Static	Fixed	Value : predictable	Length : predictable
DF(v)	Dynamic	Fixed or variable	Value : predictable	Length : predictable
DF	Dynamic	Fixed or variable	Value : unpredictable	Length : predictable
GAP	Dynamic	Fixed or variable	Value : unpredictable	Length : unpredictable

$$FlowFormatSet = \{FlowF_1, FlowF_2, \ldots, FlowF_n\}, \quad n = \text{number of flow formats of target protocol} \tag{18}$$

$$FlowF_i = \left\{ \begin{matrix} FlowFormat_{ID}, \langle MsgF_1, MsgF_2, \ldots \\ , MsgF_n \rangle, FromFlow_{ID}Set \end{matrix} \right\}, \quad n = \text{number of message formats in the flow format} \tag{19}$$

The next step is transforming F_i' to F_i'' as shown in (17), by using *MessageFormatSet*. It can be easily transformed because each message format has $FromMessage_{ID}Set$, and each message has $FromFlow_{ID}$. Therefore, $FlowSequenceSet'$ changes to $FlowSequenceSet''$ like in (16). The algorithm runs the third CSP on $FlowSequenceSet''$ to extract a set of flow formats, as shown in (18). Equation (18) is a flow format.

Support for the flow format can be calculated as the ratio of the size of $FromFlow_{ID}Set$ to the number of total flows. A flow format is a contiguous series of message formats.

IV. OVERALL DESIGN OF THE PROPOSED METHOD

In this section, we describe the overall design of the proposed protocol specification extraction method in detail after defining the terminology for better understanding.

In general, a protocol field can be categorized by its value and length. If categorized by the value of the field, it is divided into the static field and the dynamic field. A static field means it has only one value, and a dynamic field means it has multiple values. If categorized by the length of the field, it is divided into the fixed length field and the variable length field. A fixed length field means its length is fixed, and a variable length field means its length is variable. Thus, a static field is always a fixed length field, whereas a dynamic field is not necessarily a variable length field.

Hitherto, many of the previous works extract static fields and dynamic fields or only static fields, when they infer protocol syntax. They refer to the non-static fields in a message format the dynamic fields or the gaps. In contrast, to extract clear and detailed protocol syntax, the proposed method extracts four different types of protocol fields that we defined. Table 3 shows the four types of the field format.SF(v) is a static and fixed length field. There are three types of the dynamic field (DF): DF(v), DF, and GAP. Their lengths may be fixed or variable. The field formats marked with (v) indicate that their values are predictable because the randomness of their values is not too high. Hence, SF(v) and DF(v) store their values.

DF and GAP are the field formats whose values are unpredictable, because the randomness of their values is too high to predict. The difference between DF and GAP is whether the length is predictable. Although the values of both DF and GAP are unpredictable, the length of DF is somewhat predictable because of its low randomness. Thus, SF(v), DF(v), and DF stores their minimum, maximum, and average lengths. The proposed method first extracts SF(v) using the first CSP, and extracts DF(v) using CSP recursively. Next, after extracting the message formats, it extracts additional SF(v), DF(v), DF, and GAP using statistical methods for the blank parts of the message format that are not SF(v) and DF(v).

Fig. 6 shows a flowchart of the proposed method. The method is composed of five stages: preprocessing, message assemble, syntax inference, semantics inference, and behavior inference. In the preprocessing stage, if a user inputs the network traces of a single unknown protocol, it generates flows and removes control packets that have no payload, such as TCP three-way-handshake packets. Subsequently, it rectifies abnormal packets such as retransmission packets, out-of-order packets, and cross-order packets [31]. Rectifying abnormal packets must be done preemptively before analyzing the protocol structure to correctly split each flow into messages in the message assemble stage. The message assemble, syntax inference, semantics inference, and behavior inference stages are then performed step-by-step. These stages are described in the following parts of this section.

A. MESSAGE ASSEMBLE

A protocol is a set of rules for communication between two entities. When two entities communicate, the data they send and receive are transmitted as packet units, so these packets must be assembled into messages that are application-level data units (ADUs).

We defined how to split each flow into message units as shown in Algorithm 2. When each flow is inputted, it is checked for whether it is a transmission control protocol (TCP)-based flow or a user datagram protocol (UDP)-based flow. If it is a TCP-based flow, the packets in the flows are assembled into messages after setting the message unit to a series of consecutive packets with the same direction. The algorithm traverse all packets in the flow and checks the

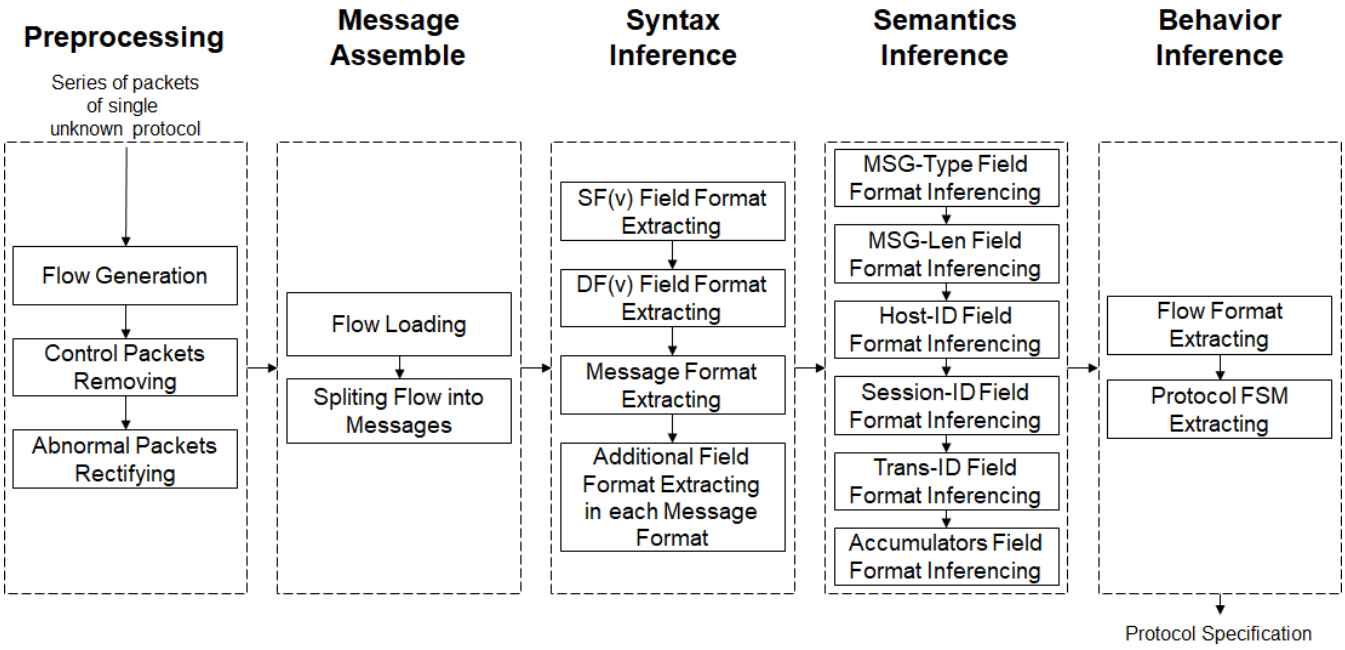

FIGURE 6. Flowchart of the proposed method.

directions of each packet. If the packet is the first packet of the flow or the direction of the packet is different from the direction of the packet that is checked earlier, the algorithm generates a new empty message. Otherwise, the algorithm continues to insert the packet into the generated message as long as the direction is not changed. If it is a UDP-based flow, each of the packets in the flow simply becomes a message. This is based on the following grounds.

TCP uses a stream-based and connection-oriented communication model, and UDP uses a simple connectionless communication model. We validated the message assemble method that we defined through experiments on eight protocols, namely hypertext transfer protocol (HTTP), file transfer protocol (FTP), simple mail transfer protocol (SMTP), and post office protocol 3 (POP3) for text protocols, and domain name system (DNS), real-time transport protocol (RTP), server message block (SMB), and dynamic host configuration protocol (DHCP) for binary protocols. HTTP, FTP, SMTP, POP3, and SMB are TCP-based protocols, and DNS, RTP, and DHCP are UDP-based protocols.

B. SYNTAX INFERENCE

The syntax inference stage is composed of four modules: SF(v) field format extracting, DF(v) field format extracting, message format extracting, and additional field format extracting in each message format. The ultimate goal of the syntax inference stage is to extract clear and intuitive message formats that are fully filled with SF(v), DF(v), DF, and GAP. Fig. 7 shows the overview of the syntax inference stage, along with the output changing shape for each module.

The SF(v) field format extracting module, as described above, extracts field formats whose type is SF(v) by running the first CSP after setting the transaction unit to a message sequence and setting the length-1 item unit to one byte.

The DF(v) field format extracting module extracts field formats whose type is DF(v) by using the CSP algorithm recursively. The pseudo-algorithm of recursive CSP is shown as Algorithm 3. When each SF(v) from the first CSP is inputted, it is checked whether this SF(v) can be converted to DF(v) through the following three conditions (Alg.3 Lines 1–2):

1) The position variance of the SF(v) is low. This means the position of this field format is almost fixed.
2) The support of the SF(v) is not 1. This means this field format may have other values, because there are message sequences that do not have this SF(v).
3) The difference between the maximum depth and minimum offset of the SF(v) is low. This means the randomness of the length for this field format is low, so the length is predictable.

If these three conditions hold, it creates a new set of message sequences that do not contain the SF(v) (Alg.3 Lines 3–6). It truncates these message sequences based on the minimum offset and maximum depth of the SF(v) (Alg.3 Lines 7–9). It runs the CSP algorithm on this set of message sequences, and it stores the bytestream that has the highest support among the CSP results into the SF(v) as another value of the SF(v) (Alg.3 Lines 10–12). This iteration process continues until no more bytestreams are extracted from the CSP (Alg.3 Lines 5–13). After all iteration processes are done, the type of this field format is converted to DF(v).

Algorithm 2 Message Assemble Algorithm

Input: *Flows consisting of packets*
Output: *Flows consisting of messages*

```
01: foreach flow in set of network traces do
02:   if flow is TCP flow then
03:     foreach packet in flow do
04:       if (packet is the first packet of flow) then
05:         M_temp = new Message;
06:         M_temp → insertPkt (packet);
07:         M_temp → direction = packet.direction;
08:       end
09:       else
10:         if packet.direction == M_temp → direction
              then
11:           M_temp → insertPkt (packet);
12:         end
13:         else
14:           flow→insertMsg (M_temp);
                    M_temp = NULL;
15:           M_temp = new Message;
16:           M_temp→insertPkt(packet);
17:           M_temp→direction=packet.direction;
18:         end
19:       end
20:       if packet is the last packet of flow then
21:         flow → insertMsg(M_temp); M_temp = NULL;
22:       end
23:     end
24:   end
25:   else if flow is UDP flow then
26:     foreach packet in flow do
27:       M_temp = new Message;
28:       M_temp → insertPkt (packet);
29:       M_temp → direction = packet.direction;
30:       flow → insertMsg(M_temp); M_temp = NULL;
31:     end
32:   end
33: end
```

Fig. 8 shows the process of recursive CSP by exemplifying the HTTP protocol. The system performs recursive CSP for two SF(v)s, namely ''GET'' and ''200'', because they satisfy the three conditions above mentioned. In the first iteration of recursive CSP for ''GET'', the system creates a set of message sequences that do not contain ''GET'' and cuts them based on the minimum offset and maximum depth of ''GET''. Then, the system runs CSP on the set of message sequences. Among the results of the first iteration, ''POST'' has the highest support, so it is stored as another value of ''GET''. In the next iteration, the system removes the message sequences that do not contain ''POST'' from the set of message sequences, and runs the CSP algorithm on the set of message sequences. This procedure is repeated until no more results are extracted from the CSP. As a result, the SF(v) is converted to a DF(v) having four values, i.e., ''GET'', ''POST'', ''HEAD'', and ''PUT''.

Algorithm 3 Recursive CSP Algorithm

Input: *FieldFormatSet and original MessageSequenceSet*
Output: *FieldFormatSet including DF(v)s*

```
01: foreach FieldF_i from 1st CSP do
02:   if (FieldF_i.PosVar ≤ 200)&(FieldF_i.Supp ≠ 1.0)
        &(FieldF_i.max_depth − FieldF_i.min_offset
        ≤ 40) then
03:     MessageSequenceSet_Temp = MessageSequence-
        Set;
04:     FieldF_Temp = FieldF_i;
05:     do while no more result is extracted by CSP
06:       MessageSequenceSet_Temp− ={M|M contain
          FieldF_Temp};
07:       foreach M_i from MessageSequenceSet_Temp
                do
08:         M_i.⟨B_1 B_2 ... B_n⟩ = M_i.⟨B_FieldF_Temp.min_offset
            ... B_FieldF_Temp.max_depth⟩;
09:       end
10:       CSP (MessageSequenceSet_Temp, Min_Supp)
11:       FieldF_Temp = bytestream with the highest
          Supp from CSP result;
12:       FieldF_i.values = FieldF_i.values
                          ∪FieldF_Temp;
13:     end
14:   FieldF_i.type = DF(v);
15:   end
16: end
```

The message format extracting module, as described in Section 3, extracts message formats by running a second CSP after setting the transaction unit to a message sequence and setting the length-1 item unit to SF(v) and DF(v).

The additional field format extracting in each message format module checks the types of all the blank parts between the two field formats in each message format. As a result, it extracts message formats that are fully filled with field formats classified by four types: SF(v), DF(v), DF, and GAP.

Algorithm 4 is the pseudo-algorithm of the additional field format extracting in each message format module. There are two thresholds for this module: Threshold1 is the variance of length, and Threshold2 is the highest length between two field formats. First, when each message format is input, it finds all message sequences that match this message format. Next, traversing all these message sequences, it finds all bytestreams that match the first blank part between two field formats in the message format. If the variance of lengths for these bytestreams is too high, the type of the blank part in the message format is GAP. i.e., if the variance is higher than Threshold1, the type of the blank part is GAP. If not, the blank part is a non-GAP field, so it checks the highest of length for these bytestreams. If the highest of the lengths is too high, i.e the highest of the lengths is higher than Threshold2, then the type of the blank part is DF. If not, the type of the blank part is SF(v) of DF(v), so it stores all the bytestreams as the set of values for the blank part. If the number of the

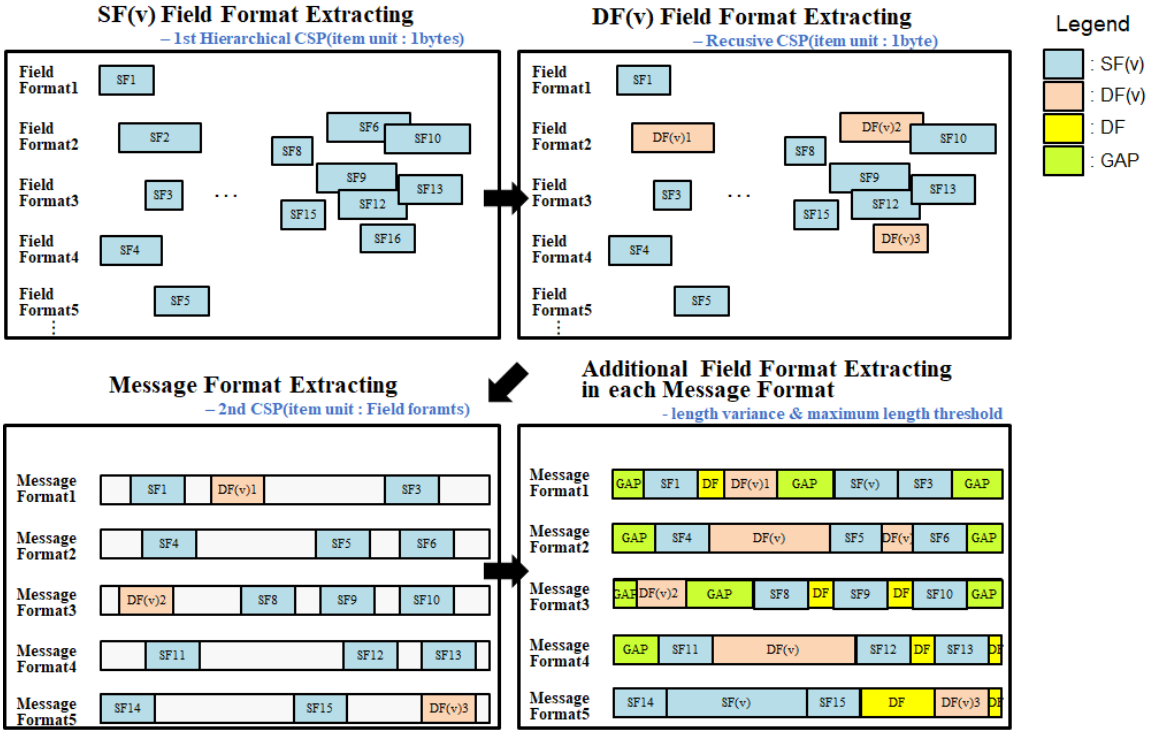

FIGURE 7. Overivew of the syntax inference stage. It shows the output changing shape of each module.

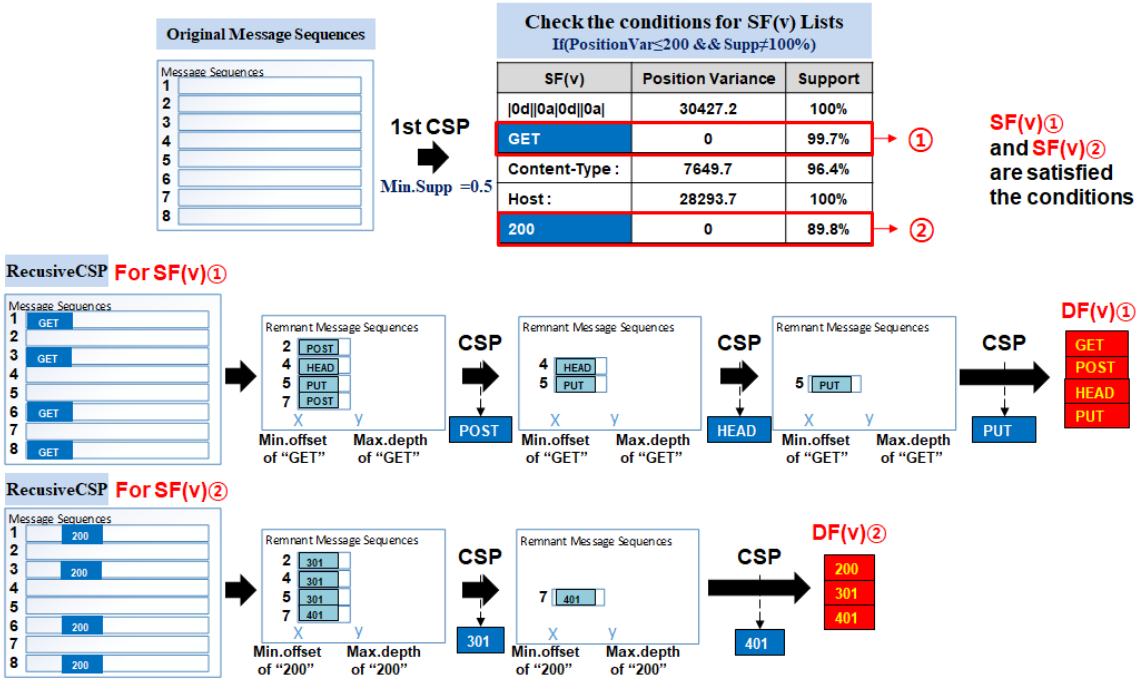

FIGURE 8. Process of recursive CSP.

set of values for the blank part is one, then the type of the blank part is SF(v). If the number of the set of value is larger than one, the type of the blank part is DF(v). After that, it performs this iteration process for all of the blank parts in the message formats. Fig. 9 shows the process of this module intuitively.

Algorithm 4 Additional Field Format Extracting Algorithm

Input: *MessageFormatSet, MessageSequenceSet*
Threshold$_1 = 5000$, *Threshold*$_2 = 25$
Output: *Intuitive and Clear MessageFormatSet*

```
01: foreach MsgF_i from MessageFormatSet do
02:   foreach FieldF_i from MsgF_i do
03:     foreach M_i whose M_i.Message_ID==
              each Message_ID of MsgF_i.
              FromMessage_IDSet do
04:       LengthList ≠ Φ;
05:       Blank = bytestream between FieldF_i.value
            and FieldF_{i+1}.value in M_i;
06:       LengthList = LengthList ∪ Blank.length;
07:     end
08:     if LengthList.variance ≥ Threshold_1 then
09:       FieldF_Temp.type = GAP;
10:     end
11:     else
12:       if LengthList.highest_length≥ Threshold_2 then
13:         FieldF_Temp.type = DF;
14:       end
15:       else
16:         foreach M_i whose M_i.Message_ID ==
              each Message_ID of MsgF_i.
              FromMessage_IDSet do
17:           FieldF_Temp.values= FieldF_Temp.values
              ∪ bytestream between FieldF_i.value and
              FieldF_{i+1}.value in M_i;
18:         end
19:         if sizeof(FieldF_Temp.values)==1 then
20:           FieldF_Temp.type = SF(v);
21:         end
22:         else
23:           FieldF_Temp.type = DF(v);
24:         end
25:       end
26:     end
27:     MsgF_i.insertNewFieldFtoBlankPart
        (FieldF_Temp);
28:   end
29: end
```

Algorithm 5 Semantics Inference Algorithm

Input: *MessageFormatSet*
Output: *MessageFormatSet with semantics*

```
01: foreach MsgF_i from MessageFormatSet do
02:   foreach FieldF_i from MsgF_i do
03:     if FieldF_i.type==DF(v) then
04:       FieldF_i.semantics =isMSG-Type(FieldF_i);
05:       FieldF_i.semantics =FieldF_i.semantics∪
                              isMSG-Len(FieldF_i);
06:       FieldF_i.semantics = FieldF_i.semantics∪
                              isHost-ID(FieldF_i);
07:       FieldF_i.semantics = FieldF_i.semantics∪
                              isSession-ID(FieldF_i);
08:       FieldF_i.semantics = FieldF_i.semantics∪
                              isTrans-ID(FieldF_i);
09:       FieldF_i.semantics = FieldF_i.semantics∪
                             isAccumulators(FieldF_i);
10:     end
11:   end
12: end
```

C. SEMANTICS INFERENCE

The semantics inference stage checks if there exist the field formats that correspond to the six predefined semantics types for all DF(v) in each message format. We use FieldHunter's semantics inference methodology [23], because it is the method that can extract the most specific types of semantics from among the previous semantics inference methodologies.

Algorithm 5 is the pseudo-algorithm of the semantics inference stage. The six predefined semantics types are MSG-Type, MSG-Len, Host-ID, Session-ID, Trans-ID, and Accumulators. The process of this stage is similar to the additional field format extracting module. When each message format is input, the algorithm traverses all the DF(v)s in the message formats, and performs the six heuristic algorithms to determine if the DF(v) corresponds to any of the six types.

1) MSG-TYPE

According to [23], a field format corresponding to MSG-Type is defined as the field format that satisfies the following conditions:

1) The values taken by this field format are neither extremely random nor constant.
2) This field format has a causal relationship with the bytestream that is in the opposite direction.

To check the first condition, it calculates the entropy $H(x) = -\sum p_i log_2 p_i$ metric. x is the value of the field format, and p_i is the probability of having the field format taking the value i. To check the second condition, it calculates the causality metric $= I(q; r)/H(q)$. q is the field format, and r is the bytestream that is in the opposite direction. If the entropy of the field format is low but above zero and the causality is high enough, the semantics type of the field format is MSG-Type.

2) MSG-LEN

A field format corresponding to MSG-Len is a field format whose values are the same as the lengths of the messages. Therefore, to calculate the correlation between the values of the field format and the lengths of the messages, it checks their Pearson correlation coefficients. If the Pearson correlation coefficient is high enough, the semantics type of the field format is MSG-Len.

3) HOST-ID

A field format corresponding to Host-ID is a field format whose values have a dependency on a source IP Address.

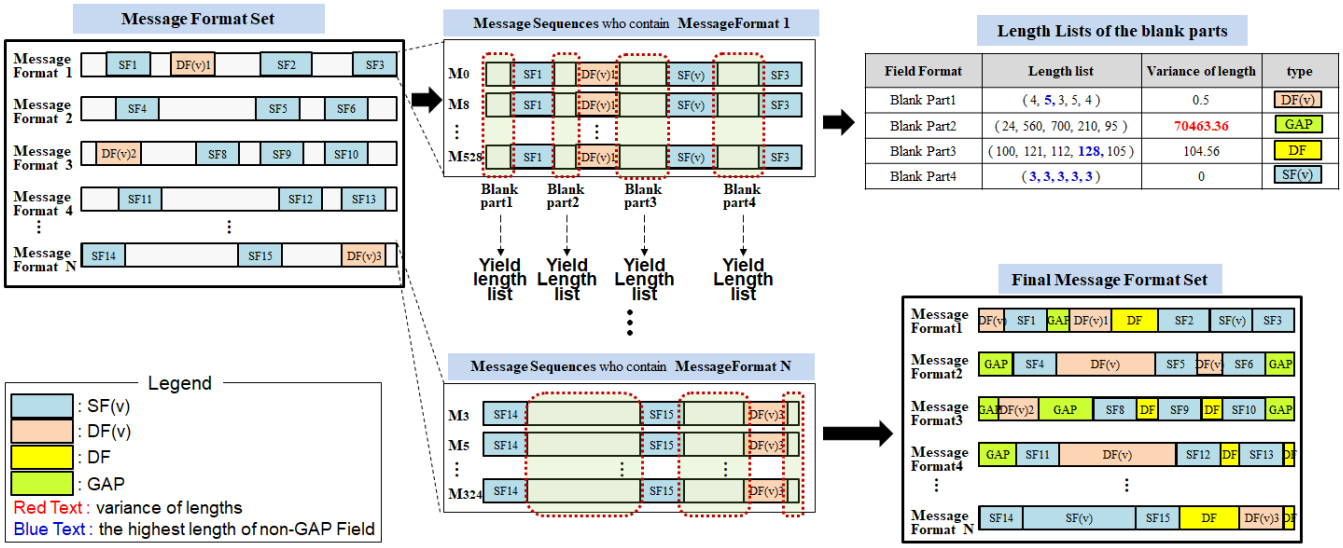

FIGURE 9. Process of additional field format extracting in each message format module.

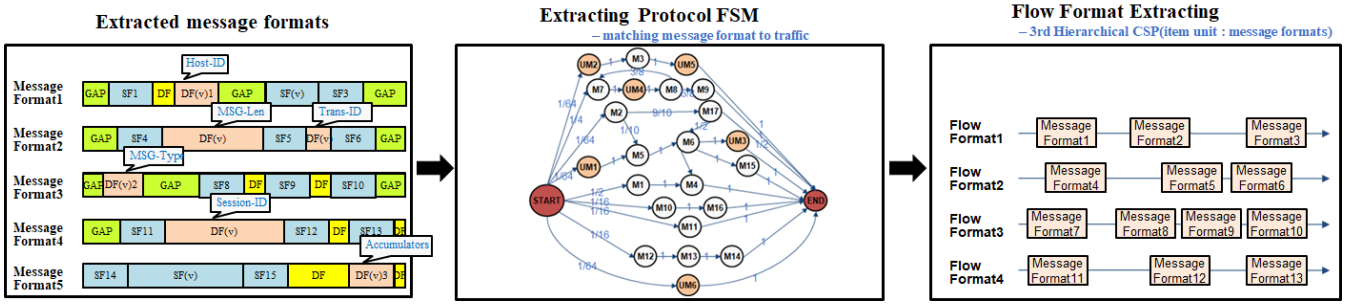

FIGURE 10. Overview of the behavior inference stage.

To check this dependency, it calculates the categorical metric $= R(x, y) = I(x; y)/H(x, y)$. x is the value of the field format, and y is the source IP address of the field format. If the categorical metric is high enough, the semantics type of the field format is Host-ID.

4) SESSION-ID

A field format corresponding to Session-ID is a field format whose values have a dependency on a session. To check this dependency, it calculates the categorical metric. x is the value of the field format, and y is the 5-tuple information of the field format. If the categorical metric is high enough, the semantics type of the field format is Session-ID.

5) TRANS-ID

A field format corresponding to Trans-ID is a field format whose values have a dependency on transaction. Transaction is a pair of request messages and response messages. Therefore, if the values of the field format are a transaction identifier, the bytestreams that are in the opposite direction must be the same as the values of the field format, and also, the values taken by this field format must be random. It calculates the entropy of the values taken by these field formats. If the entropy is high enough and most of the values of the field format are the same as the bytestreams in the opposite direction, the semantics type of the field format is Trans-ID.

6) ACCUMULATORS

To check if the field format corresponds to accumulators, the algorithm checks if there is a constant increment of the values taken by the field format.

D. BEHAVIOR INFERENCE

The behavior inference stage is composed of two modules: flow format extracting and protocol FSM extracting. Fig. 10 shows an overview of this stage.

The flow format extracting module, as described in Section 3, extracts flow formats by running the third CSP after setting the transaction unit to a flow sequence and

TABLE 4. Quantitative information on the traffic traces for verification.

Protocol	Flows	Packets	Bytes	Messages	
HTTP	359	3,841	467K	1,189	
				request	response
				598	591
DNS	2,170	4,349	740K	4,349	
				request	response
				2,337	2,012

setting the length-1 item unit to a message format. A flow format indicates the primary flow type of the protocol, so it can be helpful for understanding the structure of the unknown protocol. Moreover, it is helpful for minimizing the protocol FSM.

The protocol FSM extracting module extracts the protocol FSM whose states are set to each message format. It extracts transitions between two states by exploring input traffic traces in chronological order. This process becomes fairly easy by using *FlowSequenceSet*$''$, which is a set of F'' as described in Section 3. As shown in (17), each F'' has a list of message formats in chronological order that is contained in each F“ as an element. Finding the transitions, it counts the number of matchings for each transition. By using these counts of each transition matching, it calculates the probabilities of the transitions for each state. The extracted protocol FSM is very useful to replay packets for a network security area such as a honeypot system. Moreover, the FSM helps to know the order of occurrence of messages.

V. EXPERIMENT AND RESULT

In this section, we evaluate the efficacy of the proposed method in inferring the protocol specification of known protocols.

We implement the proposed method in a prototype system in C++ code on Linux. The system takes network capture files either in the libpcap or Netmon format as input. The system extracted four xml files describing the information

TABLE 5. Summary of experimental results for HTTP and DNS.

Protocol	HTTP	DNS
Field Format Info.	Total : 22 Request : 10 (SF(v) : 9 + DF(v) : 1) Response : 12 (SF(v) : 11 + DF(v) : 1)	Total : 13 Request : 3 (SF(v) : 3 + DF(v) : 0) Response : 10 (SF(v) : 9 + DF(v) : 1)
Message Format Info.	Total : 36 Request : 21 + Response : 15	Total : 27 Request : 5 + Response : 22
Flow Format Info.	1	0
FSM Info.	State : 17, Transition : 51	State : 12, Transition : 29
Correctness	1.0 (129/129)	1.0 (180/180)
Coverage	0.99(1188/1189)	1.0 (4349/4349)
Execution Time	11.65 s	85.89 s

ID	Direction	NumberOfFieldFormat	Support	Coverage	Structure
000	req	2	0.900	8.24%	[DF(v)] -> [GAP]
001	req	3	0.863	4.21%	[DF] -> [SF(v)] -> [GAP]
002	req	3	0.721	0.00%	[DF] -> [SF(v)] -> [GAP]
003	req	3	0.941	0.76%	[GAP] -> [SF(v)] -> [GAP]
004	req	3	0.719	0.00%	[GAP] -> [SF(v)] -> [GAP]
005	req	3	0.849	0.00%	[DF] -> [SF(v)] -> [GAP]
006	res	3	0.887	44.07%	[SF(v)] -> [DF(v)] -> [GAP]
007	res	3	0.702	0.00%	[GAP] -> [SF(v)] -> [GAP]
008	res	3	0.992	0.67%	[DF] -> [SF(v)] -> [GAP]
009	res	3	0.893	2.02%	[GAP] -> [SF(v)] -> [GAP]
010	res	3	0.704	0.00%	[GAP] -> [SF(v)] -> [GAP]
011	res	3	0.920	0.00%	[GAP] -> [SF(v)] -> [GAP]
012	res	3	0.858	0.08%	[GAP] -> [SF(v)] -> [GAP]
013	res	3	0.848	0.42%	[GAP] -> [SF(v)] -> [GAP]
014	res	3	0.787	0.00%	[GAP] -> [SF(v)] -> [GAP]
015	res	3	0.997	0.00%	[GAP] -> [SF(v)] -> [GAP]
016	req	4	0.736	6.31%	[DF(v)] -> [DF] -> [SF(v)] -> [GAP]
017	req	5	0.706	0.00%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
018	req	4	0.625	0.00%	[DF] -> [SF(v)] -> [SF(v)] -> [GAP]
019	req	5	0.707	0.00%	[GAP] -> [SF(v)] -> [GAP] -> [SF(v)] -> [GAP]
020	res	5	0.650	0.59%	[GAP] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
021	res	5	0.736	1.85%	[DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [GAP]
022	res	5	0.682	0.00%	[GAP] -> [SF(v)] -> [GAP] -> [SF(v)] -> [GAP]
023	req	6	0.610	4.29%	[DF(v)] -> [DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [GAP]
024	req	6	0.532	0.00%	[GAP] -> [SF(v)] -> [GAP] -> [SF(v)] -> [SF(v)] -> [GAP]
025	req	6	0.624	0.00%	[GAP] -> [SF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
026	res	7	0.638	0.00%	[GAP] -> [SF(v)] -> [GAP] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
027	req	7	0.525	0.17%	[DF(v)] -> [DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [SF(v)] -> [GAP]
028	req	8	0.525	0.00%	[GAP] -> [SF(v)] -> [GAP] -> [SF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
029	req	8	0.523	0.00%	[GAP] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [SF(v)] -> [GAP]
030	res	9	0.584	0.00%	[GAP] -> [SF(v)] -> [GAP] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP] -> [SF(v)] -> [GAP]
031	req	10	0.599	0.00%	[DF] -> [SF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
032	req	10	0.518	0.00%	[GAP] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
033	req	10	0.522	0.34%	[DF(v)] -> [DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [SF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
034	req	13	0.559	0.00%	[DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [SF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
035	req	14	0.515	25.90%	[DF(v)] -> [DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [SF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]

FIGURE 11. Structure of message formats for HTTP.

FID	Type	Direction	Value	Offset	Depth	Min_len	Max_len	Avg_len	Semantics
000	DF(v)	req	GET / POST DELET	0	4	5	5	-	
35:(0, 1)	DF	req		5	431	10	427	54	
004	SF(v)	req	HTTP/1.1|0d||0a|Host:	15	448	17	17	-	
35:(1, 2)	DF(v)	req	mail3.nate.com home.mail.nate.com mailimg.nate.com news.nateimg.co.kr main.nateimg.co.kr img.nate.com main2.nateimg.co.kr lithium.nate.com www.nate.com cyad1.nate.com shop1.daumcdn.net shop2.daumcdn.net adimg.nateimg.co.kr adimg.nate.com cyad.nate.com cas.criteo.com www.yebigun1.mil.kr mail1.dongwon.mil.kr cafe.naver.com lcs.naver.com cafe.nil.naver.com cc.naver.com	32	461	12	20	16	
008	SF(v)	req	|0d||0a|Connection: keep-alive	46	485	24	24	-	
005	SF(v)	req	|0d||0a|User-Agent: Mozilla/5.0 (Windows NT 10.0|3b| Win64|3b| x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181	70	600	115	115	-	
003	SF(v)	req	Safari/537.36|0d||0a|Accept:	185	624	24	24	-	
35:(4, 5)	DF	req		209	663	3	39	27	
006	SF(v)	req	|0d||0a|Referer: http://	212	681	18	18	-	
35:(5, 6)	DF(v)	req	home.mail.nate mailimg.nate www.nate mail3.nate cafe.naver	230	691	8	14	10	
002	SF(v)	req	.com/	240	696	5	5	-	
35:(6, 7)	DF	req		245	765	0	69	20	
009	SF(v)	req	|0d||0a|Accept-Encoding: gzip, deflate|0d||0a|Accept-Language: ko-KR,ko|3b|q=0.9,en-US|3b|q=0.8,en|3b|q=0.7|0d||0a|	245	853	88	88	-	
35:(7,-2)	GAP	req		-	-	-	-	-	

FIGURE 12. Representative message format for HTTP.

ID	Direction	NumberOfFieldFormat	Support	Coverage	Structure
000	req	3	1.000	14.92%	[DF(v)] -> [SF(v)] -> [DF]
001	req	2	0.989	0.00%	[DF] -> [SF(v)]
002	res	4	0.991	45.64%	[DF(v)] -> [DF(v)] -> [SF(v)] -> [GAP]
003	res	3	0.749	0.00%	[DF] -> [SF(v)] -> [GAP]
004	res	3	0.899	0.00%	[DF] -> [SF(v)] -> [GAP]
005	res	3	0.832	0.00%	[DF] -> [SF(v)] -> [GAP]
006	res	3	0.706	0.00%	[DF] -> [SF(v)] -> [GAP]
007	res	3	0.706	0.02%	[DF] -> [SF(v)] -> [GAP]
008	res	3	0.996	0.00%	[DF] -> [SF(v)] -> [GAP]
009	res	3	0.711	0.00%	[DF] -> [SF(v)] -> [GAP]
010	res	3	0.808	0.00%	[DF] -> [SF(v)] -> [GAP]
011	res	3	0.991	0.16%	[DF] -> [SF(v)] -> [GAP]
012	req	4	0.715	0.00%	[DF] -> [SF(v)] -> [DF] -> [SF(v)]
013	req	5	0.722	3.13%	[DF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF]
014	res	5	0.615	0.00%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
015	res	5	0.594	0.02%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
016	res	4	0.631	0.00%	[DF] -> [SF(v)] -> [SF(v)] -> [GAP]
017	res	5	0.620	0.00%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
018	res	5	0.621	0.00%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
019	res	5	0.708	0.00%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
020	res	5	0.781	0.00%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
021	res	5	0.633	0.11%	[DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [GAP]
022	req	6	0.664	35.69%	[DF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF] -> [SF(v)]
023	res	7	0.511	0.16%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF]
024	res	7	0.558	0.14%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
025	res	7	0.540	0.00%	[DF] -> [SF(v)] -> [DF(v)] -> [SF(v)] -> [DF] -> [SF(v)] -> [GAP]
026	res	9	0.503	0.00%	[DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF] -> [SF(v)] -> [DF]

FIGURE 13. Structure of message formats for DNS.

on field formats, message formats, flow formats, and protocol FSM and one png file showing the FSM in a directed pseudo-graph.

We first describe the dataset used for evaluating, and then explain the evaluation metrics. Lastly, we present the experimental result.

A. DATASET

We selected two well-known protocols for verification. One is HTTP, to verify for text protocol, and the other is DNS, to verify for binary protocol.

Table 4 presents the quantitative information on the traffic traces of these two protocols in flow, packet, and byte, along with the results of the message assemble stage. We collected these traffic traces from four different hosts at different times into pcap files using port-based classification method: port 80 for HTTP and port 53 for DNS. In the result of the message assemble stage, because HTTP is a TCP-based protocol, the flows were split using a series of consecutive packets with the same direction as the message unit, whereas each packet of DNS protocol became each message, because DNS is a UDP-based protocol.

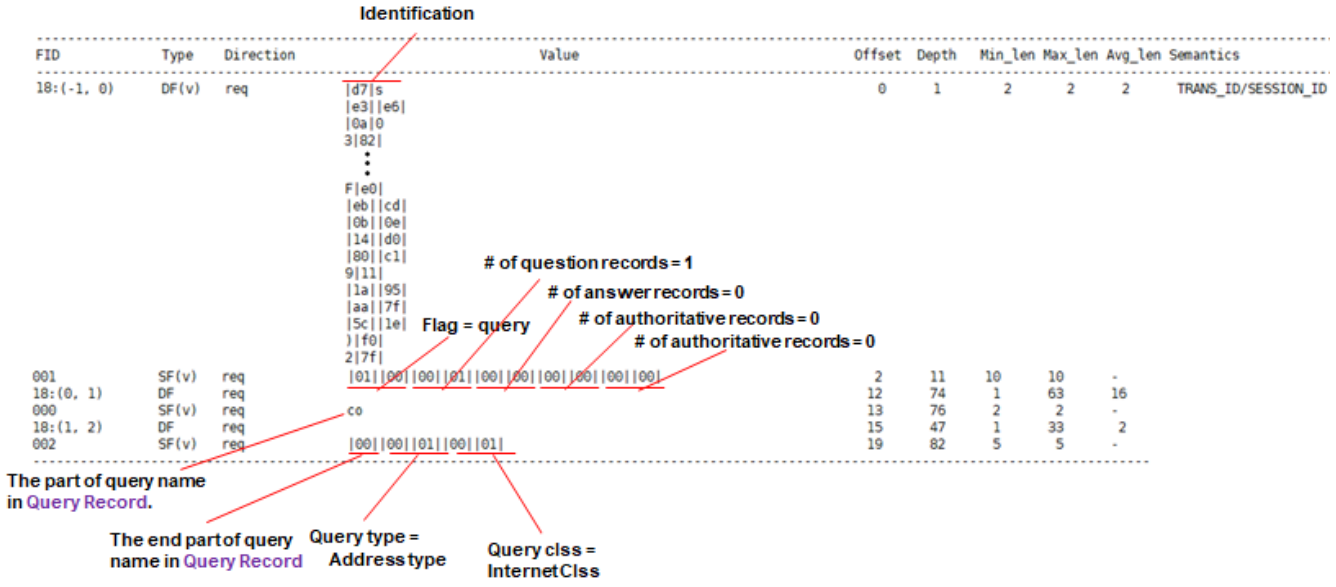

FIGURE 14. Representative message format for DNS.

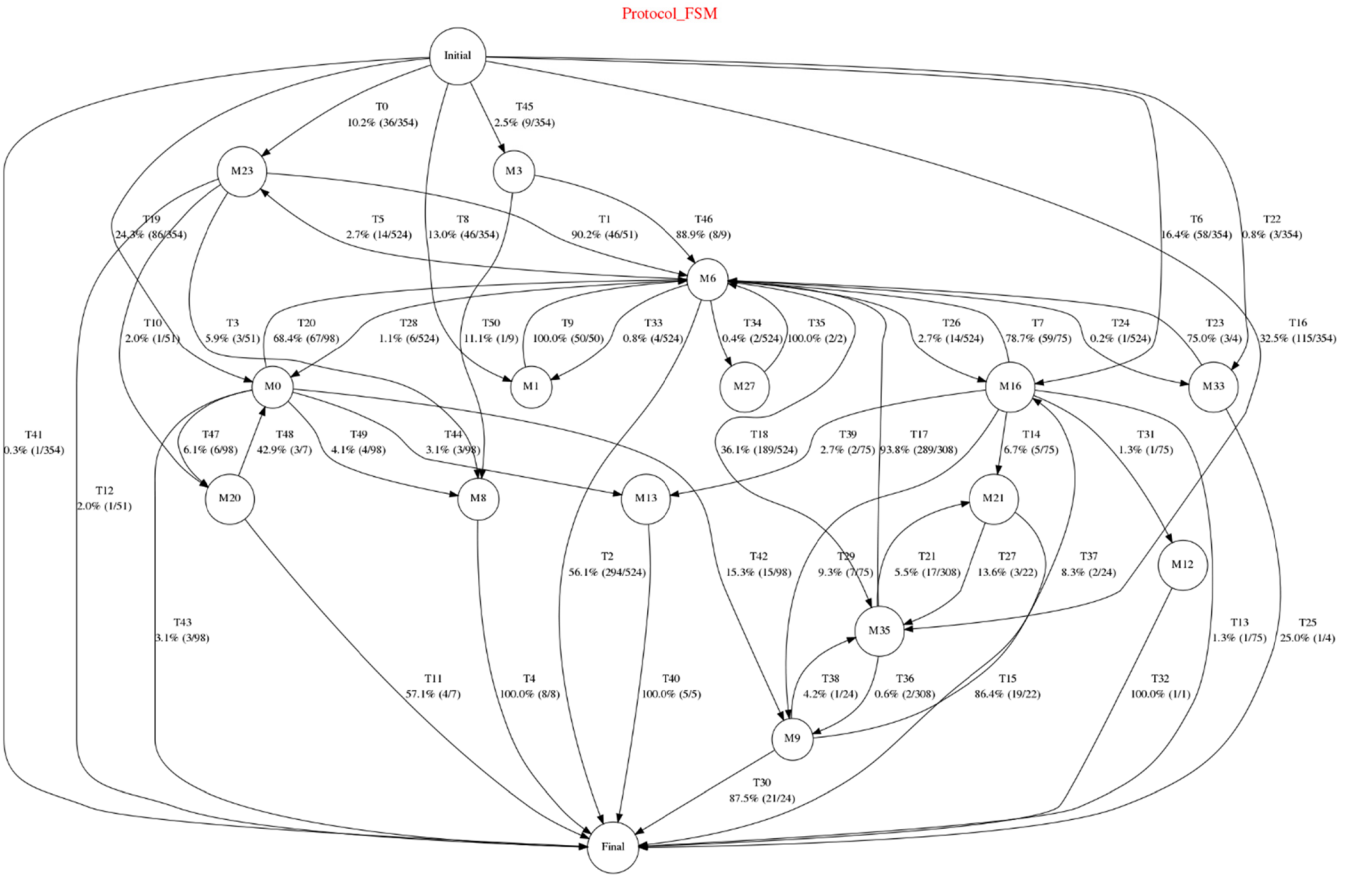

FIGURE 15. FSM for HTTP.

B. EVALUATION METRICS

The precondition for correct protocol semantics and a protocol FSM is that the protocol syntax is correct, so the main performance evaluation of protocol reverse engineering is whether the protocol syntax represents the correct message format. Hence, to evaluate the message formats, we use two

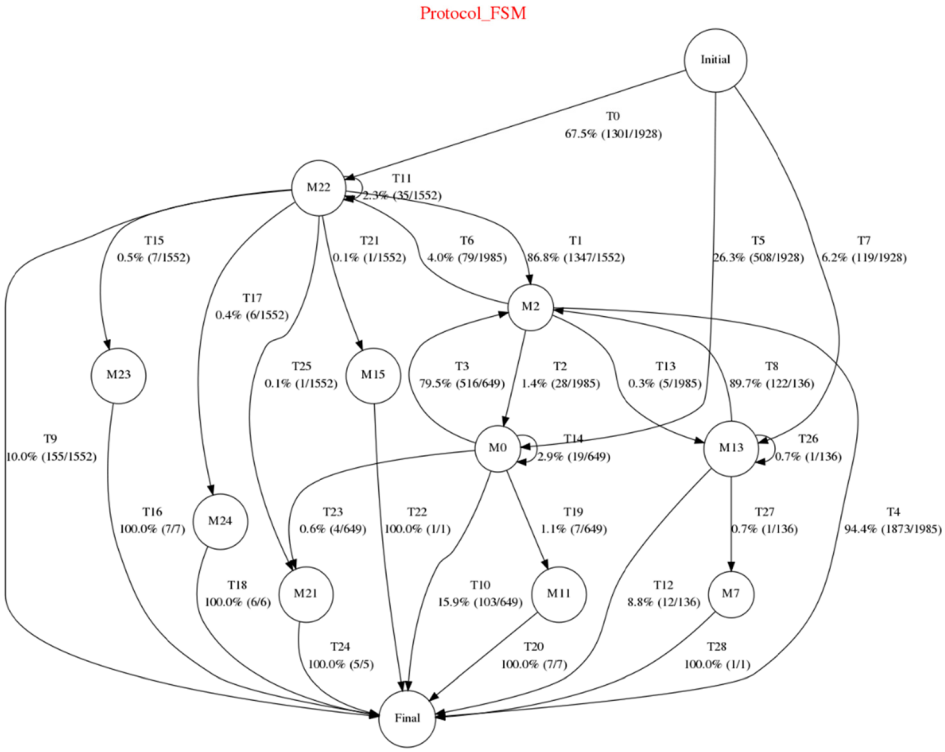

FIGURE 16. FSM for DNS.

standard metrics [18], correctness and coverage.

$$Correctness = \frac{\begin{array}{c}the\ number\ of\ true\\ message\ formats\ matched\ with\\ extracted\ message\ formats\end{array}}{\begin{array}{c}the\ number\ of\ total\\ true\ message\ formats\end{array}} \tag{20}$$

$$Coverage = \frac{\begin{array}{c}the\ number\ of\ messages\ matched\\ with\ extracted\ message\ formats\end{array}}{the\ number\ of\ total\ messages} \tag{21}$$

Correctness means how many of the true formats can be analyzed by the extracted message formats. Equation (20) shows how to calculate correctness of message formats. Coverage means how many of the messages can be analyzed by the extracted message formats. Equation (21) shows how to calculate coverage of message formats.

C. EXPERIMENTAL RESULT

Table 5 shows the summary of the experimental results. The proposed method extracts 22 field formats, 36 message formats, and 1 flow format for HTTP and 13 field formats, 27 message formats, and 3 flow formats for DNS. The coverage and the correctness of both two protocols are close to 1.0.

Fig. 11 shows the structures of message formats for HTTP. The proposed method compresses 1189 messages into 36 message formats, and each message format is appropriately subdivided into field formats without any blank parts.

Fig. 12 shows a representative message format as a sample among the whole set of HTTP message formats. This sample represents an HTTP request message format that is subdivided into 14 field formats. The field formats are in the order of Method [DF(v)] - URL [DF] - Version and Host [SF(v)] - value of Host field [DF(v)] - . . . - GAP. It reflects the mandatory components: method field, URL field, version field, and some optional HeaderName field of the HTTP request message format in non-GAP types. It provides information on minimum offset, maximum depth, minimum length, maximum length, average length, position variance, support, and semantics of all field formats that make up the message formats. This information indicates whether the field format has a fixed position, fixed length, static value, and so on.

Fig. 13 shows the structures of message formats for DNS. The proposed method compresses 4349 messages into 27 message formats, and each message format fully filled with SF(v), DF(v), DF, and GAP.

Fig. 14 shows a representative message format as a sample among the whole set of DNS message formats. This message format represents a DNS query message format that is subdivided into 6 field formats. The field formats are in the order of Transaction ID [DF(v)] – Flag and the number of each record [SF(v)] – a part of query name [DF] - a part of query name [SF(v)] - . . . – The end part of query name, query type, and query class [SF(v)]. In addition, the proposed method correctly found the semantics of Transaction ID for DNS as Trans-ID and Session-ID, as shown in Fig. 14. The reason that it found the semantics of Transaction ID as Session-ID

is that the values of Transaction ID are very random, and it changes every time the Session ID changes.

Fig. 15 and Fig. 16 shows FSMs of HTTP and DNS, respectively. Each state indicates each extracted message format, except for the initial state and the final state. Each path from the initial state to the final state refers to a flow type indicating the order in which the message formats are transmitted. Each edge of the FSM has a transition probability.

VI. CONCLUSION

In this paper, we proposed a novel protocol specification extraction method. We defined three types of formats, namely field format, message format, and flow format and four types of field formats to acquire clear protocol syntax. To extract such formats, we proposed hierarchical CSP and recursive CSP. The novelty of this method is that it extracts well-trimmed message formats and sufficiently compresses input messages into a small number of message formats so that it can determine the intuitive structure of an unknown protocol. As far as we know, no other papers have found the methods that can extract all of the protocol syntax, semantics, and FSM automatically.

The proposed method clearly has some limitations.

First, the proposed method cannot infer a specification of encrypted protocols such as SSH protocol. Encrypted protocols can be inferred only by using the application-based method. Considering fully automation and the difficulty of access to program binary, we adopted the network trace-based method.

Second, the quality of the inferred protocol specification is highly dependent on the amount of input messages. For extracting a more detailed and abundant protocol specification, it is important to collect large amounts of data in various environments.

Third, most of the existing protocol reverse engineering methods including the proposed method only focused on inferring application layer protocols. In consideration of various network environments, a method for covering all layers of the OSI seven model should be studied in the future.

Despite the fact that there are the limitations above mentioned, we believe our work could be a springboard for addressing these limitations and have solved many existing problems mentioned in Section II. In a future work, we will apply our method to several other protocols and further improve our method through experiments. Additionally, we plan to upgrade the method and make it capable of extracting a specification of protocol for all 7 OSI layers.

REFERENCES

[1] C. Yu, J. Lan, Z. Guo, Y. Hu, and T. Baker, "An adaptive and lightweight update mechanism for SDN," *IEEE Access*, vol. 7, pp. 12914–12927, Jan. 2019.

[2] K. Kifayat, T. Baker, M. Mackay, M. Merabti, and Q. Shi, *Real Time Risk Management in Cloud Computation*. Hershey, PA, USA: IGI Global, 2011.

[3] S. Al-Sharif, F. lqbal, T. Baker, and A. Khattack, "White-hat hacking framework for promoting security awareness," in *Proc. 8th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*, Larnaca, Cyprus, Nov. 2016, pp. 1–6.

[4] Sophos. *The Dirty Secrets of Network Firewalls*. [Online]. Available: https://www.sophos.com/en-us/medialibrary/Gated-Assets/white-papers/firewall-dirty-secrets-report.pdf

[5] Y. Chang, S. Choi, J.-H. Yun, and S. Kim, "One step more: Automatic ICS protocol field analysis," in *Proc. Int. Conf. Critical Inf. Infrastruct. Secur.* Cham, Switzerland: Springer, Oct. 2017, pp. 241–252.

[6] J. Duchene, C. L. Guernic, E. Alata, V. Nicomette, and M. Kaaniche, "State of the art of network protocol reverse engineering tools," *J. Comput. Virol. Hacking Techn.*, vol. 14, no. 1, pp. 53–68, Feb. 2018.

[7] B. D. Sija, Y.-H. Goo, K.-S. Shim, H. Hasanova, and M.-S. Kim, "A survey of automatic protocol reverse engineering approaches, methods, and tools on the inputs and outputs view," *Secur. Commun. Netw.*, vol. 2018, Feb. 2018, Art. no. 8370341. [Online]. Available: https://www.hindawi.com/journals/scn/2018/8370341/

[8] A. Tridgell. (Aug. 2003). *How Samba Was Written*. [Online]. Available: http://samba.org/ftp/tridge/misc/french_cafe.txt

[9] Pidgin. (2018). *About Pidgin*. [Online]. Available: http://www.pidgin.im/about

[10] J. Caballero and D. Song, "Automatic protocol reverse-engineering: Message format extraction and field semantics inference," *Int. J. Comput. Telecommun. Netw.*, vol. 57, no. 2, pp. 451–474, Feb. 2013.

[11] M. Liu, C. Jia, L. Liu, and Z. Wang, "Extracting sent message formats from executables using backward slicing," *Proc. 4th Int. Conf. Emerg. Intell. Data Web Technol.*, X'ian, China, Sep. 2013, pp. 377–384.

[12] Y. Wang *et al.*, "A semantics aware approach to automated reverse engineering unknown protocols," in *Proc. 20th IEEE Int. Conf. Netw. Protocols (ICNP)*, Oct. 2012, pp. 1–10.

[13] T. Krueger, H. Gascon, N. Kramer, and K. Rieck, "Learning stateful models for network honeypots," in *Proc. 5th ACM Workshop Secur. Artif. Intell.*, Raleigh, NC, USA, Oct. 2012, pp. 37–48.

[14] H. Li, B. Shuai, J. Wang, and C. Tang, "Protocol reverse engineering using LDA and association analysis," in *Proc. 11th Int. Conf. Comput. Intell. Secur. (CIS)*, Dec. 2015, pp. 312–316.

[15] M. A. Beddoe. (2004). *Network Protocol Analysis Using Bioinforomatics Algorithms*. [Online]. Available: http://www.4tphi.net/~awalters/PI/pi.pdf

[16] C. Leita, K. Mermoud, and M. Dacier, "ScriptGen: An automated script generation tool for Honeyd," in *Proc. 21st Annu. Comput. Secur. Appl. Conf.*, Tucson, AZ, USA, Dec. 2005, p. 2.

[17] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proc. 16th USENIX Secur. Symp.*, Boston, MA, USA, Aug. 2007, pp. 199–212.

[18] G. Bossert, "Exploiting semantic for the automatic reverse engineering of communication protocols," Ph.D. dissertation, Univ. Gif-sur-Yvette, Rennes, France, Dec. 2014.

[19] L. Wang and T. Jiang, "On the complexity of multiple sequence alignment," *J. Comput. Biol.*, vol. 1, no. 4, pp. 337–348, 1994.

[20] J.-Z. Luo and S.-Z. Yu, "Position-based automatic reverse engineering of network protocols," *J. Netw. Comput. Appl.*, vol. 36, no. 3, pp. 1070–1077, May 2013.

[21] Y. Wang, N. Zhang, Y.-M. Wu, B.-B. Su, and Y.-J. Liao, "Protocol formats reverse engineering based on association rules in wireless environment," in *Proc. 12th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Melbourne, VIC, Australia, Jul. 2013, pp. 134–141.

[22] R. Ji, H. Li, and C. Tang, "Extracting keywords of UAVs wireless communication protocols based on association rules learning," in *Proc. 12th IEEE Int. Conf. Comput. Intell. Secur.*, Wuxi, China, Dec. 2016, pp. 310–313.

[23] I. Bermudez, A. Tongaonkar, M. Iliofotou, M. Mellia, and M. M. Munafo, "Automatic protocol field inference for deeper protocol understanding," in *Proc. 14th IFIP Netw. Conf.*, Toulouse, France, May 2015, pp. 1–9.

[24] G. Ladi, L. Buttyan, and T. Holczer, "Message format and field semantics inference for binary protocols using recorded network traffic," in *Proc. 26th Int. Conf. Softw., Telecommun. Comput. Netw.*, Split, Croatia, Sep. 2018.

[25] R. Agarwal, T. Imielinski, and A. Awami, "Mining association rules between sets of items in large datagases," in *Proc. ACM SIGMOD Conf. Manage. Data*, Washington, DC, USA, May 1993, pp. 207–216.

[26] R. Agarwal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. 20th VLDB Conf.*, Santiago, Chile, Sep. 1994, pp. 487–499.

[27] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proc. IEEE Int. Conf. Data Eng.*, Mar. 1995, pp. 3–14.

[28] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Proc. 5th Int. Conf. Extending Databased Technol. Adv. Database Technol.* Springer, Mar. 1996, pp. 1–17.

[29] J. S. Park, M.-S. Chen, and P. S. Yu, "Using a hash-based method with transaction trimming for mining association rules," *IEEE Trans. Knowl. Data Eng.*, vol. 9, no. 5, pp. 813–825, Sep. 1997.
[30] K. Vyas and S. Sherasiya, "Modified Apriori algorithm using hash based technique," *Int. J. Adv. Res. Innov. Ideas Edu.*, vol. 2, no. 3, pp. 1229–1234, 2016.
[31] K.-S. Shim, J.-H. Ham, B. D. Sija, and M.-S. Kim, "Application traffic classification using payload size sequence signature," *Int. J. Netw. Manage.*, vol. 27, no. 4, pp. 1–17, Apr. 2017.

YOUNG-HOON GOO was born in Cheonan, South Korea, in 1991. He received the B.S. degree in computer and information science from Korea University, South Korea, in 2016, where he is currently pursuing the Ph.D. degree (integrated program). His research interests include the Internet traffic classification, the Internet security, and network management.

KYU-SEOK SHIM was born in Seoul, South Korea, in 1989. He received the B.S. and M.S. degrees in computer and information science from Korea University, South Korea, in 2014 and 2016, respectively, where he is currently pursuing the Ph.D. degree.

Since 2019, he has been a Research Coordinator with the Korea University Research and Business Foundation, South Korea. His research interests include the Internet traffic classification and network management.

MIN-SEOB LEE was born in Yeosu, South Korea, in 1993. He received the B.S. degree in computer and information science from Korea University, South Korea, in 2018, where he is currently pursuing the M.S. degree. His research interests include the Internet traffic classification, the Internet security, and network management.

MYUNG-SUP KIM was born in Gyeongju, South Korea, in 1972. He received the B.S., M.S., and Ph.D. degrees in computer science and engineering from POSTECH, South Korea, in 1998, 2000, and 2004, respectively.

From 2004 to 2006, he was Postdoctoral Fellow with the Department of Electrical and Computer Engineering, University of Toronto, Canada. Since 2006, he has been a Full Professor with the Department of Computer Convergence Software, Korea University, South Korea. His research interests include the Internet traffic monitoring and analysis, service and network management, the future Internet, and the Internet security.