Author: Jiang, Jiaojiao; Versteeg, Steve; Han, Jun; Hossain, M. D. Arafat; Schneider, Jean-Guy
Title: A positional keyword-based approach to inferring fine-grained message formats
Year: 2019
Journal: Future Generation Computer Systems
Volume: 102
Issue: January
Pages: 369-381
URL: http://hdl.handle.net/1959.3/451304

Copyright: Copyright © 2019 Elsevier B.V. All rights reserved. Per publisher policy, the author’s final accepted manuscript is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. https://creativecommons.org/licenses/by-nc-nd/4.0/

This is the author’s version of the work, posted here with the permission of the publisher for your personal use. No further distribution is permitted. You may also be able to access the published version from your library.

The definitive version is available at: https://doi.org/10.1016/j.future.2019.08.011
A Positional Keyword-based Approach to Inferring Fine-grained Message Formats*

Jiaojiao Jiang*a, Steve Versteega, Jun Han*a, MD Arafat Hossain*a, Jean-Guy Schneiderb

*a School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia
b School of Information Technology, Deakin University, Burwood, VIC 3125, Australia

Abstract

Message format extraction, the process of revealing the message syntax without access to the protocol specification, is important for a variety of applications such as service virtualization and network security. In this paper, we propose P-token, which mines fine-grained message formats from network traces. The novelty of our approach is twofold: a ‘positional keyword’ identification technique and a two-level hierarchical clustering strategy. Positional keywords are based on the insight that keywords or reserved words usually occur at relatively fixed positions in the messages. By associating positions as meta-information with keywords, we can more accurately distinguish keywords from message payload data. After identification, the positional keywords are used as features to cluster the messages using density peaks clustering. We then perform another level of clustering to refine the clusters with low homogeneity. Finally, the message format of each cluster is extracted based on the observed ordering of keywords. P-token improves on the current state-of-the-art techniques by successfully addressing two challenges that commonly afflict existing keyword based format extraction methods: (1) message keyword mis-identification and (2) message format over-generalization. We have conducted experiments on services and applications using various protocols, including SOAP, LDAP, IMS and a RESTful service. Our experimental results show that P-token outperforms existing methods in extracting message formats.

Keywords: Protocol message formats, positional keyword, two-level clustering.

1. Introduction

The automatic inference of protocol message formats from raw network traces is an important problem with widespread applications, such as in the domains of service virtualization and network security. Service virtualization is a critical technology for DevOps and Continuous Delivery[1], to enable automated testing in production-like conditions of software systems against their

*corresponding author
dependent systems. Service virtualization [2, 3, 4] requires the understanding of message formats in order to decode request messages and formulate appropriate response messages. In the domain of network security [5, 6, 7], intrusion detection systems (IDS) and firewall systems require the knowledge of protocol message formats before performing deep packet inspection. For both service virtualization and network security, however, the message formats required are not always available. This situation may arise in scenarios involving legacy systems, proprietary protocols or just poor documentation [8]. This illustrates the importance of automated inference of message formats used in various system applications.

Over the past few years, researchers have proposed many methods for protocol message format inference. These methods broadly fall into two categories: (1) those based on reverse-engineering, which extract protocol message formats through reverse engineering the executable code of a software application that implements a given protocol, and (2) those based on analyzing network traces, which extract protocol message formats through analyzing raw network messages of a given protocol. As discussed in [9], reverse engineering protocols typically involves manual effort. To address this problem, many methods for automatic protocol reverse engineering have been proposed. Example methods automating this process include Polyglot [10], Prospex [11], HFSM [12], and AUTOGRAM [13]. A common drawback of these approaches is that they become inapplicable when the executable code of the application is not accessible. With the trends toward cloud computing, Software-as-a-Service and container technology, getting access to the executable application code is becoming less common. In this paper, we focus on extracting protocol message formats by analyzing raw network traces.

Methods based on network trace analysis utilize statistical learning techniques from Frequent Pattern Mining [14] and Natural Language Processing [15] to mine keywords patterns from raw message traces so as to group messages into clusters reflecting their types and consequently infer the message format of each type. Examples of these methods include Discoverer [16], SPI [17], SANTA-
Class [18], AutoReEngine [19], and ProDecoder [20]. However, a challenge faced by these methods is how to reliably discerning which terms are keywords and which terms are part of message “payload”. For example, AutoReEngine extracts message keywords by splitting messages into n-grams of different lengths, and frequent n-grams are treated as keywords. In general, keyword identification in existing methods suffers a number of issues. First, a sub-string or super-string of a keyword may be wrongly identified as a keyword, i.e., keyword under-fitting or keyword over-fitting. Second, certain keyword occurrences may be wrongly treated as payload information, while other occurrences of keyword strings in payload are wrongly identified as keyword occurrences, i.e., mis-treatment of keyword occurrence. These keyword imprecision and mis-treatment issues are some of the main reasons of message mis-clustering, i.e., causing different types of messages being put into one cluster. Another reason causing mis-clustering is the imbalance between different types of messages in the message traces as clustering is generally based on the frequencies of keyword occurrences. Message mis-clustering leads to over-generalization of the derived message formats, resulting in coarse-grained message formats that accept ill-formed messages.

To address the above issues, we propose P-token, a new approach to extract message formats from raw protocol messages. It takes advantage of the properties of protocol messages, particularly those that arise from their template structure, in identifying message keywords. Machine-generated messages are formulated by a computer process or application, according to particular message templates or formats. P-token leverages the positions of keywords in the message template structure to differentiate a keyword occurrence from its string value’s occurrences in message payload and therefore identify message keywords more accurately. P-token has three major steps with corresponding techniques: (1) identifying positional keywords, i.e., keywords together with their positions in the messages, (2) clustering the messages into groups with a two-level hierarchical clustering strategy, such that each group has high homogeneity (representing a particular type of messages), and (3) inferring the message format for each cluster based on the natural positions of keywords in
messages. In Step (1), we introduce a new technique called *positional token*. By associating the position as meta-data with each token, we can more accurately discern which tokens are keywords of the message and which tokens are in the message payload. Hence, positional token can address the aforementioned keyword imprecision and mis-treatment issues faced by existing methods. In Step (2), we present a new *two-level clustering* technique, based on the extracted positional keywords. After initial clustering, it identifies the clusters that contain messages of different types, and performs a further level of clustering. It, therefore, addresses the mis-clustering and message format over-generalization issue faced by existing methods. In Step (3), the natural positions of keywords in the messages are used to extract the fine-grained format for each cluster. Compared to existing methods, which generate message formats by aligning multiple messages or mining keywords patterns [3], our approach produces more accurate representations of the protocol message formats.

With P-token, we make the following key contributions in inferring protocol message formats:

- a new positional keyword identification method, which addresses the keyword imprecision and mis-treatment issues;
- a two-level clustering strategy to separate the messages into clusters with high homogeneity, which addresses the mis-clustering and format over-generalization issue;
- a new method to derive fine-grained message formats based on the natural positions of the positional keywords, with high accuracy.

To present the effectiveness of P-token, we compare it with two state-of-the-art approaches (ProDecoder [20] and AutoReEngine [19]), and two baseline approaches (“vanilla” token and P-token without second level clustering). Experiments are conducted on real-world software applications using various protocols, including LDAP, SOAP, and IMS and a RESTful service. Our experimental results show that P-token achieves more accurate message formats.
than existing methods.

The rest of the paper proceeds as follows. We analyze the problem of extracting protocol message formats by using a real-world example in Section 2. We give the rationale of P-token in Section 3. We present the detailed techniques involved in P-token in Section 4. Experimental results on real-world protocol traces are reported in Section 5. We discuss related work in Section 7 and conclude this paper in Section 8.

2. Problem Statement

A communication protocol defines the format or structure of messages that the system or service sends and receives. In this paper, we deal with services and applications with tokenized messaging protocols, i.e., the communication messages are tokenized based on delimiters. A message format can be defined as a sequence of message fields (see Figure 1). The values of some fields are fixed in the messages of the same type, but the values of other fields vary across messages. In the context of this paper, we call the fields with variable values as payload fields, and call the fields with fixed values as keyword fields that present special meanings defined by the underlying message formats. Here, we assume that a protocol has fixed message formats where keywords in messages of a given type occur in a fixed order (even though some keywords can be optional or repetitive). Figure 2 shows some example LDAP messages, which contain keywords such as “cn” and “ou” representing an individual’s common name and an organizational unit, respectively. In this paper, a message keyword is defined a token comprising the message format for a given message type. By this definition, one keyword field may contain one or more keywords. For example,

---

2Even for protocols with free-format like HTTP, they usually have a fixed format when being used for a specific service or application so that the service or application can interpret the messages received in a simple and deterministic manner. As such, in practice, the assumption of fixed-format protocols is still applicable for free-format protocols as long as they have a fixed message format implementation for a specific service or application.
the first field “LDAP Bind Request Message ID:” in the first message in Figure 2 contains the keyword tokens “LDAP”, “Bind”, “Request”, “Message”, “ID”, and “:”. The second field contains the payload “1”.

Protocol message format extraction aims to automatically extract the message format for each message type. Many heuristic methods have been developed to first identify message keywords and then cluster the messages into type-specific groups according to keyword co-occurrence patterns across the messages. The accuracy of message formats extracted directly relies on the accuracy of keyword identification and message clustering. In this regard, existing methods has a number of limitations. First (keyword imprecision: under-fitting and over-fitting), existing methods generally rely on analysis of messages (as strings) in identifying keywords. A sub-string or super-string of a keyword may present the same or greater frequency than the actual keyword, and may be mis-identified as keywords by existing methods. They are referred to as keyword under-fitting and over-fitting respectively, or keyword imprecision in general. For example, the sub-string “ind” (shared by “Bind” and “Unbind”) may be falsely identified as a keyword as it has a higher frequency than “Bind”
or “Unbind”. Similarly, the super-string “ou=S” (associated with the keyword “ou”) has frequency 2 in Figure 2, which may be extracted as a keyword.

Second (mis-treatment of keyword occurrences), a keyword may appear multiple times in a message for different purposes. For example, “ou” is a keyword that appears twice in each message in Figure 2, which captures the hierarchical organizational unit information. However, the repetitive occurrences of a keyword in a message are retrieved as a single occurrence in some existing methods (e.g., only the first occurrence is considered as a keyword). On the other hand, some payload data may contain strings which coincidentally have the same values as keywords. For example, “dn” is a keyword in the messages in Figure 2, but it is also part of the payload in the fourth message where it is not a keyword. However, some existing methods also wrongly treat such occurrences in payload of a keyword string as keyword occurrences. In general, both of the above cases of mis-treating keyword occurrences are due to the fact that the keyword positions are not considered by the existing methods.

Third (mis-clustering), the message clustering step relies on the keywords identified and their occurrences. The identification of wrong keywords and the mis-treatment of keyword occurrences impact the clustering results. Furthermore, the imbalance between different types of messages in the message traces also impact the clustering results because clustering generally depends on the frequencies of keyword occurrences. These issues may cause different types of messages being mixed in one cluster, which eventually leads to an over-generalized format for the messages in the cluster. For example, if a cluster mixes the LDAP Add messages and Modify messages in Figure 2, it will generate an over-generalized format: LDAP • Request Message ID: • LDAP • Request Protocol Op dn: cn=.*,ou=.*,ou=.*,o=DEMOCORP,c=AU.*. It is coarse-grained that both Add and Modify messages and possibly other (ill-formed) messages can all be accepted by this format.

The above limitations lead us to design a more effective method that can accurately identify message keywords and their occurrences, and accurately differentiate different types of messages so as to infer fine-grained message formats.
The proposed approach, P-token, addresses the keyword imprecision and mis-treatment issues by a new method called *positional token*, and deals with the *mis-clustering* issue with a new *two-level hierarchical clustering* method.

### 3. Rationale

P-token aims at extracting fine-grained message formats based on the two key insights: (1) **keyword position sensitivity**, i.e., message keywords appear in relatively fixed positions across messages of the same type, and (2) **intra-cluster homogeneity**, i.e., the messages of the same type present high homogeneity in terms of their positional keyword-based structure while messages from different types present high dissimilarity. Meanwhile, as we focus on tokenized messaging protocols, messages can be analyzed in terms of tokens (after tokenization), rather than in terms of bits in untokenized protocols. Therefore, the *keyword imprecision* (under-fitting and over-fitting) issues are avoided.

**Keyword position sensitivity.** As we discussed above, protocol messages are machine-generated messages. An essential feature of machine-generated messages is their *template structure*, where the messages are structured by particular templates in the form of alternate *keyword(s) fields* and *payload fields* as shown in Figure 1. For the same type of messages, the message keywords

---

Figure 3: Distribution of keywords’ positions in the LDAP messages used in this paper.
present at relatively fixed positions in the messages, and therefore have high occurrence frequencies at these positions. Consequently, we consider message tokens at high frequency positions as keyword occurrences, and tokens at low frequency positions as non-keyword occurrences. Note that the same message token may appear at different locations with different frequencies, and therefore be considered as keyword occurrences or non-keyword occurrences depending on their frequency at the position concerned. Figure 3 shows the distribution of keywords’ occurrences over positions in the LDAP messages used as examples in this paper (the position starts from 0). Here, x-axis denotes keyword tokens, y-axis denotes the positions of their appearance, and the color bar indicates the frequency scale. As we can see, the keywords appear with high frequencies at relatively fixed positions. The keyword “Add”, for instance, appears at two fixed positions 1 and 8. The keyword “Bind” appears at three fixed positions 1, 8 and 16, and it also appears frequently around position 46. The keyword “c” frequently appears around positions 22, 31 and 36. It also appears at positions 19, 27 and 66 but with lower frequencies, indicating that these occurrences most likely are false keywords in payload fields. P-token uses this key insight of keywords’ position sensitivity to differentiate keyword occurrences from non-keyword occurrences and extract message keywords.

Intra-cluster homogeneity. The mis-identification of message keywords often leads to the mis-clustering of messages, which in turn leads to over generalization of the message formats. If a cluster has messages from different types, it is expected that the cluster will have a low intra-cluster homogeneity, i.e., the similarity among messages in the cluster is lower than otherwise. In this paper, P-token adapts the measure introduced in [21] to compute the homogeneity of messages within a cluster, which will be given later in Eq. (14). For example, if the LDAP Add messages and the Modify messages in Figure 2 are correctly grouped into two separate clusters, their homogeneity values are 3.70 and 5.88, respectively. If they are mixed in one cluster, its homogeneity is 1.82, which is much lower than the homogeneity values when the messages are correctly clustered. P-token uses this key insight to identify mixed message
clusters. For a mixed cluster, P-token performs another level of clustering to
group the messages into sub-clusters specific to message types.

The above key insights are also the fundamental assumptions of our proposed
approach. More specifically, we assume that message keywords appear in rela-
tively fixed positions across messages of the same type and the messages of the
same type present high homogeneity while message from different types present
high dissimilarity. In the next section, we introduce our proposed approach
based on these two key insights.

4. The Proposed Approach: P-token

In this section, we present the details of the proposed approach, P-token,
aimed at obtaining fine-grained message formats from message traces of un-
known protocols. The key novelty of P-token lies in its ability to (1) accurately
identify message keywords and (2) accurately group the messages into clusters
reflecting their types.

4.1. Overview

P-token involves three steps: Keyword Identification, Message Clustering,
and Format Extraction (see the architecture in Figure 4). In the first step
(Keyword Identification) of P-token, we extract keywords from raw protocol
messages, which are the constant parts of messages of the same type and appear
around relatively fixed positions in the messages. We first tokenize each message
and associate each token with its starting position in the message, and select the tokens that have greater frequencies across messages as candidate positional keywords. We then analyze the concentration of candidate keywords around positions in messages so as to differentiate true keywords from false keywords. The ones in a high-density position window are considered as true keywords, while those in a low-density position window are primarily parts of payload fields and are considered as false keywords. In the second step (Message Clustering), we cluster messages into different groups reflecting their types, through identifying the consistent co-occurrence of positional keywords from message to message. For clusters with low message homogeneity (i.e., containing different types), we perform another level of clustering by re-grouping the messages in the sub-clusters so that each cluster contains messages of the same type. In the last step (Format Extraction), we first convert the messages in each cluster to token sequences of alternating keyword and payload tokens, based on the natural positions of keywords in the messages. Then, we infer a common keyword(s)-payload sequence, and finally we convert the sequence to a regular expression as the message format for the cluster.

4.2. Keyword Identification (Step 1)

In this step, we identify message keywords according to the following two criteria: (i) keywords are common to messages of the same type, and (ii) they appear at relatively fixed positions in the messages. It involves two sub-steps: (1) extracting candidate positional keywords through frequency analysis of tokens and (2) determining the true keywords through studying the occurrence and recurrences of candidate positional keywords.

![Figure 5: An example of positional tokens.](image)
4.2.1. Candidate Positional Keywords Identification

In this sub-step, each message in a given message trace is split into tokens using tokenization (lines 4–7 in Algorithm 1), a well developed method in Natural Language Processing. It is the process of breaking an input text into words, terms, symbols, or some other meaningful elements called tokens, based on separators and/or delimiters in the input text. For example, the first message in Figure 2 can be broken down to tokens: “LDAP”, “Bind”, “Request”, “Message”, “ID”, etc. For the purpose of differentiation in this paper, we call the tokens (without any position information) as vanilla tokens. In this paper, we are particularly interested in the position for each token. Then, the first message can be broken into the following positional tokens (i.e., tokens with their sequential token numbers): “LDAP(0)”, “Bind(1)”, “Request(2)”, “Message(3)”, “ID(4)”, etc (see Figure 5). Formally, given a set $M$ of protocol messages, we use $P_m = \{t_m^0, t_m^1, \ldots, t_m^{l_m-1}\}$ to denote the consecutive positional tokens in $m$-th message, where $l_m$ is the number of tokens in the message, and $t_m^i$ is the positional token in message $m$ at position $i$. If message ID is not specified, we use $t(i)$ to denote a positional token at position $i$; we use $t$ to denote a vanilla token.

After generating positional tokens, we identify candidate positional keywords based on the number of messages in which a vanilla token appears, rather than the number of all occurrences of a vanilla token in all messages (lines 8–14). This consideration emphasizes the importance of a token’s appearance across different messages, and de-emphasizes the multiple occurrences of a token in a single message, because a keyword we are looking for appears in all messages of the same type. We call the number of messages containing a vanilla token $t$ its message frequency, denoted as $h(t)$. The vanilla tokens with high message frequency are used to extract candidate positional keywords (lines 16–22).

Formally, we use $F(t(i))$ to denote the set of messages containing the positional token $t(i)$:

$$F(t(i)) = \{m \mid \exists t_m^i \in \text{message } m, \text{s.t.}, t_m^i = t(i), 1 \leq m \leq |M|\}, \quad (1)$$
Algorithm 1 Positional Keyword Identification

1: Inputs: Threshold $\rho$, and a set $M$ of raw messages.

2: Output: A set $S$ of positional tokens and their frequencies in set $\{f(t_{(i)})\}$; and, a set $G$ of frequent tokens and their frequencies in set $\{f(t)\}$ and their message frequencies in set $\{h(t)\}$.

3: Initialize: A multiset $V = \emptyset$ for vanilla tokens, a multiset $P = \emptyset$ for positional tokens, a set $H = \emptyset$ for frequent vanilla tokens.

4: for $(m$-th message and $m \in \{1, \ldots, |M|\})$ do

5: Generate a multiset $P_m$ of positional tokens and a multiset $V_m$ of vanilla tokens for message $m$.

6: $P = P \cup P_m$, $V = V \cup V_m$.

7: Convert $V_m$ from multiset to set.

8: for $(t \in V_m)$ do

9: if $(t \notin H)$ then

10: $H = H \cup \{t\}$, $h(t) = 1$

11: else

12: $h(t) = h(t) + 1$

13: end if

14: end for

15: end for

16: $[G, \{f(t)\}] \leftarrow \text{CountFrequency}(V)$

17: $[S, \{f(t_{(i)})\}] \leftarrow \text{CountFrequency}(P)$

18: for $(t \in H)$ do

19: if $(h(t) < \rho|M|)$ then

20: $G = G \setminus \{t\}$, $H = H \setminus \{t\}$, $S = S \setminus \{t_{(i)} \mid t_{(i)} = t\}$.

21: end if

22: end for

where, $|M|$ is the cardinality of message set $M$, and $t_{(i)}^m = t_{(i)}$ means they share the same token $t$ and appear at the same position. Then, the following gives
the frequency (or message frequency) of $t_{(i)}$:

$$f(t_{(i)}) := |F(t_{(i)})|. \quad (2)$$

The generic frequency, $f(t)$, of a vanilla token $t$ can be calculated by

$$f(t) = \sum_{i: t_{(i)} = t} f(t_{(i)}), \quad (3)$$

where, $t_{(i)} = t$ means they share the same token $t$. Hence, $f(t)$ gives the total number of occurrences of $t$ in all messages. Similarly, we use $H(t)$ to denote the set of messages containing vanilla token $t$:

$$H(t) = \{ m | \exists t_{(i)} \in \text{message } m, s.t., t_{(i)} = t, \ 1 \leq m \leq |M| \}. \quad (4)$$

Then, the following gives the message frequency of vanilla token $t$:

$$h(t) := |H(t)|. \quad (5)$$

Hence, $h(t)$ gives the number of messages that contain one or more occurrences of token $t$.

Suppose $\rho$ is a threshold for keyword extraction, the following set $G$ returns all frequent vanilla tokens:

$$G = \{ t | h(t) \geq \rho |M| \}, \text{ where } 0 \leq \rho \leq 1. \quad (6)$$

Then, the following set $S$ gives all the candidate positional keywords:

$$S = \{ t_{(i)} | t_{(i)} = t, \ t \in G \}. \quad (7)$$

Taking the messages in Figure 2 as an example, if $\rho = 0.25$ (i.e., message frequency threshold is 2, given there are 8 messages), the positional tokens such as “LDAP(0)”, “Add(1)” and “Request(2)” will be extracted as candidate positional keywords. Note that, payload data such as “Cheong” and “PRICE” are often extracted as keywords by existing methods, as their generic frequency is 3. By using message frequency, P-token can correctly exclude them from candidate keywords, as their message frequency is 1. Hence, message frequency can successfully remove those kinds of false keywords, which appear repeatedly in a
single message (or small set of messages) but are not present in the other messages. In the next sub-step, we analyze token occurrences so as to determine true keywords from the set $S$ of candidate positional keywords.

4.2.2. Keyword Determination via Keyword Occurrence Analysis

A token may appear multiple times in a message. Some of these occurrences are keywords while others are in payload. Having identified the candidate positional keywords in the previous sub-step, in this sub-step (see Algorithm 2), we distinguish the true keyword occurrences from the false keyword occurrences (appeared in payload) by studying the occurrence and recurrences of candidate keywords in individual messages.

Let us first consider the most straightforward case where a vanilla token occurs the same number of times in each of the messages containing it. In this case, all its occurrences in these messages are regarded as (true) keyword occurrences. For example, the token “o” appears once in all the messages containing the token in Figure 2. It corresponds to 3 different positional tokens “o(32)”, “o(28)”, and “o(27)”. As they correspond to the same keyword, we use “o(27)” (at the lowest position) to represent all the three positional tokens, update frequency $f(o(27))$ to 5 (i.e., the message frequency of the vanilla token “o”), and remove “o(32)” and “o(28)” from $S$. Another example would be the token “ou”, which is a repetitive keyword with a fixed repetition 2. It corresponds to positional tokens: “ou(24)”, “ou(28)”, “ou(19)” and “ou(23)”. For simplicity, we treat the repetitive “ou” as one keyword. This will not affect the clustering step, as they act like a unit that always appear simultaneously in the messages containing them. Hence, we keep “ou(19)” (at the lowest position) in $S$, remove all the other positional tokens of “ou” from $S$, and update the frequency $f(ou(19))$ to 5 (i.e., the message frequency of the vanilla token “ou”).

Formally, for a vanilla token $t$, we use $X = \{x \mid t_{(x)} = t, t_{(x)} \in S\}$ to denote the set of all the possible positions of $t$ in $S$, and $i_0$ and $i_U$ denote the minimum and maximum positions in $X$. If $t$ has a fixed occurrence across the messages containing $t$, we use $t_{(i_0)}$ as the keyword to represent all positional token of $t,$
update the frequency $f(t_{(i)})$ as the message frequency, $h(t)$, of vanilla token $t$.

Please refer to lines 4–7 in Algorithm 2.

On the other hand, messages containing a given vanilla token may have different occurrences of the token from message to message. For example, the token “dn” appears 1 time in all the messages containing the token, except 2 times in the 4-th message. It corresponds to 3 positional tokens: “dn(17)”,”dn(12)” and “dn(41)”, with frequencies 1, 3 and 1, respectively. Note that “dn(17)” and “dn(12)” correspond to the same keyword, but “dn(41)” is a false keyword from payload. For these cases, we determine the true keyword occurrences by exploring their position distribution.

Formally, for a vanilla token $t$, we use $X(t) = \{ x \, | \, t(x) = t, t(x) \in S \}$ to denote the set of all the possible positions of $t$ in $S$. Given a window size $\delta_t$, we can split $X$ into $\left[ \frac{x_U - x_0}{\delta_t} \right]$ windows, where $x_U$ and $x_0$ are the maximum and minimum positions in $X$. For each window, say $[x, x + \delta_t)$, we can calculate the probability density of $t$ in the window as follows,

$$p(t, x) = \frac{\sum_{i \in [x, x + \delta(t))} f(t(i))/\delta(t)}{V_t},$$

where $V_t$ is the volume of $X(t)$. Intuitively, the high-density window is more likely to contain the true keyword occurrences, while the low-density window may contain false keyword occurrences. In order to identify those high-density windows, we adopt Parzen-Window Density Estimation [22] to estimate the window size:

$$\delta(t) = 1.06 \cdot \sigma_t \cdot V_t^{-1/5},$$

where $\sigma_t$ is the standard deviation of $t$’s positions. Note that the smaller the standard deviation, the smaller the size of the window.

To filter out the false keyword occurrences, we particularly set a probability density threshold $\epsilon(t)$ as half of $t$’s average probability density:

$$\epsilon(t) = \frac{1}{2} \cdot \frac{\sum_{x \in X} f(t(x))/\delta(t)}{N(t) \cdot V_t},$$

where $N(t) := \left[ \frac{x_U - x_0}{\delta_t} \right]$ is the number of $t$’s position windows. Then, for each window, say $[x, x + \delta_t)$, if $p(t, x) \geq \epsilon(t)$, we use the first positional token $t(x_0)$ in
the window as the keyword to represent all positional tokens of \( t \) in the window, and aggregate the frequencies in the window to \( t_{(x_0)} \) as follows:

\[
f(t_{(x_0)}) = \sum_{i \in [x, x+\delta(t)]} f(t_{(i)}),
\]

and remove all the other positional tokens of \( t \) in the window from set \( S \); if \( p(t, x) < \epsilon(t) \), we remove all the positional tokens of \( t \) in the window and skip to the next window of \( t \). Please refer to lines 8–25 in Algorithm 2.

After simple calculation, we get the window size for “dn” as 7 and the threshold \( \Delta_{dn} = 0.018 \). Hence, we obtain 4 windows \([12, 19), [19, 26), [26, 33) \) and \([33, 40) \) with probability densities: 0.120, 0, 0 and 0.017, respectively. Hence, only the positional tokens in the first window can be treated as a true keyword. Thus, we use “dn(12)” to represent all the positional tokens in the first window, and we update its frequency \( f(dn(12)) \) to 5. Algorithm 2 presents the details of keyword determination via keyword occurrence analysis analysis.

### 4.3. Message Clustering (Step 2)

The purpose of this step is to group messages into clusters so that each cluster is of high homogeneity in terms of the keywords sequence across all the messages in the cluster and therefore corresponds to a message type. This step involves two sub-steps: (1) vectorizing messages based on positional keywords, (2) clustering messages through a two-level hierarchical clustering strategy.

#### 4.3.1. Message Vectorization

Now, we use the positional keywords in set \( S \) as features for message clustering. In total, we have \( |S| \) features, where \( |S| \) denotes the size of set \( S \). Hence, for an arbitrary message \( m \), we can define a \( 1 \times |S| \) vector, \( v_m \), as follows:

\[
v_m = [w(t_{(i)})]_{1 \times |S|}, \quad \forall \ t_{(i)} \in S,
\]

where, \( w(t_{(i)}) \) is the weight of feature \( t_{(i)} \) in message \( m \). For each feature \( t_{(i)} \in S \), we measure its weight in message \( m \) by examining if it is covered by the token sub-sequence \( \{t^m_{(i)}, t^m_{(i+1)}, \ldots, t^m_{(i+\delta(t_{(i))})}\} \) in message \( m \), where \( \delta(t_{(i)}) \)
**Algorithm 2** Keyword Determination via Keyword Occurrence Analysis

1: **Inputs:** A set $S$ of positional tokens and their frequencies $\{f(t(i))\}$, and a set $G$ of vanilla tokens and their message frequencies $\{h(t)\}$.

2: **Output:** Updated set $S$ and set $\{f(t(i))\}$, and a set $\Delta$ of window sizes.

3: **for** $(t \in G)$ **do**

4:  Get $t$’s positions: $X = \{x | t(x) = t, t(x) \in S\}$; sort $X$ in ascending order.

5:  **if** $(t$ has a fixed occurrence across the messages containing $t$) **then**

6:      Set the window size $\delta(t) = i_U - i_0$, where $i_U = \max X$, $i_0 = \min X$.

7:      Update $f(t(i_0)) = h(t)$ and $S = S \setminus \{t_x | x \in X, x \neq i_0\}$

8:  **else**

9:  Get window size $\delta(t)$ from Eq. [9] and threshold $\epsilon(t)$ from Eq. [10].

10:  **for** $(j = 0 .. |X| - 1)$ **do**

11:       **for** $(k = (j + 1) .. (|X| - 1))$ **do**

12:          **if** $(X(k) - X(j) > \delta_t)$ **then**

13:              Get density $p(t, j)$ in window $[X(j), X(k))$ using Eq. [8].

14:              **if** $(p(t, j) \geq \epsilon(t))$ **then**

15:                  $f(t_{(i(j))}) = \sum_{i \in [X(j), X(k))} f(t(i))$.

16:                  $\Delta = \Delta \cup \{X(k-1) - X(j)\}$, $S = S \setminus \{t_{(i)} | i \in (X(j), X(k))\}$

17:              **else**

18:                  $S = S \setminus \{t_{(i)} | i \in [X(j), X(k))\}$

19:          **end if**

20:      **break**;

21:  **end if**

22:  **end for**

23:  $j = k$.

24:  **end for**

25:  **end if**

26: **end for**

is the window size of $t_{(i)}$. If $t_{(i)}$ is not covered by the sub-sequence, we set $w(t_{(i)}) = 0$; otherwise, set $w(t_{(i)})$ as the frequency of $t_{(i)}$ normalized by the
Inverse Document Frequency (IDF) \[23\]:

\[ w(t_{(i)}) = f(t_{(i)}) \cdot \log \left( \frac{|M|}{f(t_{(i)})} \right), \]

(13)

where, \( f(t_{(i)}) \) is the frequency of feature \( t_{(i)} \) in its corresponding window \( \delta(t_{(i)}) \) (see Eq. \[11\]). IDF normalization was proposed by Sparck Jones \[23\] with the aim to give greater weight on discriminator terms that occur in few messages and less weight on those terms that occur in many messages. For example, “ID” appears in every LDAP message in Figure 2 but its IDF normalized weight is 0. “Add” and “Modify” are message-type keywords, with a much higher IDF normalized weight 2.4. Therefore, using IDF normalization can help us differentiate messages of different types by emphasizing their differences in keywords.

Finally, we get a \(|M| \times |S|\) weight matrix, \( W = [w_{mk}] \), for the given message set \( M \). If the \( k \)-th feature appears in the \( m \)-th message, we put \( w_{mk} \) at the corresponding position in matrix \( W \), where \( w_{mk} \) is the weight in Eq. \[13\]. Please refer to lines 3–14 in Algorithm 3 for the detail of this sub-step.

4.3.2. Two-level Hierarchical Message Clustering

In this sub-step (lines 15–22 in Algorithm 3), we adopt density peaks (DP) clustering algorithm \[24\], a popular density-based clustering method, to cluster the weighted vectors. DP aims at finding the nodes (i.e., messages) with the largest density around their neighboring nodes as the cluster centers. It is based on the intuition that the cluster centers are surrounded by neighbors with lower local density, and cluster centers are at a relatively large distance from any nodes with a higher local density. Here, the distance between any two messages is calculated as the Euclidean distance based on their corresponding vectors. The neighbors of a node are determined by a given cutoff distance threshold. The local density of a node is defined as the number of neighboring nodes within the cutoff distance. Readers can refer to reference \[24\] for the detailed process of the DP algorithm.

Note that, after performing DP clustering, we may mix messages of different types in one cluster, due to the high consistency of the certain extracted key-
**Algorithm 3** Message Clustering

1. **Inputs:** A set $M$ of messages, and a set $S$ of positional keywords and their window sizes in set $\Delta$ and their frequencies in set $f$. 
2. **Output:** Message clusters $C = \{c_i\}$. 
3. **Initialize:** A zero matrix $W \in \mathbb{R}^{|M| \times K}$. 
4. **for** $(t(i) \in S)$ **do** 
   5. Calculate the IDF normalized weight $w(t(i))$ based on Eq. (13). 
5. **end for** 
6. **for** $(k \in \{0, \ldots, |S| - 1\})$ **do** 
   7. $t(i) \leftarrow S(k)$; get $t(i)$’s window size $\delta(t(i))$ from set $\Delta$. 
   8. **for** $(m \in \{1, \ldots, |M|\})$ **do** 
   9. if $(t(i) \subseteq \{t^m(i), t^m(i+1), \ldots, t^m(i+\delta(t(i)))\})$ then 
   10. Set $W(m, k) = w(t(i))$. 
   11. **end if** 
   12. **end for** 
6. **end for** 
13. Apply DP clustering on matrix $W$ and obtain clusters $C = \{c_1, c_2, \ldots, c_L\}$. 
14. Get $Sim(c_k)$ for each cluster $c_k \in C$ from Eq. (14) and $\tau$ from Eq. (15). 
15. **for** (cluster $c_k \in C$) **do** 
16. if $(Sim(c_k) < \tau)$ then 
17. Perform a second level clustering on $c_k$ to get its sub-clusters 
18. Put the sub-clusters into $C$ and remove $c_k$ from $C$. 
19. **end if** 
20. **end for** 

words in those messages. This issue also commonly affects other keyword based format extraction methods. For example, the LDAP Add messages have many positional keywords in common with Modify messages (see Figure 2), so that their vectors present a relatively small distance. Hence, they are likely to be clustered together. In the following, we identify such mixed clusters and refine each of them through a second level clustering.
To determine which clusters need refinement, we adapt the measurement introduced in [21] to calculate the average similarity (i.e., homogeneity) of messages within each cluster. For a given cluster, say $c_k$, the following calculates the average similarity of messages in $c_k$:

$$\text{Sim}(c_k) = \frac{|c_k|^2}{\sum_{m_1 \in c_k} \sum_{m_2 \in c_k} d(m_1, m_2)}.$$  \hspace{1cm} (14)

where, $|c_k|^2$ the cardinality of cluster $c_k$, and the distance $d(m_1, m_2)$ is calculated based on the Needleman-Wunsch alignment for messages $m_1$ and $m_2$. Here, instead of aligning the raw messages of $m_1$ and $m_2$, we encode tokens as one byte characters/symbols and align the encoded message. This substantially shortens the message length and hence significantly decreases the computational cost for Eq. (14). Given a threshold $\tau$, if $\text{Sim}(c_k) < \tau$, we refine the cluster $c_k$ by performing a further clustering on the messages in $c_k$. Suppose we obtain the sub-clusters $\{c_{k_j}\}$ of $c_k$, and we replace $c_k$ in $C$ by the newly generated sub-clusters $\{c_{k_j}\}$. If $\text{Sim}(c_k) > \tau$, we keep the cluster $c_k$ in $C$. Similar to the setting of density threshold $\epsilon(t)$ in Eq. (10), we particularly set the similarity threshold $\tau$ as half of the average similarity across all clusters:

$$\tau = \frac{1}{2|C|} \sum_{c_k \in C} \text{Sim}(c_k).$$  \hspace{1cm} (15)

As such, even if the LDAP Add messages and Modify messages are mixed in one cluster, we can refine the cluster and successfully separate them into individual clusters. Hence, P-token can achieve fine-grained message clusters.

### 4.4. Format Extraction (Step 3)

Finally, we infer message formats for the clusters in $C$. For each cluster, we extract the common keyword pattern from the multiple sequences (messages) to infer the message format. Many works have proven that multiple sequence alignment is a NP-hard problem [25 26]. Some heuristic algorithms have been proposed, but they are still very time complex [27]. Here, we utilize the natural positions of positional keywords for format inference. The obtained message formats are presented in the form of regular expressions.
For each cluster \( c_k \in C \), we first extract a set \( S_k \) of common positional keywords by using Algorithms 1&2 with threshold \( \rho = 1.0 \). Then, we examine the positional tokens in each message \( m \in c_k \). For an arbitrary positional token \( t_{(i)}^m \) in message \( m \), if there exists a positional keyword \( t'_{(j)} \in S_k \) such that \( t_{(i)} \) and \( t'_{(j)} \) have the same string and position \( i \) is in the window of \( [j, j + \delta(t_{(j)})] \), we use \( t'_{(j)} \) to replace \( t_{(i)}^m \) in message \( m \); otherwise, we label \( t_{(i)}^m \) as payload (denoted as "\( \Box \)"). For consecutive "\( \Box \)", we convert them into a single "\( \Box \)". In this way, we convert each message in cluster \( c_k \) into a sequence of tokens, which are either keywords or "\( \Box \)". For instance, we get the following sequence of common tokens for the LDAP Add messages in Figure 2:

LDAP Add Request Message ID: □ LDAP Add Request Protocol Op dn: cn= □,ou=□, ou=□,o=DEMOCORP,c=AU mail: □@ca.com sn: □

To derive the message format for each cluster of messages, we need to consider the issues that a message may contain optional sub-sequences of tokens or repetitive sub-sequences of tokens. Note from Section 4.2.2 that, some keywords have a fixed occurrence, but some have a variable occurrence, for a given type of message. Therefore, the keyword(s)-payload sequences of the messages in a cluster may have different lengths. For example, if we remove "\( \text{ou=Liaison,} \)" from the second message in Figure 2, "\( \text{ou=} \)" will become an optional keyword in the LDAP Add messages. Then, we will generate two different keyword(s)-payload sequences for the Add messages. Furthermore, some messages may have repetitive sub-sequences, e.g., the "\( \text{,ou=}\Box \)" pattern appearing multiple times in the messages in Figure 2. In order to address these issues in message formats, we utilize Synoptic [28] to extract message format out of the keyword(s)-payload sequences (messages) of a cluster. Synoptic extracts a finite state machine (FSM) event model from event sequences, by merging and generalizing the event sequences. Here, we treat tokens (either keywords or "\( \Box \)") as ‘events’. We feed the keyword(s)-payload sequences (messages) of each cluster into the Synoptic tool.

The output from Synoptic is the message format in the form of a FSM in terms of the tokens. Finally the FSM model is converted into a regular expression as
Algorithm 4 Format Extraction

1: Inputs: Message clusters \(C\).
2: Output: Message formats \(\{f_i \mid i \in \{1, \cdots, |C|\}\}\).
3: for (cluster \(c_k \in C\)) do
4: Extract a set \(S_k\) of common positional keywords and their window sizes in set \(\Delta_k\).
5: for (message \(m \in c_k\)) do
6: Get the positional tokens for message \(m\): \(P_m = \{t^m_{(0)}t^m_{(1)} \cdots t^m_{(l_m-1)}\}\).
7: for \( (t^m_{(i)} \in P_m)\) do
8: if \( (\exists t'_{(j)} \in S_k \text{ s.t. } t = t' \& 0 \leq i - j \leq \Delta_k(t_{(j)}) ) \) then
9: Replace \( t^m_{(i)} \) by \( t'_{(j)} \) in \(P_m\), continue.
10: else
11: Set \( t^m_{(i)} = \square \).
12: end if
13: end for
14: Convert consecutive \(\sqcap\) into single \(\sqcap\) in \(P_m\).
15: end for
16: Feed the token sequences \(\{P_m\}\) into Synoptic to generate a FSM, and convert the FSM to a regular expression as the message format, \(f_i\).
17: end for

the message format by replacing \(\sqcap\) with \(\text{.*}\). The details of the process in this step are presented in Algorithm 4.

Note that, Synoptic can successfully capture the pattern of repetition (e.g., the iterative occurrences of \(\sqcap\) and \(\text{ou=}\)) in the messages in Figure 2 from individual message instances, which commonly occur in protocol message formats but fail to be captured by previous methods. For example, after generating the FMS of tokens and converting the FMS into a regular expression, we get the following message format for the LDAP Add messages:

\[
\text{LDAP Add Request Message ID: .* LDAP Add Request Protocol Op dn: cn= (.*,ou=)'.*,ou=.*,o=DEMOCORP,c=AU mail: .*@ca.com sn: .* cn: .*}
\]
5. Experimental Results

In this section, we evaluate our proposed approach, P-token, for extracting protocol message formats on raw message traces collected from real-world services and applications. Here, we first introduce the datasets used to evaluate our approach, then define the evaluation metrics, and finally present the experimental results.

5.1. Datasets

We have used Wireshark [29] – an open source packet analyzer – to capture message traces of the selected systems. The captured traces were used as input to the Java implementation of our approach. We have applied our approach to four datasets corresponding to four different real-world systems using four different protocols.

LDAP (Lightweight Directory Access Protocol) is a broadly accepted standard for high-speed maintaining and accessing distributed directory information [30]. It is a binary protocol that uses an ASN.1 [31] encoding to encode and decode text-based message information to and from its binary representation, respectively. It is mainly used in Internet applications. The textual LDAP interaction traces are used for experiments in this paper, which contain 2,181 messages of 8 different message types, including Bind, Unbind, Search, Add, Delete, Compare, Modify and Modify DN.

SOAP (Simple Object Access Protocol) is a lightweight messaging protocol used for exchanging structured information between peers in distributed and decentralized web services [32]. It relies on XML for its messaging format and other application layer protocols, in particular HTTP and SMTP. The SOAP dataset used in our experiments is collected from a banking web service. It contains 1000 messages of 6 different message types, i.e., DepositMoney, WithdrawMoney, GetTransactions, GetAccount, GetNewToken and DeleteToken.

REST (REpresentational State Transfer) paradigm is a lightweight implementation of a service-oriented architecture [33]. RESTful API services a
lightweight implementation of a service-oriented architecture. Here, the Twitter REST API is used to collect data, which provides a number of service operations to automate Twitter functionalities. The message contents are in JSON format. The collected Twitter dataset contains 1825 messages of 5 different types, including SearchTweets, StatusesUsertimeline, StatusesUpdate, StatusesShow, and FriendshipsShow.

IMS (IBM Information Management System) is a joint hierarchical database and information management system with extensive transaction processing capabilities [34]. IMS is a binary mainframe protocol. The IMS traces dataset used in our experiments consists of 800 messages containing 5 types of IMS operations, which are ADD, DELETE, DISPLAY, UPDATE, and ACK.

To provide the reference points for our evaluation, we have manually separated the messages into clusters according to the message types and obtained the ground truth for the message clustering to compare to.

5.2. Evaluation Metrics

To compare the effectiveness of P-token with other state-of-the-art techniques in clustering protocol messages and extracting protocol message formats, we adopt the following three commonly used metrics: Precision, Recall and F-measure, from the area of information retrieval [35].

\[
\text{Precision} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}}, \tag{16}
\]

\[
\text{Recall} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}}, \tag{17}
\]

\[
\text{F-measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{18}
\]

Message Clustering: For message clustering, a TruePositive pair assigns two messages of the same type to one cluster. A FalsePositive pair assigns two different types of messages to one cluster. A FalseNegative pair assigns two messages of the same type to different clusters. Hence, Precision defines the
fraction of message pairs correctly put in the same cluster and Recall defines the fraction of ground-truth pairs that were identified.

**Format Extraction:** For a particular message type, we define \( \text{TruePositive} \) as the number of messages of this type that are accepted by the extracted format(s) corresponding to this type; \( \text{FalseNegative} \) as the number of messages of this type that are rejected by the corresponding extracted message format(s); and \( \text{FalsePositive} \) as the number of messages of other message types that are accepted by the inferred format(s) corresponding to this type. We further apply 10-fold cross validation [36] to evaluate the accuracy in message formats extraction. More specifically, for a given set of protocol messages, we calculate Precision, Recall and F-measure for each message type and get the average Precision, Recall and F-measure across all message types.

5.3. Results

In this subsection, we present the experimental results in terms of the accuracy of message clustering and the effectiveness of format extraction. In this paper, we typically set the threshold \( \rho = 0.07 \), i.e., we assume that the smallest cluster accounts for 7% over all messages in a message trace. Hence, the keywords appearing in at least 7% messages will be extracted in first level of clustering, and others can be caught in the second level of clustering. Here, we compare the result of our approach with two state-of-art tools (ProDecoder [20, 37] and AutoReEngine [19]) and two baseline methods: “vanilla” token and P-token without the second level clustering (called “Naive” P-token).

**ProDecoder.** It adopts the LDA model [15] (used for frequent pattern mining in Natural Language Processing) to detect keyword patterns. Instead of using a keyword transition matrix, ProDecoder utilizes keyword tuples as features in the clustering algorithm, Information Bottleneck (IB) [38]. Finally, a Multiple Sequence Alignment (MSA) algorithm is adopted to infer message formats. We compare P-token with ProDecoder to emphasise the importance of considering keywords’ positions in message formats extraction. As suggested in [20], we set the number of “topics” as 40, the number of “top words” as 100, \( \alpha=0.1, \beta=0.01, \)
and the number of iterations as 2000.

AutoReEngine. It adopts the Apriori [39] algorithm for keywords extraction. Those with large position variations are filtered out as noise. Then, messages are clustered based on the similarity of keyword sequences. The merged keyword series in each cluster is regarded as the corresponding message format. We compare P-token with AutoReEngine to emphasise the importance of considering multiple keyword occurrences, false keyword, and message formats overgeneralization. Similar to P-token, we set the threshold $\rho$ as 0.07 for keyword identification and keyword group extraction.

Vanilla. Compared to P-token, the first baseline method, “vanilla token”, follows the basic approach presented in this paper, but we do not consider the position of tokens, neither do we perform the second level clustering. Hence, by comparing P-token with this baseline, we can see the importance of considering the position information of keywords.

P-token without the second level clustering (Naive P-token). For the second baseline method, we consider positional keywords but without refining the cluster by a second level clustering. Again, all the other parts are the same as P-token. Hence, by comparing P-token with this baseline, we can see the effects of two-level hierarchical clustering before inferring message formats. The parameter settings for P-token were also used for the two baseline algorithms.

5.3.1. Message Clustering

Table 1 reports the results of message clustering. As we can see, P-token outperforms the state-of-art methods and the two baseline methods in terms of Precision and F-measure in message clustering on all the datasets. In the following, we analyze the performance of P-token by comparing to other methods.

ProDecoder uses LDA [15], from Natural Language Processing, for message clustering. However, protocol messages are machine generated languages which follow determined message formats/templates in terms of a keyword-payload sequence, while human natural languages does not follow these kinds of formats. Hence, directly applying LDA on machine generated languages fails to extract
<table>
<thead>
<tr>
<th>Protocols</th>
<th>ProDecoder</th>
<th>AutoReEngine</th>
<th>Vanilla</th>
<th>Naive P-token</th>
<th>P-token</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAP</td>
<td>P 0.97</td>
<td>0.97</td>
<td>0.67</td>
<td>0.91</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>R 0.81</td>
<td>0.13</td>
<td>0.76</td>
<td>0.87</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>F 0.88</td>
<td>0.23</td>
<td>0.71</td>
<td>0.89</td>
<td>0.91</td>
</tr>
<tr>
<td>SOAP</td>
<td>P 0.89</td>
<td><strong>1.00</strong></td>
<td>0.91</td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td></td>
<td>R 0.98</td>
<td>0.82</td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td></td>
<td>F 0.93</td>
<td>0.90</td>
<td>0.96</td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td>REST</td>
<td>P 0.99</td>
<td>0.85</td>
<td>0.79</td>
<td>0.88</td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td></td>
<td>R 0.88</td>
<td>0.15</td>
<td>0.83</td>
<td><strong>0.93</strong></td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>F 0.93</td>
<td>0.26</td>
<td>0.81</td>
<td>0.90</td>
<td><strong>0.94</strong></td>
</tr>
<tr>
<td>IMS</td>
<td>P 0.78</td>
<td>0.99</td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td></td>
<td>R 0.58</td>
<td>0.99</td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
</tr>
<tr>
<td></td>
<td>F 0.68</td>
<td>0.99</td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1.00</strong></td>
</tr>
</tbody>
</table>

*Note: P denotes Precision, R denotes Recall, and F denotes F-measure.*

the relationships between message keywords correctly. In contrast, P-token takes the advantages of keyword’ positions in the keyword-payload formatted messages. Therefore, P-token outperforms ProDecoder. ProDecoder shows 0.88 in F-measure for LDAP, while P-token achieves 0.91 in F-measure. ProDecoder has F-measures of 0.93, 0.93 and 0.68 for SOAP, REST and IMS, respectively, whereas P-token achieves F-measures of 1.00, 0.94 and 1.00 on these datasets.

AutoReEngine uses positions for keyword identification. For any candidate keyword, it is kept as a keyword if it has a low standard deviation of positions; otherwise, it is filtered as a noise. Note that, due to the big variation of message payload in terms of payload length, many keywords present large standard deviations of positions. Hence, AutoReEngine often fails to extract those kinds of keywords, but identifies many false keywords and uses them for clustering. This leads to AutoReEngine having a lower performance than P-token. Furthermore, as AutoReEngine does not assume the number of message types is known, it
often separates messages of one type into many groups and, consequently, generates more clusters than the actual clusters. Thus, it shows relatively high Precision but low Recall in some datasets. For example, the recall is below 0.20 on LDAP and REST.

The proposed P-token outperforms the two baseline methods (“Vanilla” and “Naive P-token”). (1) Overall, “Vanilla” shows better results than ProDecoder and AutoReEngine on SOAP and IMS, and “Naive P-token” performs better than “Vanilla” on all the datasets. “Vanilla” adopts the tokenization technique instead of the n-gram technique used in ProDecoder and AutoReEngine for keyword extraction. Tokenization automatically segments a message into tokens (keywords or payload strings) by using the natural separators/delimiters in messages, while n-gram based approaches break messages into sub-strings of chosen lengths. The n-gram technique can be used to extract keywords from binary messages (where there may be no delimiter), but the limitation of applying an n-gram approach to a textual protocol is that many sub-strings of true keywords are wrongly recognized as true keywords. On the other hand, as “Vanilla” simply uses the frequency analysis without considering the template structure of protocol messages. Therefore, “Vanilla” outperforms ProDecoder and AutoReEngine on some datasets (SOAP and IMS) but shows lower performance on some other datasets (LDAP and REST). This justifies the effectiveness of using tokens for keyword extraction. (2) Compared to “Vanilla”, “Naive P-token” further utilizes the position information for each token. As we presented, associating tokens with position information can successfully address the keyword repetition issue and the ambiguity between true keywords and false keywords. Hence, “Naive P-token” outperforms “Vanilla”. This justifies the effectiveness of associating tokens with position information for the purpose of keyword identification. (3) Comparing P-token to the “Naive P-token” method, the results show that P-token achieves better clustering results, by addressing the mixed cluster issue. While P-token sometimes finds more clusters than the ground truth, which slightly decreases the Recall value, it achieves higher Precision and F-measure than “Naive P-token”. This justifies the effectiveness of refining
Table 2: Accuracy of Format Inference

<table>
<thead>
<tr>
<th>Protocols</th>
<th>ProDecoder</th>
<th>AutoReEngine</th>
<th>Vanilla</th>
<th>Naive P-token</th>
<th>P-token</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAP</td>
<td>P 0.88</td>
<td>0.70</td>
<td>0.90</td>
<td>0.90</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>R 0.65</td>
<td>0.19</td>
<td>0.78</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>F 0.75</td>
<td>0.30</td>
<td>0.84</td>
<td>0.95</td>
<td>0.99</td>
</tr>
<tr>
<td>SOAP</td>
<td>P 0.93</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>R 0.93</td>
<td>0.60</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>F 0.93</td>
<td>0.74</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>REST</td>
<td>P 0.79</td>
<td>0.30</td>
<td>0.92</td>
<td>0.81</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>R 0.71</td>
<td>0.17</td>
<td>0.92</td>
<td>0.91</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>F 0.75</td>
<td>0.22</td>
<td>0.92</td>
<td>0.86</td>
<td>0.99</td>
</tr>
<tr>
<td>IMS</td>
<td>P 0.78</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>R 0.53</td>
<td>0.31</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>F 0.63</td>
<td>0.47</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

*Note: P denotes Precision, R denotes Recall, and F denotes F-measure.

clustering by using a two-level hierarchical clustering.

Additionally, as we see P-token achieves 1.0 F-measure for SOAP and IMS, but only 0.91 and 0.94 F-measure for LDAP and REST. Based on our observation, LDAP and REST present larger variation in terms of payload length than SOAP and IMS. Note that, a large variation in payload length leads to a large window size $\delta(t)$ (see Eq. (9)). This may incorporate terms from payloads into keywords. Hence, P-token presents low performance on LDAP and REST.

5.3.2. Format Extraction

In Table 2 we report the results of extracting message formats on the four datasets. Overall, our approach significantly outperforms ProDecoder and AutoReEngine. As we can see, P-token achieves near perfect Precision, Recall and F-measure for all the datasets. As ProDecoder and AutoReEngine infer message formats from the message clusters, their format accuracy is dependent on the message clustering accuracy. As we can see in Table 2 AutoReEngine
presents a relatively low Recall in message format extraction, as it generates many more clusters than the ground-truth classes in message clustering. As a consequence, the inferred message formats became too specific and could not accept many messages of same type from the test datasets. Thus, AutoReEngine achieves very low Recall values for message format inference for every dataset. Although ProDecoder presents comparatively better Precision and Recall than AutoReEngine, ProDecoder classified messages as the wrong type due to the mis-identification of keywords. Further, it adopts MSA to infer message formats, which often involves substring from payload into message formats. As a consequence, ProDecoder often generates over-generalized message formats or error formats, which decreases the Precision and Recall of format extraction. As P-token achieves comparatively high accuracy in clustering protocol messages, it also achieves high performance in extracting message formats compared to ProDecoder and AutoReEngine.

From the results of the two baseline methods, we can see in Table 2 that the “Naive P-token” baseline outperforms the “Vanilla” baseline, and the proposed P-token outperforms the “Naive P-token” baseline in all of the four datasets. This further justifies the effectiveness of our approach in considering both the position of keywords and refining the clusters with low homogeneity for fine-grained format extraction.

6. Discussions and Limitations

In this section, we further reflect on our approach and discuss limitations of its applicability in the context of different types of protocols. Based on our observations in analyzing message formats, protocols can, in general, be classified into three groups: (i) protocols with fixed message formats, that is, a fixed order of keywords, (ii) protocols with free formats (i.e. keywords can, in principle, appear in arbitrary order) but fixed end-point specific interpretations (i.e. each end-point expects a particular order in which keywords occur), and (iii) protocols with free formats and free end-point implementations where keywords
can occur in arbitrary order and no specific order is followed. In the following, we will elaborate on each of these three groups in more detail.

Protocols with fixed formats, such as LDAP [30] and IMS [34], have a fixed order of keywords, that is, for each message type, the extracted keywords always have the same order. As an example, consider the messages given in Fig. 2.
The LDAP protocol defines a fixed order in which the keywords appear: for Add messages, the order is \{"ID", "cn", "ou", "o", "c", "mail", "sn", "cn"\}. Similar fixed orders of keywords exist for all other types of LDAP messages. Even though some of the keywords are repetitive (such as "ou") and some are optional (such as "mail"), they always occur in the same, fixed order. Based on the fundamental assumption of our approach (in Section 3) that message keywords appear at relatively fixed positions, our approach is able to deal with this type of protocols.

The second group of protocols, in principle, do not enforce (or require) a particular order in which keywords are encoded in messages (they do not require a fixed format), but each end-point expects a particular order to be followed – keywords cannot appear in arbitrary order. For example, consider both REST [33] and SOAP [32], that per se do not define a fixed order or keywords. Application developers can use either REST or SOAP as a kind-of transport layer to communicate information between services. However, for a specific service, such as the RESTful Twitter API [33], information to be communicated is generally structured in a particular, pre-defined order, resulting in fixed order in which keywords occur, even if the underlying protocol (in case of Twitter, REST) does not require this particular order. As another example, consider the two different HTTP-encoded messages given below:

GET https://api.twitter.com/1.1/friendships/show.json?source_id=5246&target_id=13309&include_entities=1&include_RTs=1

{"Headers":"Accept":"*/\*","User-Agent":"Opera/9.27 (Windows NT 5.1; U; en) ", "Host": "api.github.com", "Content-Type": "application/json; charset=utf-8", "Request-URI": "/repos/google/guava/issues/2095", "Method": "GET"}
The former type of HTTP format is adopted by the RESTful Twitter API (used in this paper). The later type of HTTP format is adopted by the GitHub REST API. As such, in practice, our approach is still applicable for this kind of free-format protocols as long as they have a fixed message format implementation for a specific service or application, as we have illustrated in Section 5.

Finally, protocols with free formats and free end-point implementation have keywords occurring in arbitrary order in a message, even when being used for a specific service or application. In our experiments, we are yet to come across specific end-point services that process messages that do not follow a particular encoding structure. However, we can envisage situations where information is encoded in key/value pairs (e.g., a map data structure in Java or a Dictionary in Python), but the order in which these key/value pairs is transmitted does not matter and, in practice, occurs in “random” order. These kinds of protocols do not satisfy the fundamental assumption of our approach, that is, keywords appearing at relatively fixed positions. Hence, our approach has limited applicability with protocols of this nature.

In summary, our proposed method is applicable for the protocols with fixed formats and those free-format protocols with fixed end-point specific interpretations, but it has limitations for free-format protocols with free end-point implementations. Based on our observations, many protocols used in practice belong to the first two types, and hence we would argue that our approach is widely (but not universally) applicable. Additionally, our proposed method uses tokenization to segment each message into tokens (keywords or payload strings) by using the natural separators/delimiters in messages, as we have illustrated in Section 2. Hence, our approach has limited applicability with services or applications with non-tokenized messaging protocols.

---

3 https://developer.github.com/v3/
7. Related Work

So far, many approaches have been proposed for extracting protocol message formats from raw message traces. Based on the techniques used in extracting message keywords, existing approaches for extracting protocol message formats can be divided into two categories: (1) n-gram based methods and (2) tokenization based methods.

7.1. n-gram Based Methods

To identify distinct keywords for message format inference, many methods find string patterns of arbitrary length, called n-grams, where \( n \) denotes the number of characters or bytes in the pattern.

AutoReEngine \[19\] first splits messages into \( n \)-grams of different lengths. Then, the Apriori algorithm \[39\] is adopted to identify keywords from the \( n \)-grams. The variation of keywords’ positions is further calculated. Those with large variations are filtered out as noise. Finally, messages are classified according to the similarity among keyword sequences. The keyword series in each cluster is regarded as the corresponding message format. As AutoReEngine adopts Apriori algorithm to extract keywords from \( n \)-grams, parts of message payloads are often included as part of keywords, leading to keyword over-fitting. If a keyword is over-fitted to some messages, AutoReEngine will extract many more formats than the actual formats. Taking the messages in Figure 2 as an example, if “ou=S” is extracted as a keyword, then the Add messages will have two different keyword series. Thus, it will generate two formats for the Add messages.

ProDecoder \[20, 37\] adopts \( n \)-gram for keyword identification in a different way. Instead of identifying a maximum \( n \)-gram as a message keyword, it adopts Latent LDA \[15\] from Natural Language Processing and identifies the keywords as a set of semantically related \( n \)-grams. ProDecoder first decomposes the input messages into \( n \)-grams of the same length and treats them as words. Then, it adopts LDA to extract the topics of the words by analyzing the semantic
relationship among the $n$-grams. The probability of an $n$-gram assigning to each topic is calculated. Then, the probabilities are input into a clustering algorithm, Information Bottleneck (IB) \cite{35}. Finally, an MSA algorithm is adopted to extract message formats. Due to the difference between natural languages and machine generated languages, directly applying LDA causes the inaccuracy of ProDecoder in message clustering. Often, different types of messages are put into a single cluster, consequently leading to format over-generalization.

We recently proposed an $n$-gram based approach for fine-grained message format extraction \cite{4}. In this approach, we first cluster messages through identifying the field that defines message types. More specifically, we generate a common message template by aligning all the messages of a given trace, then the entropy of each field in the common template is calculated, and the one with the lowest entropy is recognized as the message-type field. Hence, messages can be partitioned into type-specific clusters according their message type fields. Then, for each message cluster, we split the messages into $n$-grams and extract those with high frequency as keywords. Finally, for each cluster, the keywords are used to tokenize the messages in the cluster, and we obtain a common sequence of keyword and payload tokens as the message format. In this approach, the repetition of keywords and the ambiguity of true keywords and false keywords were not considered. More importantly, it becomes not applicable for those protocols whose message type fields are not easily identifiable.

7.2. Tokenization Based Methods

As many systems adopt tokenized messaging protocols, where the communication messages can be tokenized by the natural separators and delimiters, such as space(s), tabs, special characters, etc. Applying $n$-gram based methods on tokenized messaging protocols often involve errors as many sub-strings of true keywords are wrongly recognized as true keywords. Hence, some methods focus on tokenized messages.

Discoverer \cite{16} uses a set of predefined delimiters and/or separators to split the messages into tokens. It applies recursive clustering techniques to group
messages with similar patterns of tokens. If two messages present the same token patterns or are very similar, they are placed in the same cluster. Finally, for each cluster, the message format is extracted by using multiple sequence alignment. Wang et al. [40] proposed a technique to extract message formats by using Jaccard index [41] to filter out the tokens with low frequencies and then build an FSM from token sequences as the message formats. The approach proposed by Marko and Tomi [42] focuses on modeling the sequential activities of legitimate network traffic. They particularly applied the method on intrusion detection. It generates a prototype of legitimate network traffic in XML by searching some predefined fields (i.e., source port, IP address, destination port, etc.) in the network traffic. This method becomes not applicable when such prior knowledge about the protocol is not available. The above methods often split a single message type into multiple clusters, due to the conservatism followed when determining whether or not two different messages should be included in the same cluster.

Our technique, P-token, is a tokenization based method for protocol message format extraction. Compared to existing n-gram based methods and tokenization based methods, P-token successfully addressed the keyword misidentification issues using token position variations, and addressed the format over-generalization issue via a two-level hierarchical clustering strategy. Experiments on various protocols demonstrate the effectiveness and efficiency of the proposed P-token for fine-grained message formats extraction.

8. Conclusion and Future Work

In this paper, we have proposed a novel approach called P-token for extracting protocol message formats from raw message traces. P-token does not assume prior knowledge about message structures, nor does it require access to the executable code of applications implementing the protocols concerned. P-token involves three steps: (i) tokenization-based positional keywords identification, (ii) message clustering based on positional keywords, and (iii) positional
keyword-based message format extraction for message clusters. In particular, we identify message keywords based on the position information of keywords, which addresses the keyword mis-identification issues (i.e. keyword underfitting, keyword overfitting, and false keywords) that existed in existing methods. Furthermore, to infer the find-grained message formats, we refine the clusters with mixed message types (after the first level message clustering), and perform a second level of clustering for these clusters. This enables our approach to address the message format over-generalization issue. We have compared our approach with the state-of-the-art approaches for four datasets obtained from real-world applications implementing four protocols. The experimental results have shown that our approach outperforms existing approaches for both precision and recall.

There are a few issues we intend to further explore as future work. As discussed in the end of Section 5.3.1, P-token presents a lower performance for the message traces with large variations in terms of payload length. For these kinds of message traces, P-token might incorporate some terms from payloads into message keywords. Another future work is to analyse message traces with complex nesting structures and infer the nesting structured message formats.

References


URL http://www.wireshark.org

URL http://www.rfc-editor.org/info/rfc1777


[34] W. C. McGee, The information management system ims/vs, part i: General structure and operation, IBM Systems Journal 16 (2) (1977) 84–95.


