Exploiting Intra-Packet Dependency for Fine-Grained Protocol Format Inference

Qun Huang[1], Patrick P. C. Lee[1], and Zhibin Zhang[2]
[1]The Chinese University of Hong Kong
[2]Institute of Computing Technology, Chinese Academy of Sciences, China

***Abstract*—Given the increasing volume and complexity of network traffic nowadays, network operators often leverage application-layer protocols to differentiate network traffic, so as to improve quality-of-service control, security protection, and resource profiling. We present *ProGraph*, a tool that accurately infers protocol message formats at both byte-level and bit-level granularities. Unlike existing approaches that mainly exploit statistical features across packets, ProGraph exploits intra-packet dependency among the values of different portions of a packet payload. It systematically constructs a graphical model that captures intra-packet dependency, using various techniques in graph theory and information theory. It also achieves several important design properties for real deployment, including fine-grained inference, protocol independence, simple parameterization, robustness to noisy training sets, and fast execution. We show via trace-driven evaluations that ProGraph achieves more accurate inference than existing approaches. We further show how ProGraph can be used for classifying traffic.**

I. Introduction

Network operators often leverage application-layer protocols to differentiate network traffic, so as to achieve better quality-of-service control [15] and enhance the designs of security systems such as intrusion detection systems and firewalls [14]. A protocol may perform different tasks (e.g., requests, responses, heartbeats, etc.), so classifying specific tasks of a protocol is also critical for in-depth profiling.

Due to the lack of protocol specifications in general, protocol format inference provides a pre-processing step for traffic classification. It often works by examining packet payloads and inferring protocol *formats*, which specify the lengths and positions of various *fields* that collectively define a protocol. Existing protocol format inference approaches divide packet payloads into consecutive bits (e.g., [3], [8], [9], [12], [16]) and parse the features of the bit groups via statistical analysis. Each bit group needs to be sufficiently long (e.g., one byte [9], [12], [16] or four bits long [3], [8]), so as to provide high degrees of freedom for meaningful statistical analysis. On the other hand, some protocols, such as DNS, define single-bit fields, and this potentially complicates statistical analysis.

We present *ProGraph*, a fine-grained protocol format inference tool that can operate at both byte-level and even bit-level granularities. Our observation is that the values of a portion of a packet payload may determine the values of another portion of the same packet payload. For example, a field that identifies a specific task will affect how other fields are defined related to the task. ProGraph leverages such *intra-packet dependency* as the key payload features. This differentiates ProGraph from previous payload-based inference approaches that either consider the entire packet payload (e.g., [16]) or examine individual payload portions independently (e.g., [3], [8], [12]). To our knowledge, ProGraph is the first work that specifically addresses protocol format inference at the bit-level granularity and exploits intra-packet dependency to address this issue.

ISBN 978-3-901882-68-5 © 2015 IFIP

ProGraph characterizes intra-packet dependency by constructing a *graphical model* for a target protocol. The graphical model defines the legitimate values associated with different portions of a packet payload in the protocol. Starting with a training set of packet payloads, ProGraph exploits various techniques in graph theory and information theory to iteratively refine the graphical model with respect to the statistical distributions of the values observed in different portions of payloads. To this end, ProGraph aims for several design goals:

- **Fine-grained inference:** ProGraph accurately infers the protocol formats in which fields are defined at the byte-level or even bit-level granularities.
- **Protocol independence:** ProGraph builds on statistical analysis, and can extract packet features from any protocol. It also does not rely on any protocol specification for inference.
- **Simple parameterization:** ProGraph involves only few configurable parameters. Also, the selection of parameters is simple, without compromising the accuracy of the inference results.
- **Robustness to noisy training sets:** In general, obtaining a clean training set is non-trivial, as labor-intensive manual inspection is often required to filter unwanted packets [13]. ProGraph maintains accurate inference even only when noisy training sets are available.
- **Fast execution:** ProGraph achieves high-throughput inference in the face of the huge volume of network traffic.

We demonstrate that the graphical models constructed by ProGraph enable us to classify network traffic. We conduct trace-driven evaluations on ProGraph using IP packet traces collected from real-world 3G cellular data networks. We show that ProGraph works well for different protocols operating at byte-level or bit-level granularities. Specifically, it achieves low error rates in all cases we consider, and provides more accurate classification than existing approaches based on longest common subsequences (e.g., [12], [16]). It maintains high accuracy even if we inject noisy packets to the training sets.

The rest of the paper proceeds as follows. Section II reviews related work. Section III overviews the ProGraph design. Section IV elaborates the graphical model construction of ProGraph. Section V describes how to apply the graphical model to classify network traffic. Section VI presents trace-driven evaluation results. Section VII concludes the paper.

II. Related Work

Protocol format inference often operates by examining packet payloads and extracting protocol features. Discoverer [7] automatically infers the protocol format and extracts payload signatures. It uses printable characters in payloads to determine field boundaries, which is ineffective for binary protocols. Polyglot [6], AutoFormat [11], dynamic taint analysis [18], and Netzob [4] leverage semantic information (e.g., execution logs) for reverse engineering, but their approaches are difficult to generalize as the semantic information may be unavailable. ProDecoder [17] applies n-gram techniques to partition binary payloads into fixed-length fields, but fixed-length partitioning is shown to degrade accuracy [19].

Some inference approaches exploit the statistical features of payloads to characterize protocol formats. ACAS [9] treats payloads as features and applies machine learning algorithms. Ma *et al.* [12] build statistical models for the payload distribution, and perform classification based on the likelihood that a packet fits the distribution. Bonfiglio *et al.* [3] identify Skype traffic by examining the extent of randomness of groups. KISS [8] partitions payload into fix-length groups, calculates the Chi-Square value of each group, and then applies machine learning to perform classification. SANTaClass [16] extracts keywords based on common substrings across flows. ProWord [19] partitions payloads and identifies keyword boundaries based on statistical models. However, they do not address the inference of single-bit protocol fields. Instead, each group needs to have high degrees of freedom (e.g., one byte [9], [12], [16] or four bits long [3], [8]) for statistical features to be characterizable.

III. ProGraph Overview

ProGraph is a tool that infers the message formats of a network protocol by examining packet payloads. Our insight is that there exists *intra-packet dependency* across the actual values of different portions of a packet payload. Even though a field may take multiple legitimate values defined by a protocol, its actual value appearing in a packet payload is related to the current values of other fields. For example, DNS uses bit 16 to indicate whether a packet is a request (value 0) or a response (value 1). If the packet is a response, DNS uses bit 24 of the response to indicate whether a DNS server supports recursive queries. Thus, if bit 16 is 1 (response), then both 0 and 1 are legitimate values of bit 24; otherwise, if bit 16 is 0 (request), then the only legitimate value of bit 24 is 0 as it is not used by a request.

ProGraph exploits intra-packet dependency in its design. It takes a training set containing the payloads of multiple packets from a target network protocol as the input. It then constructs a *graphical model* to characterize intra-packet dependency, so as to (1) identify the set of legitimate values associated with each portion of a payload, and (2) identify how the value of a portion changes with respect to the value of another portion. ProGraph uses various techniques in graph theory and information theory to iteratively refine the graphical model. The output graphical model defines the formats of fields and the dependencies among them.

The graphical model can be used by traffic classification applications, such as deep packet inspection (DPI). For example, in DPI, we can classify protocol types of network traffic by inspecting if a packet matches the graphical model. If so, the packet belongs to the protocol characterized by the graphical model. With additional information, we can also determine the task carried out by the packet within the protocol.

Assumptions: ProGraph operates on a per-packet basis. We assume that protocol information is often embedded in the first few bytes of packet payloads [10], so ProGraph can still infer protocol formats even if a protocol encrypts sensitive user data.

However, not all packets contain protocol information. For example, HTTP only puts the HTTP header in the first packet of a request. ProGraph can tolerate the existence of packets without protocol information in the training set. It regards such packets as noisy packets and automatically excludes them during the graphical model construction. While we focus on per-packet inference, the same methodology applies to per-flow inference as well.

IV. Graphical Model Construction

A. Model Definitions

Table I summarizes the major notation used by the model in this paper. We consider payloads of multiple packets, such that each payload consists of a sequence of bits or bytes (e.g., the first 128 bits, or 16 bytes, of each packet payload). We divide each packet payload into a sequence of variable-size *chunks*, each of which corresponds to a consecutive portion of bits (or bytes) at the same offsets of all packet payloads (e.g., a chunk may represent bits 16 to 20 of every packet payload). A chunk is of *variable size*: initially, a chunk is formed by one bit (or byte), and the adjacent chunks may be merged depending on their values, as explained in the following discussion.

ProGraph takes a training set $\mathcal{T}$ of multiple packet payloads obtained from a protocol. It constructs a graphical model that is represented as a weighted directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ denotes a set of nodes and $\mathcal{E}$ denotes a set of edges. Each node $u \in \mathcal{N}$ represents a chunk[1]. Let $\mathcal{T}(u)$ be the set of values of node u observed in the training set $\mathcal{T}$. However, a training set may include some noisy packets that do not belong to the protocol. Thus, each node u is also associated with a set of legitimate values $\mathcal{V}_N(u)$ actually defined by the protocol. If $\mathcal{T}$ is noise-free, we have $\mathcal{T}(u) = \mathcal{V}_N(u)$; otherwise, we have $\mathcal{V}_N(u) \subset \mathcal{T}(u)$. Each directed edge $\langle u, v \rangle \in \mathcal{E}$ (where $u, v \in \mathcal{N}$) describes the dependency of the legitimate values of node v on those of node u. We associate each edge with a *weight* to quantify the dependency; a higher weight implies stronger dependency. We will discuss how to determine the weights and edge directions in Section IV-C1. Each edge $\langle u, v \rangle \in \mathcal{E}$

[1]The terms "node" and "chunk" are used interchangeably in this paper when there is no ambiguity.

TABLE I
MAJOR NOTATION USED IN THE PAPER.

Notation	Description
$\mathcal{T}$	training set
$\mathcal{N}$	set of nodes in the graphical model
$\mathcal{E}$	set of edges in the graphical model
$\mathcal{G}$	graphical model $\mathcal{G} = (\mathcal{N}, \mathcal{E})$
$\langle u, v \rangle$	edge from node u to node v, where $u, v \in \mathcal{N}$
$\mathcal{T}(u)$	set of values of node u observed in $\mathcal{T}$
$\mathcal{V}_N(u)$	legitimate values for node $u \in \mathcal{N}$
$\mathcal{V}_E(\langle u, v \rangle)$	legitimate value pairs for edge $\langle u, v \rangle \in \mathcal{E}$
B	length of each payload in the training set
X_u	random variable for the value of node u
$P(X_u)$	probability distribution of X_u
$P(X_u, X_v)$	joint distribution of X_u and X_v
$P(X_v \mid X_u)$	conditional distribution of X_v given X_u
$H(X_u)$	entropy of X_u
$H(X_v \mid X_u)$	conditional entropy of X_v given X_u
$d_{i,j}$	density of the single cluster composed of nodes $u_i, u_{i+1}, \cdots, u_j$
D_j	maximum sum of densities of all clusters induced by nodes $u_1, u_2, \cdots, u_j$
$\mathcal{I}_u$	set of incoming clusters of node u
$\mathcal{O}_u$	set of outgoing clusters of node u
$C^I_{u,v}$	number of distinct elements in either $\mathcal{I}_u$ or $\mathcal{I}_v$, but not both
$C^O_{u,v}$	number of distinct elements in either $\mathcal{O}_u$ or $\mathcal{O}_v$, but not both

is associated with a set of legitimate value pairs $\mathcal{V}_E(\langle u, v \rangle)$, such that a value pair $(x, y) \in \mathcal{V}_E(\langle u, v \rangle)$ if $x \in \mathcal{V}_N(u)$, $y \in \mathcal{V}_N(v)$, and y is a function of x (i.e., the value y is used due to the value x).

Figure 1 shows a graphical model for DNS packet payloads from bit 16 to bit 31. Bits 18:19:20 form a chunk (node) that corresponds to a part of the opcode field of DNS, while bits 29:30:31 form another chunk (node) that corresponds to a part of the response code field. Each of other chunks (nodes) is a single bit and corresponds to a single-bit flag in DNS. The figure details the legitimate values of all nodes, as well as the legitimate value pairs of edge from bits 18:19:20 to bits 29:30:31. For example, if bits 18:19:20 have value 0, then bits 29:30:31 can take any legitimate value; however, if bits 18:19:20 have value 4 or 5, then bits 29:30:31 can only take value 0 or 5. Note that the graphical model does not necessarily match the exact protocol format. For example, DNS uses bits 17:18:19:20 as the opcode and bits 28:29:30:31 as the response code. However, our model partitions bit 17 and bit 28 as individual nodes because they always take value 0. Also, even though bits 18:19:20 and bits 29:30:31 can have many possible legitimate values as stated in the DNS specification, only a few of them appear in practice, and our model only includes those that appear in our training set.

Our current graphical model has two open issues. First, a protocol may contain variable-length fields, whose lengths differ across packets. In the case, ProGraph partitions a long field into multiple chunks that are dependent on each other. Second, a field may be shifted to start at different offsets of a packet. Shifted fields cause some chunks to represent multiple fields in different packets. In this case, ProGraph includes all legitimate values of the fields in those chunks.

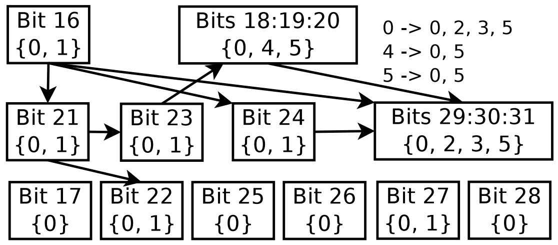


Fig. 1. Graphical model for the DNS packet payloads from bit 16 to bit 31.

Despite the presence of variable-length and shifted fields, we find that ProGraph remains accurate in characterizing inter-packet dependency and classifying traffic, as shown in our evaluations (see Section VI).

B. Overview of Model Construction

Algorithm 1 summarizes the procedure of the graphical model construction of ProGraph. The inputs are the training set $\mathcal{T}$ of packet payloads of a target protocol and the length B (in units of bits or bytes) of each payload considered by the model. ProGraph follows a bottom-up design. Specifically, it starts with a number of small chunks of size one unit (in one bit or one byte), and builds an initial graph with B nodes and no edges (Line 1). ProGraph iteratively updates G, the set $\mathcal{V}_N(u)$ of legitimate values for every node $u \in \mathcal{N}$, and the set $\mathcal{V}_E(\langle u, v \rangle)$ of legitimate value pairs for every edge $\langle u, v \rangle$ (Lines 2-14), based on the statistical distributions of the observed values in the training set. Let X_u be the random variable of the observed values in $\mathcal{T}(u)$ of node u, $P(X_u)$ be the probability distribution of the observed values of node u, and $P(X_u, X_v)$ denote the joint probability distribution of the observed value pairs of nodes u and v. In each iteration, ProGraph computes $P(X_u)$ for every node u and $P(X_u, X_v)$ for every pair of nodes u and v (Lines 3-4). It updates the topology of $\mathcal{G}$ by adding/deleting edges and merging nodes (Lines 5-6). It also recomputes $P(X_u)$ and $P(X_u, X_v)$ based on the updated topology (Line 10) and updates the set of edges (Line 11). Finally, it adds legitimate values and value pairs for nodes and edges, respectively (Line 12), and filters noisy packets from the training set $\mathcal{T}$ (Line 13). ProGraph repeats the above steps until the topology of $\mathcal{G}$ has no further update (Lines 7-9). It will output $\mathcal{G}$, $\{\mathcal{V}_N(u) \mid u \in \mathcal{N}\}$, and $\{\mathcal{V}_E(\langle u, v \rangle) \mid \langle u, v \rangle \in \mathcal{E}\}$. We elaborate the details in the following subsections.

C. Updating the Graph Topology

There are two main operations of updating the topology of $\mathcal{G}$: (1) adding/deleting edges and (2) merging nodes.

1) Adding/Deleting Edges: ProGraph adds (resp. deletes) edges to $\mathcal{G}$ between node pairs if they have high (resp. low) dependency, which we quantify based on information theory measures. We consider the *entropy* of X_u, which measures the uncertainty of X_u over all values of node u observed in the training set. It is given by:

$$H(X_u) = -\sum_{x \in \mathcal{T}(u)} P(X_u = x) \log_2 P(X_u = x).$$

Algorithm 1 Graphical Model Construction of ProGraph

```
Inputs: Training set $\mathcal{T}$; Payload length $B$
Outputs: $\mathcal{G}=(\mathcal{N},\mathcal{E})$; $\{\mathcal{V}_N(u) \mid u\in\mathcal{N}\}$; $\{\mathcal{V}_E(\langle u,v\rangle) \mid \langle u,v\rangle\in\mathcal{E}\}$
 1: Create $\mathcal{G}$ with $B$ nodes and no edges; set $\mathcal{V}_N(.)$ and $\mathcal{V}_E(.)$ empty
 2: while true do
 3:     Compute $P(X_u)$ for each node $u \in \mathcal{N}$
 4:     Compute $P(X_u, X_v)$ for each pair of nodes $u, v \in \mathcal{N}$
 5:     Add/delete edges among nodes
 6:     Merge nodes
 7:     if no edge is updated and no nodes are merged then
 8:         return $\mathcal{G}$, $\{\mathcal{V}_N(u) \mid u \in \mathcal{N}\}$, $\{\mathcal{V}_E(\langle u,v\rangle) \mid \langle u,v\rangle\in\mathcal{E}\}$
 9:     end if
10:     Recompute $P(X_u)$ and $P(X_u, X_v)$
11:     Update the edges as in Line 5
12:     Add values to $\{\mathcal{V}_N(u) \mid u \in \mathcal{N}\}$ and $\{\mathcal{V}_E(\langle u,v\rangle) \mid \langle u,v\rangle\in\mathcal{E}\}$
13:     Filter noisy packets in $\mathcal{T}$
14: end while
```

Also, for a pair of nodes u and v, we consider the *conditional entropy* of X_v given X_u, which measures the expected uncertainty of X_v conditioning on X_u and is given by:

$$H(X_v|X_u) = \sum_{x\in\mathcal{T}(u)} P(X_u = x)H(X_v|X_u = x).$$

We now quantify the dependency between nodes u and v using the *information gain ratio (IGR)*, which is defined as:

$$\text{IGR} = \frac{H(X_v) - H(X_v|X_u)}{\min\{H(X_v), H(X_u)\}}.$$

The numerator $H(X_v) - H(X_v|X_u)$ is the *mutual information*, which measures the expected distance between the distributions $P(X_v)$ and $P(X_v|X_u)$. It can be shown that $0 \leq H(X_v) - H(X_v|X_u) = H(X_u) - H(X_u|X_v) \leq \min\{H(X_u), H(X_v)\}$. Thus, the IGR can be viewed as the normalized mutual information ranging from 0 to 1 inclusively.

We add (resp. delete) an edge between nodes u and v if their dependency, which is quantified as the IGR, is above (resp. below) a threshold. We select the threshold using an adaptive approach, which we discuss in Section IV-E. The edge weight is set to be the IGR.

Every edge is directed. We use the following heuristic to determine the edge direction. We direct an edge from the node with fewer legitimate values to the node with more legitimate values (e.g., an edge is directed from bits 18:19:20 to bits 29:30:31); if there is a tie, we direct an edge from the node at a lower offset to the node at a higher offset (e.g., an edge is directed from bit 16 to bit 24). The edge directions will determine how we identify the fields for task classification (see Section V). Note that based on our heuristic, $\mathcal{G}$ is acyclic.

2) Merging Nodes: ProGraph merges strongly correlated nodes whose legitimate values depend on each other. The goal is to recover the fields from the partitioned chunks. For example, in Figure 1, we merge bits 18, 19, and 20 as they belong to the opcode field of DNS.

Our merge operation is motivated by three principles. First, we merge adjacent chunks in a payload (e.g., bit 18 and bit 19 are adjacent 1-bit chunks), since a field typically comprises contiguous bits or bytes. Second, if two nodes belong to the same protocol field, then they should be connected by an edge indicating that they have high dependency. Third, the two nodes should be connected to, and hence have high dependency with, the same (or similar) set of neighbors.

To merge nodes, ProGraph analyzes $\mathcal{G}$ at two levels: clustering analysis and neighboring analysis, as described below.

Clustering analysis: Let $|\mathcal{N}|$ denote the number of nodes in $\mathcal{G}$. We sort the nodes of $\mathcal{G}$ in the order of their bit offsets in a payload, and denote the ordered sequence of nodes by $u_1, u_2, \cdots, u_{|\mathcal{N}|}$. We partition the nodes into different clusters (subsets), each of which consists of nodes corresponding to adjacent chunks in a payload. Our observation is that nodes of the same field likely belong to the same cluster and are connected by edges with high weights. We define the *density* of a cluster as the ratio of the sum of edge weights in the cluster to the number of node pairs in the cluster. Our goal is to find a partitioning solution (i.e., a set of clusters) that maximizes the sum of the densities of the clusters that cover all $|\mathcal{N}|$ nodes.

Let $d_{i,j}$ (where $1 \leq i \leq j \leq n$) be the density of the single cluster composed of nodes $u_i, u_{i+1}, \cdots, u_j$, and we assume $d_{i,i} = 0$. Also, let D_j (where $1 \leq j \leq |\mathcal{N}|$) be the maximum sum of densities of all clusters induced by nodes $u_1, u_2, \cdots, u_j$. We can express D_j as a recursive equation:

$$D_j = \max_{1\leq i\leq j}\{D_{i-1} + d_{i,j}\}, \text{ where } 1 \leq j \leq |\mathcal{N}|.$$

We set $D_0 = 0$. The maximum sum of densities of the clusters that cover all $|\mathcal{N}|$ nodes is $D_{|\mathcal{N}|}$. The above equation can be solved via dynamic programming. To identify all clusters, we first determine the cut-point index (call it l) that maximizes $D_{|\mathcal{N}|}$, i.e., $l = \arg\max_{1\leq i\leq j}\{D_{i-1} + d_{i,j}\}$. The set of nodes $\{u_l, u_{l+1}, \cdots, u_{|\mathcal{N}|}\}$ forms a cluster. We further identify the next cut-point index that maximizes D_{l-1}. We repeat this process until the cut-point index $l = 1$.

Neighboring analysis: After identifying all clusters, ProGraph further considers the neighboring properties of nodes. Specifically, if node u has an edge from (resp. to) another node v and v belongs to a cluster that does not contain u, we say that the cluster is an *incoming cluster* (resp. *outgoing cluster*) of u. Let $\mathcal{I}_u$ and $\mathcal{O}_u$ be the sets of incoming clusters and outgoing clusters of node u, respectively. Let $C^I_{u,v} = |\mathcal{I}_u \cup \mathcal{I}_v - \mathcal{I}_u \cap \mathcal{I}_v|$ and $C^O_{u,v} = |\mathcal{O}_u \cup \mathcal{O}_v - \mathcal{O}_u \cap \mathcal{O}_v|$, where $C^I_{u,v}$ (resp. $C^O_{u,v}$) denotes the number of distinct elements that reside in either $\mathcal{I}_u$ or $\mathcal{I}_v$ (resp. either $\mathcal{O}_u$ or $\mathcal{O}_v$), but not both. Intuitively, if $C^I_{u,v}$ and $C^O_{u,v}$ are small, then the neighboring clusters of nodes u and v are similar. We say that nodes u and v have *similar neighbors* if both $C^I_{u,v} \leq 1$ and $C^O_{u,v} \leq 1$, where the threshold value 1 is empirically chosen based on our traces (see Section VI).

Putting it all together: We merge two nodes u and v if and only if all the following conditions hold: (1) both correspond to the adjacent chunks in a payload; (2) both are in the same cluster from our clustering analysis; and (3) both have similar neighbors from our neighboring analysis. We recompute the probability distributions after merging (Line 10

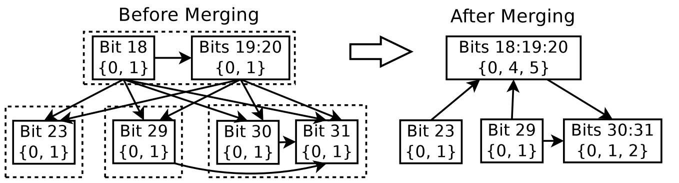


Fig. 2. Example of the merging operation for the DNS packet payloads. Each dotted rectangle identifies a cluster of nodes. The merge operation will merge bit 18 and bits 19:20, as well as merge bit 30 and bit 31.

of Algorithm 1). Since the probability distributions are updated, we add/delete all edges again as stated in Section IV-C1 (Line 11 of Algorithm 1).

Example: Figure 2 illustrates an example of the merging operation for the DNS packet payloads. The figure is taken from part of Figure 1 in the middle of the graphical model construction. The clustering analysis forms multiple clusters (in dotted rectangles), in which bit 18 and bits 19:20 form one cluster, bit 30 and bit 31 form another cluster, and the other clusters are all single nodes. In the cluster of bit 18 and bits 19:20, both nodes have no incoming cluster and identical outgoing clusters (i.e., bit 23, bit 29, and the cluster of bits 30 and 31). Thus, we can merge bit 18 and bits 19:20. Similarly, we can also merge bits 30 and 31, as neither of them has any outgoing cluster and their incoming clusters only differ by one (bit 29 is an incoming cluster of bit 31 but not bit 30). Regarding the edge directions, we follow the heuristic in Section IV-C1. Before merging, all nodes have two values, so every edge is directed from the node with a lower offset to the one with a higher offset. After merging, there are three values in bits 18:19:20. This reverses the directions of the edges between bits 18:19:20 and bit 23 and between bits 18:19:20 and bit 29.

3) Chunks with Constant Values: A chunk may take a constant value (i.e., the value at the same position of all payloads in the training set is a constant). Its corresponding node has no edge connecting any other node in the graphical model, because its value is independent of the values of other chunks. For example, in Figure 1, bits 17 and 28 (which belong to the opcode field and the response code field, respectively) always take value 0, and they form individual nodes in the graph. We call such individual nodes as *constant nodes*.

If the constant nodes reside at the beginning or the end of a field, the other nodes of the same field can still be merged (e.g., bits 29:30:31 of the response code field). In some cases, a constant node may appear in the middle of a field, thereby preventing other nodes of the same field from being merged. For example, the opcode field of DNS comprises bits 17:18:19:20 (see Figure 1), but in practice, it takes only three values 0, 4, and 5. Thus, both bit 17 and bit 19 have the constant value 0, but bit 19 separates bit 18 and bit 20. To address the problem, ProGraph applies a heuristic to merge the constant nodes with other non-constant nodes before running the merge operation in Section IV-C2. Consider two nodes u and v whose corresponding chunks are non-adjacent. If there exists an edge connecting nodes u and v and all chunks between u and v correspond to constant nodes, then we merge all constant nodes into either node u or node v (the results will be unaffected). For example, referring to Figure 2, bit 19 is merged with bit 20 because it takes a constant value 0 and bits 18 and 20 are connected by an edge. Note that merging constant nodes into a non-constant node does not alter the dependency with the remaining nodes in the graph.

D. Extracting Legitimate Values and Pairs

We now extract legitimate values for each node $u \in \mathcal{N}$ and legitimate value pairs for each edge $\langle u, v\rangle \in \mathcal{E}$ (Line 12 of Algorithm 1). One challenge is that the training set may include *noisy* packets that do not belong to the target protocol. Our goal is to exclude all noisy packets from the training set.

We assume that for each node, its values due to the noisy packets occupy only a small fraction of the packet payloads in the training set, so ProGraph can leverage the probability distributions to extract legitimate values and value pairs. ProGraph employs a threshold-based approach. For any value x associated with node u in the training set $\mathcal{T}$ (i.e., $x \in \mathcal{T}(u)$), if $P(X_u = x)$ of value x exceeds a threshold, then we include x into $\mathcal{V}_N(u)$. Similarly, for any value pair (x, y) associated with edge $\langle u, v\rangle$, if both $P(X_u = x|X_v = y)$ and $P(X_v = y|X_u = x)$ exceed the respective thresholds for the distributions, we include x into $\mathcal{V}_N(u)$, y into $\mathcal{V}_N(v)$, and (x, y) into $\mathcal{V}_E(\langle u, v\rangle)$. We select the thresholds adaptively based on the distributions, as described in Section IV-E.

We remove noisy packets from the training set (Line 13 of Algorithm 1) after we extract the legitimate values and value pairs. Specifically, for a given packet, if there exists a chunk whose value is not treated as a legitimate value, or there exists a pair of chunks such that the corresponding nodes are connected by an edge but the value pair is not treated as a legitimate value pair, then the packet is regarded as a noisy packet and is removed from the training set.

E. Adaptive Threshold Selection

ProGraph employs thresholds to differentiate values in two cases: (1) in Section IV-C1, we differentiate node pairs with strong and weak dependencies to decide whether to add or delete edges; and (2) in Section IV-D, we differentiate legitimate values and value pairs from those that belong to noisy packets. In both cases, our goal is to select an appropriate threshold that partitions a set of values into two subsets.

We formulate the threshold selection problem as follows. Consider a multi-set of n floating-point values in the range $[0, 1]$. In the case of adding/deleting edges (Section IV-C1), the floating-point values correspond to the IGRs of node pairs; in the case of extracting legitimate values and value pairs, the floating-point values correspond to probabilities of observed values or value pairs in the training set (see Section IV-D). However, it is non-trivial to determine an appropriate threshold, as we iteratively refine the distributions $\{P(X_u)\}$ and $\{P(X_u, X_v)\}$. Thus, we need an *adaptive* approach to determine the most appropriate threshold.

ProGraph determines the appropriate threshold for the n floating-point values using the minimum-description-length

(MDL) technique [1], as it fulfills our design goals of simple parameterization and fast execution. We first transform the input floating-point values into integers by multiplying them with a large value (e.g., 10^6) and rounding them off to the nearest integers. We sort the n integers in descending order and denote them by $a_1 \geq a_2 \geq \dots a_n$. The MDL technique searches for a cut-point k (where $1 \leq k \leq n$) to partition the n sorted integers into two subsets $\{a_1, a_2, \dots, a_k\}$ and $\{a_{k+1}, a_{k+2}, \dots a_n\}$, such that the total description length for the two subsets is minimized. Specifically, we calculate the means of the integers in both subsets (denote them by μ_1 and μ_2) and the difference of each integer from the mean in the same subset. The total description length is the total number of bits to represent the means and all differences:

$$\lceil \log_2(\mu_1) \rceil + \sum_{1 \leq i \leq k} \lceil \log_2 |a_i - \mu_1| \rceil + \lceil \log_2(\mu_2) \rceil + \sum_{k < i \leq n} \lceil \log_2 |a_i - \mu_2| \rceil.$$

The MDL technique returns the corresponding floating-point value of a_{k^*} as the threshold, where k^* minimizes the above expression.

Note that the threshold selection does not require any manually specified parameters. The complexities of the sorting and the cut-point search are $O(n \log n)$ and $O(n)$, respectively. In adding/deleting edges, n is the number of node pairs in $\mathcal{G}$ (i.e., $n = O(|\mathcal{N}|^2)$); in extracting legitimate values and value pairs, n is the number of observed values and value pairs in the training set. Our evaluations show that the threshold selection can be done quickly (see Section VI).

V. Application

After we construct a graphical model for a target protocol from ProGraph, we can use it to classify traffic at two levels: we can first check if a packet belongs to the target protocol; if so, we can identify the specific task associated with the packet.

Protocol classification: Given an input packet, we traverse every node and its outgoing edges. For each traversed node, we check if the corresponding chunk value and value pairs of the input packet belong to the sets of legitimate values and value pairs, respectively. If all nodes and edges pass the checking, then we associate the input packet with the target protocol. The complexity is $O(|\mathcal{N}| + |\mathcal{E}|)$, where $|\mathcal{N}|$ and $|\mathcal{E}|$ are the total numbers of nodes and edges in $\mathcal{G}$, respectively.

Task classification: To identify the specific task of a packet in a protocol, we propose a heuristic, in which we focus on the set of nodes with zero in-degree and non-zero out-degree, and we call such nodes as *source nodes*. According to our definition of edge direction (see Section IV-C1), a source node is more likely to be a field for task identification based on two observations. First, a single value of the task identification field determines multiple values of other protocol fields, so a node with fewer legitimate values is more likely used for task identification. Second, if two protocol fields have the same number of legitimate values, the task identification field likely appears at a lower offset. Both of these observations guide us to choose the source nodes for task classification.

Note that the source nodes are not the unique choices for task classification. For example, in DNS, we may either use bit 16 to distinguish request and response packets, or use bits 18:19:20 to classify the packets into three types: query, update, and notify. Nevertheless, our evaluations show that the source nodes provide useful information for task classification in some common protocols (see Section VI).

VI. Evaluation

A. Datasets

We run ProGraph on two real-world IP traces (denoted by Trace 1 and Trace 2) collected from the 3G cellular networks in two different cities in mainland China. Trace 1 has around 4.1 billion packets with 2.2TB of traffic in one day of November 2010. Trace 2 has around 450 million packets with 480GB of traffic in 15 heavy-loaded hours over three days (5 hours per day) of November 2013. We focus on the data-plane IP traffic, while the 3G-specific control-plane traffic is not our focus here. Our evaluations only consider the first 16 bytes of the payload of an IP packet, which suffice for traffic classification [10]. We do not examine the remaining payload and ensure that the privacy is preserved.

We consider four application-layer protocols in the evaluation: HTTP, DNS, BitTorrent, and WeChat. We use HTTP, DNS, and BitTorrent for our validations, as their specifications are available and provide the ground truths. On the other hand, WeChat is one of the most popular mobile messaging applications worldwide [5], but its specifications are proprietary and unknown to the public. We use WeChat to show how ProGraph is applied in traffic classification.

Preparation of training sets: For each protocol, we prepare a training set for the graphical model construction. For HTTP, DNS and BitTorrent, we leverage their protocol specifications as ground truths and construct the training sets based on our real-world traces. The training sets of HTTP and DNS come from Trace 2, while that of BitTorrent comes from Trace 1. We first extract all packets of the protocols from the respective traces based on the protocol specifications. We then uniformly sample 1% of the packets as the training sets. Thus, each training set only covers a subset of the protocol packets, and this enables us to verify the robustness of ProGraph. We regard the training sets as *pure*, in the sense that they do not contain other noisy packets that belong to other protocols. Later in our experiments, we inject noisy packets to the pure training sets. Our training sets of HTTP, DNS, and BitTorrent contain $200,596$, $98,458$, and $60,564$ packets, respectively.

However, building the training set for WeChat is challenging due to its lack of specification. Thus, we capture WeChat traces in a controlled environment. Specifically, using a mobile device that has WeChat installed, we perform various WeChat operations and capture traffic traces. We observe that WeChat traffic comprises both TCP and UDP traffic. In our experiments, we focus on the UDP packets destined to the server `voip.weixin.qq.com`. Such UDP packets correspond to real-time chatting functionalities of WeChat. We use them as the training set, which comprises $4,077$ packets. We do not assume that the training set is pure, but ProGraph can effectively remove noisy packets as shown in our experiments.

B. Setup

Testbed: We implement ProGraph, including the algorithms for model construction and classification, in C++ with 2000+ lines of code. We deploy ProGraph on a server equipped with 2.40GHz CPU. The server runs Linux 2.6.32. We compile our code using g++ 4.3 with the -O3 option.

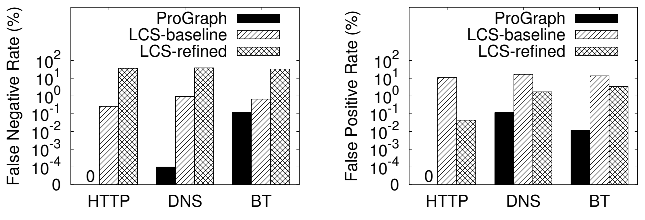
Metrics: We mainly evaluate the accuracy and execution speed of ProGraph. We consider the following metrics:

- *False negative rate*: The classification of a packet is a *false negative* if it belongs to the target protocol but is misclassified as not belonging to the protocol. The false negative rate is the ratio of the number of false negatives to the total number of packets that actually belong to the target protocol.
- *False positive rate*: The classification is a packet is a *false positive* if it does not belong to the target protocol but is misclassified as belonging to the protocol. The false positive rate is the ratio of the number of false positives to the total number of packets that do not belong to the protocol.
- *Throughput*: It is the total IP payload size that has been processed by ProGraph divided by the execution time. To eliminate disk seek overheads, we load all packets (first 16 bytes each) into main memory before measurements.

C. Performance Results

We present performance results of ProGraph using HTTP, DNS, and BitTorrent, all of which can be validated through their specifications. In particular, both HTTP and BitTorrent operate at the byte level, while DNS operates at the bit level. After constructing the graphical models, we run protocol classification (see Section V) on all packets from the combined traces of Trace 1 and Trace 2.

Experiment 1 (Accuracy of protocol classification): We compare the accuracy of ProGraph on protocol classification with that of the longest common subsequence (LCS) approaches, whose idea is to identify the longest subsequence of elements (where the elements need not be contiguous) appearing in two packet payloads. Since there can be many LCSes, the LCSes are usually selected with two thresholds [12]: (1) the length of a selected LCS should exceed a minimum length threshold; and (2) the fraction of packets that contain a selected LCS should exceed a minimum coverage threshold. The classification is to check whether a packet payload matches the selected LCSes. We consider two variants of LCS-based classification: (1) *LCS-baseline*, which selects the LCSes that pass the minimum length and minimum coverage thresholds; (2) *LCS-refined*, which refines the selected LCSes with the subsequence handling heuristic proposed by SANTaClass [16]: if an LCS is a subsequence of another LCS, the one that appears in a larger number of occurrences is retained; if both LCSes have the same number of occurrences, the longer one is retained. We do not consider other refining techniques in [16] because they are used for text-based protocols and fail to work for DNS and BitTorrent in our evaluation.



(a) False negative rate (b) False positive rate

Fig. 3. Experiment 1 (Accuracy of protocol classification).

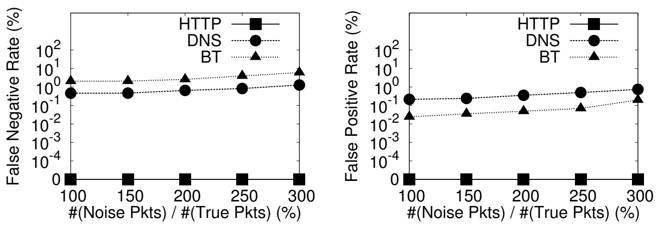
In our implementations of the LCS approaches, we extract the first 64 bytes of packets [12] (as opposed to first 16 bytes in ProGraph). We store and match the selected LCSes using a finite state machine [16]. We set the minimum length threshold as 4 bytes and the minimum coverage threshold as 5%, which give the best results in our evaluation. Both ProGraph and the LCS approaches operate on the pure training sets.

Figure 3 presents the results. LCS-baseline has a low false negative rate ($< 1\%$), but its false positive rate is above 10% for all three protocols. LCS-refined significantly reduces the false positive rate to less than 1%, yet it also incurs a false negative rate of above 30%. The reason is that the subsequence handling heuristic may drop many important LCSes. For example, both "`GET /`" and "`GET http://`" may be the LCSes in HTTP, and the former is a subsequence of the latter. Since the LCS "`GET http://`" appears more frequently in our training set, the LCS "`GET /`" is dropped. However, this incurs many false negatives containing the LCS "`GET /`".

In contrast, ProGraph has low error rates: for HTTP, it achieves perfect classification; for DNS, its false negative rate is below 0.0001% and its false positive rate is around 0.1%; for BitTorrent, its false negative rate and false positive rate are around 0.1% and 0.01%, respectively. ProGraph associates legitimate values to the chunks so as to accurately characterize the payload features. For example, it includes both subsequences "`GET /`" and "`GET http://`" as the legitimate values, so as to control the false negative rate. In addition, ProGraph controls the false positive rate by ensuring that the value of a subsequence is legitimate for specific chunks.

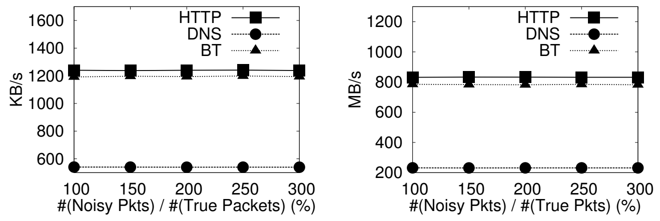
Experiment 2 (Robustness to noisy training sets): We show that the ProGraph achieves accurate classification even in the presence of the noisy training sets. To construct a noisy training set for a protocol, we uniformly select from our traces a fraction of packets that do not belong to the target protocol into the pure training set. We define the *noise ratio* as the ratio of the number of noisy packets to the number of packets in the pure training set, and we vary the noise ratio from 100% to 300%. ProGraph generates the graphical models of all protocols using the noisy training sets, and performs protocol classification on all packets in the traces. Although we inject a huge number of noisy packets, we expect that their values only occupy a small fraction in packet payloads (see Section IV-D).

Figure 4 shows the results. For HTTP, ProGraph still has zero error rates. However, the noisy training sets increase the



(a) False negative rate (b) False positive rate

Fig. 4. Experiment 2 (Robustness to noisy training sets).



(a) Model Construction (b) Classification

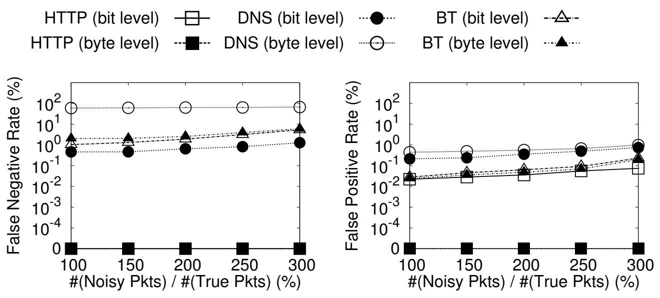
Fig. 5. Experiment 3 (Throughput).

error rates for DNS and BitTorrent. For example, when the noise ratio is 300%, the false negative rates for DNS and BitTorrent increase to 1.27% and 6%, respectively (see Figure 4(a)). The reason is that the noisy packets introduce new values for the chunks, making the model construction remove the legitimate values that appear with low probabilities. The false positive rates for DNS and BitTorrent also increase to 0.7% and 0.2%, respectively (see Figure 4(b)), mainly because the graphical models wrongly include noisy values as the legitimate values. Nevertheless, ProGraph still achieves low error rates in general (less than 6% in all cases).

Experiment 3 (Throughput): We measure the throughput of ProGraph in both model construction and classification, using the noisy training sets. Figure 5(a) shows the throughput for model construction. We see that the throughput of HTTP and BitTorrent exceeds 1100KB/s, while that of DNS is around 540KB/s. In particular, the threshold selection (see Section IV-E) only accounts for 10% of the overall running time (not shown in the figure). Model construction is an offline procedure and is expected to be less frequently executed. Nevertheless, our throughput values are considered to be high, compared to those of the LCS-based approaches, which achieve only 10-20KB/s for model construction. ProGraph also outperforms some existing offline traffic classification techniques, such as ProDecoder [17] (0.31KB/s) and ProWord [19] (10-20KB/s), under similar hardware configurations.

Figure 5(b) shows the classification throughput. The throughput of both HTTP and BitTorrent is nearly 800MB/s, while that of DNS is only 230MB/s. DNS has a slower speed because it operates at the bit level and its graphical model has much more nodes and edges than the other two.

Experiment 4 (Comparisons of bit-level and byte-level classifications): We evaluate the impact of the operational



(a) False negative rate (b) False positive rate

Fig. 6. Experiment 4 (Comparison of bit-level and byte-level classifications).

granularity on the accuracy of traffic classification. Figure 6 compares the error rates of both bit-level and byte-level classifications on HTTP, DNS, and BitTorrent. As expected, ProGraph achieves effective bit-level classification for HTTP and BitTorrent, and the error rates are close to those of byte-level classification. In contrast, the accuracy of byte-level classification for DNS is significantly worse than that of bit-level classification, and has at least 60% of false negative rate in all cases. The reason is that byte-level classification misses many bit-level protocol fields defined in DNS.

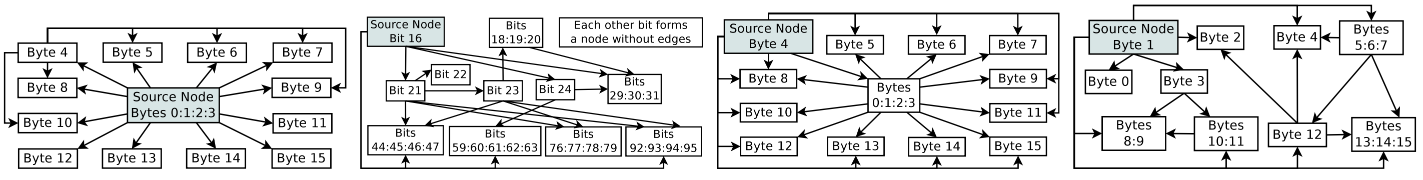
Since bit-level classification also provides accurate results for the protocols that operate at the byte-level granularity, in practice, we can always configure ProGraph to work at the bit level for the unknown protocols, with trade-off of having lower throughput (see Experiment 3).

D. Protocol Formats

We examine how ProGraph facilitates task classification (see Section V). Figure 7 present the constructed graphical models for the four protocols. We summarize the key findings below.

Source nodes for task classification: We identify the source nodes from different protocols for task classification (the shaded nodes in Figure 7). For HTTP, the source node comprises the first four bytes. It distinguishes the HTTP response (“HTTP”) and various HTTP request methods (e.g., “GET ”, “POST”, “CONN”). For DNS, the source node is bit 16, which indicates if a packet is a request or a response. For BitTorrent, the source node is byte 4, which distinguishes the handshake messages (value “T”) and ten types of messages exchanged between peers (values from 0 to 9) [2].

For WeChat, we have tested both bit-level and byte-level classifications, and obtained identical results. Thus, we present the model based on byte-level classification for brevity (see Figure 7(d)). We identify around 1.36M WeChat UDP packets with 810MB of application payloads in our traces. The source node is byte 1, while all other nodes depend on the source node. We find that the source node has four legitimate values. By correlating them with the operational logs when we capture the traces, we find that one value corresponds to heartbeat messages, two values correspond to full-duplex VoIP chats, and one value correspond to half-duplex walkie-talkie chats. Our results show that ProGraph remains effective even when the protocol specification is unavailable.



(a) HTTP (byte-level) (b) DNS (bit-level) (c) BitTorrent (byte-level) (d) WeChat UDP (byte-level)

Fig. 7. Graphical models for HTTP, DNS, BitTorrent, and WeChat.

Partitioned fields: Some fields, such as the HTTP Method field, have a variable length. ProGraph partitions a long field into multiple nodes. For example, ProGraph treats bytes 0:1:2:3 as a field, but the HTTP Method "CONNECT" will be partitioned as "CONN", "E", "C", and "T" in bytes 0:1:2:3, 4, 5, and 6, respectively. However, based on the first four bytes "CONN", ProGraph can still infer that it belongs to the CONNECT method.

Constant chunks for unused bytes/bits: A protocol may define a long field, but only set a portion of it. For example, DNS defines a field from bits 32 to 47 for the number of queries in a request, but the length is only stored in bits 44:45:46:47. ProGraph will identify a chunk for bits 44:45:46:47, and treat the remaining 12 bits as single-bit constant chunks.

Ignored dependencies: In some cases, two nodes should be dependent based on the protocol specifications, but ProGraph does not connect them with an edge because the protocol is implemented differently. For example, DNS employs bits 76:77:78:79 to count the number of authoritative servers in the response, so it should depend on bit 16. However, there is no edge between the chunks bit 16 to bits 76:77:78:79. We find that the two chunks have low dependency in our traces because most responses do not list the authoritative servers.

Unidentified fields: One limitation of ProGraph is that in some cases, it cannot merge the chunks that belong to the same field of a protocol. One such case is that some fields may carry randomly generated data. For example, the first 16 bits in DNS carry a random session ID, in which each bit has no dependency with any other bits. Another case is that the fields are of variable-length and shifted (see Section IV-A). In this case, the chunks are not merged. The unidentified fields do not compromise the effectiveness of ProGraph. As shown in Section VI-C, we can achieve high accuracy in protocol classification.

VII. Conclusions

We present ProGraph, a fine-grained protocol format inference tool that can operate on packet payloads at both byte-level and bit-level granularities. ProGraph leverages intra-packet dependency of packet payloads as the key feature, and characterizes intra-packet dependency via a graphical model. Extensive trace-driven evaluations show that ProGraph achieves high accuracy and remains robust in the presence of noisy training sets. ProGraph can serve as a critical preprocessing tool for many traffic classification applications in quality-of-service control, network security, and resource profiling.

Acknowledgments

This work was supported in part by: GRF CUHK413711 from the Research Grant Council of Hong Kong, 2012CB316303 and 2013CB329602 from the National Basic Research Program of China (973 Program), 61232010 from National Science Foundation of China, and 2012BAH46B04 from the National Key Technology R&D Program.

References

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of SIGMOD*, 1998.
[2] BitTorrent Specification. https://wiki.theory.org/BitTorrentSpecification.
[3] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli. Revealing Skype Traffic: When Randomness Plays with You. In *Proc. of SIGCOMM*, 2007.
[4] G. Bossert, F. Guihéry, and G. Hiet. Towards Automated Protocol Reverse Engineering using Semantic Information. In *Proc. of ASIACCS*, 2014.
[5] By the Numbers: 20 Amazing WeChat Statistics. http://expandedramblings.com/index.php/wechat-statistics/.
[6] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *Proc. of CCS*, 2007.
[7] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *Usenix Security Symposium*, 2007.
[8] A. Finamore, M. Mellia, M. Meo, and D. Rossi. KISS: Stochastic Packet Inspection Classifier for UDP Traffic. *IEEE/ACM Transactions on Networking*, 18(5):1505–1515, 2010.
[9] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS : Automated Construction of Application Signatures. In *SIGCOMM Workshop on MineNet*, 2005.
[10] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet Traffic Classification Demystified : Myths , Caveats , and the Best Practices. In *Proc. of CoNEXT*, 2008.
[11] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Proc. of NDSS*, 2008.
[12] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. In *Proc. of IMC*, 2006.
[13] A. W. Moore and D. Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Proc. of SIGMETRICS*, 2005.
[14] M. Roesch. Snort Lightweight Intrusion Detection for Networks. In *LISA*, 1999.
[15] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-Service Mapping for QoS : A Statistical Signature-based Approach to IP Traffic Classification. In *Proc. of IMC*, 2004.
[16] A. Tongaonkar, R. Keralapura, and A. Nucci. SANTaClass : A Self Adaptive Network Traffic Classification System. In *IFIP Networking Conference*, 2013.
[17] Y. Wang, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo. A Semantics Aware Approach to Automated Reverse Engineering Unknown Protocols. In *Proc. of ICNP*, 2012.
[18] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic Network Protocol Analysis. In *Proc. of NDSS*, 2008.
[19] Z. Zhang, Z. Zhang, P. P. C. Lee, Y. Liu, and G. Xie. ProWord : An Unsupervised Approach to Protocol Feature Word Extraction. In *Proc. of INFOCOM*, 2014.