Inferring Protocol State Machine from Network Traces: A Probabilistic Approach*

Yipeng Wang[13], Zhibin Zhang[1], Danfeng (Daphne) Yao[2], Buyun Qu[13], and Li Guo[1]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2] Department of Computer Science, Virginia Tech, Blacksburg, VA, USA
[3] Graduate University, Chinese Academy of Sciences, Beijing, China

Abstract. Application-level protocol specifications (i.e., how a protocol should behave) are helpful for network security management, including intrusion detection and intrusion prevention. The knowledge of protocol specifications is also an effective way of detecting malicious code. However, current methods for obtaining unknown protocol specifications highly rely on manual operations, such as reverse engineering which is a major instrument for extracting application-level specifications but is time-consuming and laborious. Several works have focus their attentions on extracting protocol messages from real-world trace automatically, and leave protocol state machine unsolved.
In this paper, we propose *Veritas*, a system that can automatically infer protocol state machine from real-world network traces. The main feature of *Veritas* is that it has no prior knowledge of protocol specifications, and our technique is based on the statistical analysis on the protocol formats. We also formally define a new model – probabilistic protocol state machine (P-PSM), which is a probabilistic generalization of protocol state machine. In our experiments, we evaluate a text-based protocol and two binary-based protocols to test the performance of *Veritas*. Our results show that the protocol state machines that *Veritas* infers can accurately represent 92% of the protocol flows on average. Our system is general and suitable for both text-based and binary-based protocols. *Veritas* can also be employed as an auxiliary tool for analyzing unknown behaviors in real-world applications.

Keywords: Protocol Model Inference and Analysis; Probabilistic Protocol State Machine; Network Security

1 Introduction

Detailed knowledge of protocol specifications is helpful in many network security applications, such as intrusion detection systems [16], vulnerability discovery [14], and protocol analyzers for Internet traffic monitoring [17]. Furthermore,

* This work is supported by the National Basic Research Program "973" of China (Grant No. 2007CB311100)

given a protocol specification, it is important for application fingerprinting [15] and mapping traffic to applications [7]. However, many network protocols, such as private and non-standard protocols, have no public protocol specifications available. Therefore, it is a crucial network security problem for Internet Service Providers (ISP) to find out these unknown protocol specifications. In the context of obtaining protocol specifications, inferring protocol state machines plays a more important role in practice. Generally, the target of protocol specification discovery concerns not only protocol message formats (i.e., the packet encapsulation mechanism), but also the protocol state machine. The protocol state machine is a finite state automaton illustrating the states in the protocol and their transitions (i.e., the state transition manner). Discovering message format is useful in identifying protocols in monitored network traffic and building intrusion detection systems; and discovering protocol state machine can depict the behavior of an application. Much previous work [5–9] was focused on extracting the protocol format information, without resolving any protocol state machine. However, there are a few exceptions [3, 4]. For example, Prospex [3] is an elegant solution for both protocol format and state machine inferencing, which is useful for malware analysis. Prospex's analysis is based on observing the dynamic execution of the program and thus requires the binary code.

Our paper provides a novel technique for inferring protocol state machine *solely* based on the real-world network trace of an application. There are several advantages associated with our approach. First, analyzing network traffic can be easily automated and requires less manual effort. The analysis does not require the distinction between client and server applications. Second, up to 40% of Internet traffic belongs to unknown applications [19], many of them ran by botnets. The binary code of these applications may not be available for reverse engineering. Inferring the state machine of unknown protocol from its real world trace can help ISPs have a better understanding to the behaviors of traffic passing through their networks.

We propose *Veritas*, a system that automatically extracts protocol state machine for stateful network protocols from Internet traffic. The input to *Veritas* is the network trace of a specific application. Our output is a *probabilistic* description of the protocol state machine. This probabilistic protocol state machine (P-PSM) is a new and powerful model that we define for capturing and representing any protocols with incomplete knowledge. In order to test and verify *Veritas*, we apply our system to several real-world applications, including a client-server protocol SMTP, two peer-to-peer protocols PPLIVE [21] and XUNLEI [20]. The experiment results show that our system is capable of correctly recognize and classify 86% SMTP flows, 100% PPLIVE and 90% XUNLEI flows. Our tool has the following features: (a) requiring no knowledge of protocol specifications, (b) based on the statistics of protocol formats, and (c) effective for both text and binary protocols. Our contributions are summarized as follows.

– We introduce and formalize a new model – probabilistic protocol state machine (P-PSM) – for describing the protocol state machine in a probabilistic fashion when there is incomplete knowledge about the protocol. P-PSM

model is general and can be used for representing any stateful protocol with uncertain information.
- We design a system called *Veritas*, which can automatically infer the protocol state machine of a specific protocol from its real-world trace with no prior knowledge about the protocol specification. We propose a new technique to extract protocol messages formats that is independent of the type of the target protocol.
- We apply our system to verify real world applications. The applications contain both text-based and binary-based protocols, which are quite complex. Our results demonstrate that *Veritas* is capable of inferring protocol state machine of good quality in practice.

The rest of the paper is organized as follows. Section 2 is dedicated to the related work. In Section 3, we introduce the architecture of *Veritas* and present each portion of the system. In Section 4, we make use of *Veritas* for protocol inference and evaluate the whole system with different protocols. Finally, we conclude our work with future research directions in Section 5.

2 Related work

We divide our discussion of related work into three areas, namely automatic protocol reverse engineering, protocol message format extraction, and inferring protocol state machine.

Automatic protocol reverse engineering. Accurately reversing protocols typically involves manual efforts, such as in the cases of Gaim [23] and [22]. There are several proposals on automating this process. Lim *et al.* [1] proposed a method, which automatically extracted the format from files and application data output. Their works depend on some parameters, such as the output functions, which may not be available. Polyglot [5] proposed a new approach for reverse engineering a protocol by using dynamic analysis of program binaries. In our work, we assume that the program binary is not available; thus our work is orthogonal to the above.

Protocol message format extraction. Much work in the current literature is focused on protocol message format extraction. Kannan *et al.* [8] presented algorithms on extracting the structure of application sessions embedded in the connections. Haffer [7] automated the construction of protocol signatures on traffic that contains the known instance of each protocol. Ma [9] proposed an unexpected means of protocol inference without relying on the port numbers. His method classify the network data belonging to the same protocol automatically. Cui *et al.* [6] introduced a tool, which is for automatically reverse engineering the protocol message format from its network trace. His method divided protocol formats into different tokens by some experiential delimiters. In those studies, inferencing protocol state machine was not investigated.

Inferring protocol state machine. Inferring protocol state machine plays an important role in protocol specifications. The works that are closest to ours include ScriptGen and Prospex. ScriptGen [4] aims to infer protocol state machine

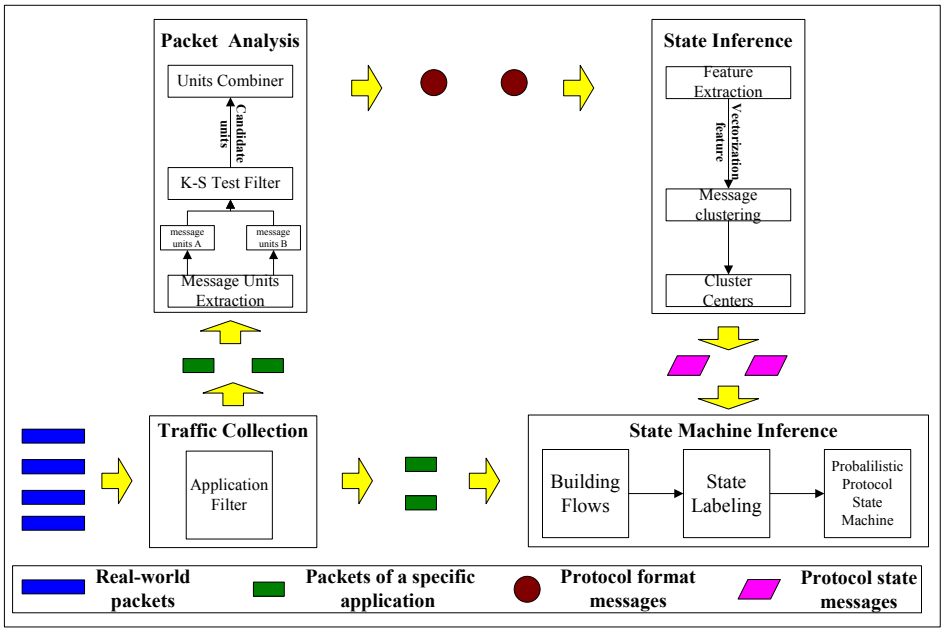

Fig. 1. The Architecture of Veritas

from network traffic. However, ScriptGen has to rebuilds TCP flows based on several assumptions, and it can not handle each TCP session precisely. So those limitations prevent it from emulating all possible protocols. Prospex [3] infers protocol state machine by means of analyzing the execution trace of a program on a stand-alone host. In comparison, our inference is based on observed network traffic that can be performed by ISPs.

3 Architecture of Veritas

The objective of our system is to infer the specifications of a protocol that is used for communication between different hosts. More specifically, given the packet sequences of flows of a specific application, we investigate how protocol state changes from one state to another in the flow. Our approach is to perform machine learning and probabilistic/statistical analysis on the syntax of observed network traces. In this section, we give the definitions used in *Veritas* and an overview of *Veritas* architecture.

We define a protocol as a Markov chain model on sessions of length at most n, which has a discrete state space. The Markov property states that the conditional probability distribution of a system at the next time period depends *only* on the current state of the system, i.e., not depending on the state of the system at previous time periods.

Definition 1. *The message that identifies the state of a protocol is referred to by us as a* ***protocol state message****.*

Protocol state messages are important for understanding the behaviors of the protocol. However, one may not always be able to obtain the state message directly from network traces. Our approach is to infer or estimate protocol state messages by observing and analyzing messages that frequently occur next to them.

Definition 2. *The* ***protocol format message*** *refers to the most frequent string (i.e., the keyword) in the protocol format.*

Protocol format messages are useful for both protocol format inference and obtaining the protocol state message. Protocol format messages include protocol state messages, which we will give more description in the following subsection.

In comparison, research on protocol format extraction typically regards a protocol as a distribution on sessions of length at most n [9], which is static. In other words, the existing solutions work only for extracting the format of a protocol, and cannot be used to describe the protocol states and their transitions.

The input to our system is the network trace of an application. In application-layer packet headers, there may be some protocol state messages. Each of these messages has a message *type*, which indicates the protocol state of the packet. The sequence of packets (belonging to the same flow) is determined by the protocol state machine of a specific application. Meanwhile, the protocol state machine describes how packets of different message types are transmitted in the traces.

Our assumptions In our work, we assume that the network trace is not encrypted. In addition, we assume that the network trace is only composed of flows from the application to be investigated. That is, there is no mixed traffic of multiple protocols.

Veritas has several components as shown in Figure 1, including network data collection, packet analysis, state message inference, and state machine inference, which we describe in more details next.

Network data collection. In this phase, network traffic of a specific application (such as SMTP, DNS, BitTorrent *etc.*) is collected carefully. There are several ways to get packets of a specific protocol, that is the ground truth. For example, the GT [2] method, capturing packets on a specific transport layer port, by means of reverse engineering and so on are all widely used. In this paper, the method of collecting packets on a specific transport layer port is adopted.

Packet analysis. During the phase of packet analysis, we first look for high frequency message units from off-line application-layer packet headers. Then, we employ Kolmogorov - Smirnov (K-S) test [11] to determine the optimal number of message units. Finally, we replay each application-layer packet header and construct protocol message formats with candidate message units.

State message inference. In this phase, we extract the feature from each protocol format message. The feature is used to measure the similarity between

format messages. Then, the partitioning around medoids (PAM) clustering algorithm [12] is applied to group similar messages into a cluster. Finally, the medoid message of a cluster will be a protocol state message.

State machine inference. In order to infer protocol state machine, we should be aware of the protocol state sequence of flows. In order to label protocol state, firstly our system builds flows for a specific protocol. Then, each packet under analysis (if it has a state) will be assigned with a state. Afterwards, by constructing the relationship between different states, a protocol state machine is constructed. Moreover, in each flow the transitions probabilities of diverse states are counted. Finally, together with the protocol state messages, the probabilistic state machine is constituted.

3.1 Packet Analysis

The first stage of *Veritas* is to acquire the formats of protocol messages. In *Veritas*, we extract message formats by applying statistical learning methods on protocol packets. Protocol format messages can be extracted from application-layer packet headers by searching for frequently occurring strings (i.e., keywords). These keywords are typically encapsulated at the beginning of application-layer packets. Taking SMTP (Simple Mail Transfer Protocol) for example, both of the strings "MAIL FROM:" and "RCPT TO:" are its format messages, which usually reside in the SMTP protocol application headers. We assume that the protocol specifications is not available to us. Next, we describe in details how we analyze collected packets in order to infer protocol format messages.

Message Units Extraction Protocol format messages are defined by us as the most frequently occurred strings in the traces of a protocol. From a statistical perspective, if each protocol format can be partitioned into a set of all possible subsequences with fixed lengths, the frequency of these subsequences can be counted precisely.

However, there are two practical problems. The first issue is how to choose the length l of these subsequences, which is critical to the performance of *Veritas*. The second issue is that given a packet how to determine the number n of bytes that are protocol related (i.e., not payload). The latter problem arises, as it is unnecessary to the payload of a packet. Thus, the problem message units extraction turns to determining proper values for l and n.

Definition 3. *The l-byte subsequence originated from the first few bytes of each packet header in network traces is referred to by us as a* ***message unit****.*

We investigate several common application-layer protocols, and find that the minimum field length of those protocols, both text and binary, is at least three bytes. In addition, it is easy to see that the subsequences with length three will be more differentiable than those with length two. The sequence set with three bytes is larger than that with two characters, so high frequency of three-byte sequences is more prominent in special subsequence seeking. On the other hand,

the subsequences with length four or more will weaken its occurrence frequency. Therefore, in *Veritas* we set $l = 3$. Furthermore, if the packet length s is smaller than three bytes, we regard $l = s$. For the other parameter n, we just give a tentative value 12.

It should be noticed that not all of the message units obtained from the method aforementioned are protocol relative. In order to get the high frequency units helpful in characterizing application layer protocol, *Veritas* introduces the two-sample Kolmogorov-Smirnov statistical testing method (abbr. K-S test) [11] to tackle the resulting message units set.

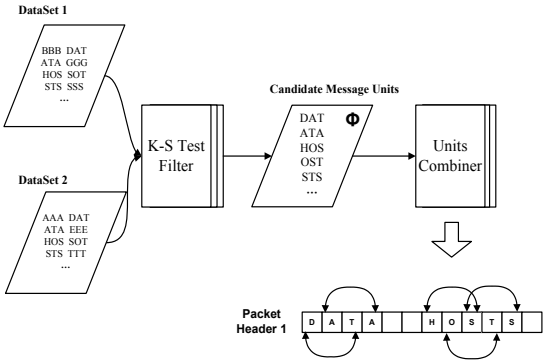

Fig. 2. Packet Analysis of Veritas

K-S Test Filter In this part, we employ a K-S test filter to obtain message units, which are associated with protocol formats. A concrete example is illustrated in Figure 2. The input to K-S test filter is two groups of message units which is obtained by packet analysis. Before conducting K-S test, *Veritas* will turn message units into numeric values. The output to K-S test filter is candidate units which are used for constructing protocol format messages.

The essence of K-S test is to estimate the similarity of two samples according to their empirical probability distributions in a nonparametric way. Given two samples, say S_n and $S_{n'}$, where the subscripts represent the sample sizes, their empirical distribution functions (denote F_n and $F_{n'}$ respectively) can be calculated as $F(X) = \frac{1}{n}\sum_{i=1}^{n} I_{X_i \leq x}$, where $I_{X_i \leq x}$ is the indicator function, equal to 1 if $X_i \leq x$ and equal to 0 otherwise. Then K-S test conducts the similarity measurement by quantifying the distance between F_n and $F_{n'}$ as a statistic $D_{n,n'} = \sup_x |F_n(x) - F_{n'}(x)|$. The null hypothesis is that the samples are drawn from the same distribution, without specifying what that common distribution is. The null hypothesis is rejected (at level α) if $\sqrt{nn'/(n+n')}D_{n,n'} > K_\alpha$, where K_α is the critical value which can be found from $Pr(K \leq K_\alpha) = 1 - \alpha$ under the Kolmogorov distribution.

So, in order to apply K-S test, the packet collection of a specific protocol should be randomly partitioned into two disjoint groups $\mathbb{A}$ and $\mathbb{B}$ with approximately the same size by utilizing the units extraction strategy described in Section 3.1, two groups will yield two message-unit sets, A and B respectively. Then after turning message units into numeric values, the frequency f_x of element x in each set can be counted easily. Then for set A, we partition those elements with frequency higher than or equal to λ into a subset A_λ. Here, λ is a frequency threshold. Doing the same thing on B generates B_λ. Now the application of K-S test on A_λ and B_λ is just to choose a suitable value λ under which the null hypothesis is acceptable. The rejection of K-S test on A_λ and B_λ means the threshold λ is not high enough to cut off useless units.

In the circumstance of *Veritas*, the aim of K-S test is to filter out message units that is not relevant to protocol formats. That is, it requires the result sets A_λ and B_λ responsible for the reflection of protocol formats. Put it in the way of statistical testing, the K-S test on A_λ and B_λ should be accepted at a extremely low level (i.e., $1-\alpha$ should be small enough). In *Veritas*, $1-\alpha$ is valued less than 10^{-8}. Then, for the purpose of accepting the K-S test under the chosen reject level α, *Veritas* manipulates λ in a progressive way: it is initialized as 10^{-5} and gradually increases by 10^{-5} till K-S test accepts.

Once the K-S test finished (i.e., been accepted), the elements in $A_\lambda \bigcap B_\lambda$ will be called **candidate message units**. Then *Veritas* attempts to recover the protocol format messages from these candidate message units.

Protocol Format Message Inference Obtaining protocol format messages is important, as these messages are used for inferring protocol state messages, which is described in Section 3.2. We design a units combiner, which is employed to recover protocol format messages from candidate message units obtained. Here we give a concrete example to explain the process of protocol format messages reconstruction. As shown in Figure 2, the candidate units set ϕ is comprised of five message units (DAT, ATA, HOS, OST and STS). The possible protocol format message can be checked out as follows,

1. Randomly selecting a group of packets from the traffic collection, say Packet Header 1 in Figure 2.
2. With candidate message units, the units combiner tries to rebuild all sequences as long as possible (maybe more than one) for each packet header. So all of these sequences only contain possible three-byte subsequences which are lying in ϕ.
3. All of these obtained sequences are regarded as protocol format messages, such as '$DATA\ \ HOSTS$' in Figure 2.

Furthermore, not all packets contain protocol format message, since some packets only transmit data. Next, we describe our machine-learning methods for inferring protocol state messages.

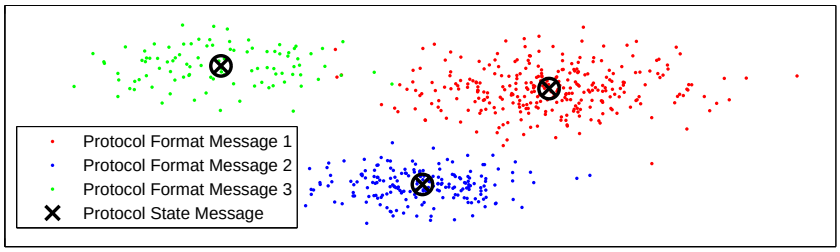

Fig. 3. The Relationship between Protocol Format Messages and Protocol State Messages

3.2 Inferencing Protocol State Messages

As defined earlier, protocol state messages are important and can be used to represent states of a protocol. From our packet analysis, our system obtains a set of protocol format messages from application-layer packet headers, based on which we derive protocol state messages. This derivation is based on a statistical approach, namely using a clustering algorithm.

We need to assign a *type* to each protocol format message, which can be accomplished in two steps. First, we define the features of a protocol format message, as well as a similarity metric between messages. We assume that messages of the same type share a similar message format. Second, our system will group similar format message into a cluster using machine learning methods. Similar messages are likely to be placed in the same cluster, which is labeled by us with the proper type. We define the center of each cluster as a *protocol state message*, which can be used to represent other messages in the cluster. The relationship between protocol state messages and protocol format messages is illustrated in Figure 3. In Figure 3, each dot represents one protocol format message. Furthermore, as shown in Figure 3, protocol state messages are part of protocol format messages, and a protocol state messages is the center message of a cluster.

However, there are two technical challenges in message clustering. First, we have no knowledge of the similarities between messages or their types. Second, we get no prior knowledge about how many state messages are in a certain protocol under analysis. Next, we describe the details of our similarity computation, and how we realize clustering and address the challenges.

Feature Extraction and Similarity Calculation In order to group similar format messages, we need to extract the feature from each format message. In *Veritas*, the feature of a protocol format message is expressed by a vector in $\in \mathbb{R}^{256}$, with the i-th (starting from 0) component counting the number of one-byte character (values i) in that message. Meanwhile, we regard that two format messages of the same type should have a similar character composition. After-

wards, our system carries out similarity calculation between different format messages.

For the purpose of comparing the similarity between two format messages, we make use of the Jaccard index [13], which is defined as follows:

$$J(a,b) = \frac{|a \cap b|}{|a \cup b|}, \tag{1}$$

where, a is the set of elements associated with the feature of the first message, while b is the set that stands for the same feature of the second message. $J(a,b)$ gains its maximum value 1 when all the items in the given set are the same and it will achieve its minimum value 0 when all the items in the given set are distinct.

Message Clustering Based on the feature and the similarity function introduced in the previous subsection, we define the distance between two protocol format messages, which is crucial in clustering. The distance of two protocol format messages a and b is defined as $d(a,b) = 1 - J(a,b)$, where J is the Jaccard index in this paper.

In order to group protocol format messages, we make use of the Partitioning Around Medoids (PAM) algorithm [12]. In contrast to the k-means algorithm, the partitioning around medoids algorithm is more robust to noise and outliers. Therefore, PAM algorithm are suitable for protocol state messages inferring. Just like most other clustering algorithms, the partitioning around medoids algorithm needs an integer value k (the number of clusters) as the input. In order to find out a proper k value, we use a generalization of the Dunn index [18] as a measurement. The Dunn index is a standard intrinsic measurement of clustering quality, defined as follows.

$$D(k) = \frac{\min_{1 \le i \le k}\{\min_{1 \le j \le k}\{\delta(C_i, C_j)\}\}}{\max_{1 \le i \le k}\{\Delta(C_i)\}}, \tag{2}$$

where $C_i, .., C_k$ are the clusters, $\Delta(C_i)$ is the diameter of cluster C_i, and $\delta(C_i, C_j)$ is the distance between two clusters. According to Equation 2, we may see in a clear way that the numerator of Equation 2 is a measure of cluster separation and denominator is a measure of cluster compactness. In our experiment, the k, which maximizes the Dunn index, would be chosen. Finally, the format message of each cluster center is regarded as a protocol state message, and the *type* of the protocol state message is represented by π.

3.3 Probabilistic Protocol State Machine

Because our analysis is based on statistical methods, *Veritas* is able to represent protocol state relations probabilistically. In this section, we introduce a novel

expression of protocol state machine – probabilistic protocol state machine (P-PSM). P-PSM can be used to describe both protocol state transitions and their probabilities. Moreover, the probabilistic protocol state machine is helpful for identifying critical paths of a protocol.

Notation. Let Σ be the set of characters (256 possibilities) and Σ^* be the set of protocol state messages that can be built from Σ. In Σ, symbols can be denoted as (`\00`, `\01`, `\02`, ... , `\ff`) and protocol state messages in Σ^* will be represented by alphabet letters (`a`, `b`, ... , `z`). Therefore, a protocol state transition $\mathcal{T}_{ij}$ can be denoted by (σ_i,σ_2,... ,σ_j) from a starting state i to an accepting state j, where $\forall\sigma \in \Sigma^*$. $\Pr(\mathcal{T}_{ij})$ is a probability $\prod_{k=i,\sigma_k\in\Sigma^*}^{j} \sigma_k$. Moreover, the distribution must satisfy the equation $\sum_{ij\in\Sigma^*} \Pr(\mathcal{T}_{ij}) = 1$. The distribution can be modeled by a probabilistic protocol state machine $\mathcal{A}$ (defined next). The protocol under analysis will be described by $\mathcal{A}$ in a probabilistic manner.

Formal Definition of P-PSM We give the formal definition for probabilistic protocol state machine (P-PSM). P-PSM is a specialization of the general probabilistic finite-state machine [10] in the (network) protocol context.

Definition 4. *A P-PSM is a tuple* $\mathcal{A}$.

$\mathcal{A} = \langle Q_{\mathcal{A}}, \Sigma^*, \delta_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}}, P_{\mathcal{A}}\rangle$, where:

- $Q_{\mathcal{A}}$ is a finite set of states;
- Σ^* is the set of protocol state messages;
- $\delta_{\mathcal{A}} \subseteq Q_{\mathcal{A}} \times \Sigma^* \times Q_{\mathcal{A}}$ is a set of transitions;
- $I_{\mathcal{A}} : Q_{\mathcal{A}} \longrightarrow \mathbb{R}^+$ (initial-state probabilities);
- $F_{\mathcal{A}} : Q_{\mathcal{A}} \longrightarrow \mathbb{R}^+$ (final-state probabilities);
- $P_{\mathcal{A}} : \delta_{\mathcal{A}} \longrightarrow \mathbb{R}^+$ (transition probabilities).

$I_{\mathcal{A}}, F_{\mathcal{A}}, P_{\mathcal{A}}$ are function such that:

$$\sum_{q\in Q_{\mathcal{A}}} I_{\mathcal{A}} = 1, \tag{3}$$

and

$$\forall q \in Q_{\mathcal{A}}, F_{\mathcal{A}}(q) + \sum_{x\in\Sigma^*, q'\in Q_{\mathcal{A}}} P_{\mathcal{A}}(q, x, q') = 1. \tag{4}$$

By convention, P-PSMs are illustrated by directed labeled graphs. In Figure 4, we give a concrete example of P-PSM. In what follows, the subscript $_{\mathcal{A}}$ will be dropped when there is no ambiguity. Typically, a protocol description by means of P-PSM begins with starting states (q_0 in Figure 4) and finishes with accepting states (q_2, q_3 in Figure 4). In Figure 4, there are four states $Q = \{q_0, q_1, q_2, q_3\}$, only one initial-state ($I(q_0) = 1$) and the real numbers in the states are the final-states probabilities. In addition, there are five protocol state messages, $\Sigma^* = \{a, b, c, d, e\}$, and real numbers in the arrows are transition probabilities.

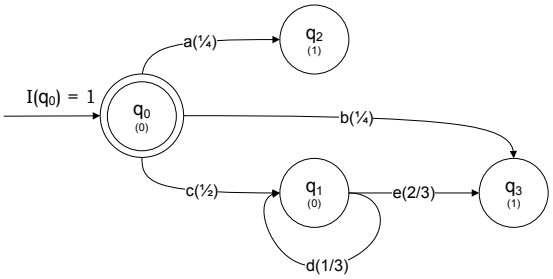

Fig. 4. An Example of P-PSM

3.4 State Machine Inference

Veritas constructs the protocol state machine based on protocol state messages, which are obtained from message clustering. Since each stateful packet has its own *type*, a TCP or UDP flow f_i, can be represented as a sequence $F_i = (\pi_1, .., \pi_h)$, where $\pi_1, .., \pi_h \in M$ and M is the set of protocol state messages. Next, we give details on how to associate a network packet with a state label and to construct the protocol state machine in a probabilistic fashion.

State Labeling We describe how to label each network packet with a state *type*, which is assigned at message clustering. Since our state labeling method is entirely based on a flow model, a 5-tuple of a flow, (source address, destination address, source port, destination port, timeout), is needed as a distinction of different flows. In a 5-tuple, the timeout value indicates the duration of a flow. In our work, several timeout values (16s, 32s, 64s) have been examined in our experiments. From our experiment results, we find that the timeout value is not sensitive in our system, and different timeout values will yield the same experiment results. As a result, in the following experiments, the timeout value will be set to 64s.

As it is defined in previous section, $\pi_i, .., \pi_k$ are cluster center messages (protocol state messages). In this phase, after aligning the two messages to be compared, we denote the feature of the packet header under analysis with ρ, and the feature of the cluster center message π_i with θ_i. For each packet, our system calculates the distance between ρ and θ_i, and labels the packet header ρ with the type of pi_i, which satisfies that $\arg\min d(\rho, \theta_i)$, where $i \in [1, k]$.

However, not each packet header have a state *type*. For example, some data transmission packet do not contain any protocol format message, so it will be not marked with any state *type*. Assuming that $\Delta(C_i)$ is the diameter of cluster C_i, $d_{\max}$ can be defined as follows, $d_{\max} = \max_{1 \le i \le k}\{2\Delta(C_i)\}$. C_h is the cluster that is nearest to the packet header ρ' under analysis. If $d(\theta_h, \rho') > d_{\max}$, the packet header ρ' will be assigned with an unknown state *type*.

After labeling all packets of a specific protocol, *Veritas* constructs a probabilistic protocol state machine, as explained next.

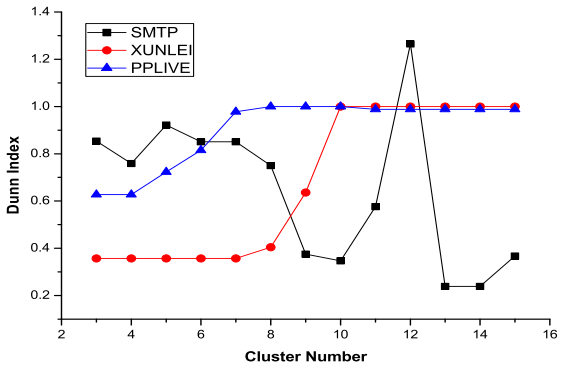

Fig. 5. The changes of Dunn indices with the number of clusters for the three protocols.

Obtaining Probabilistic Protocol State Machine After the phase of state labeling, we are aware of the state *type* π of the packet in each flow. And then in each TCP or UDP flow F_i, $F_i = (\pi_1, .., \pi_h)$, our system calculate the frequency of each state type pair, such as (π_i, π_{i+1}). Therefore, *Veritas* will obtain both the order of different state types and the transition probability from state *type* π_i to π_{i+1}. For the reason that network packets may be out of order in real-world transmission environment, we employ a threshold value as a filter, which can get rid of state *type* pairs that is out of order. The system only keeps the state *type* pairs with a frequency above 0.005.

According to the set of state *type* pairs, our system is able to depict the linkage of each state *type* pair with a directed labeled graph. And all linkages of state *type* pairs are employed to construct a deterministic finite automaton (DFA) of the protocol under analysis, T. Afterwards, we find the minimal DFA that is consistent with T. In the end, probabilistic protocol state machine (P-PSM) will be the combination of minimal DFA and the set of state transition probabilities.

4 Experimental Evaluation

In evaluation section, in order to verify the effectiveness of *Veritas*, we use two kinds of protocols, text and binary. For each protocol under analysis, the input to our system is real-world trace of the protocol, and the output to the system is the protocol state machine described in a probabilistic mode.

4.1 Text Protocol

In this paper, we choose SMTP (Simple Mail Transfer Protocol), which is a stateful and text-based protocol, as a verification of text protocol for our system. In order to infer the P-PSM of SMTP, we capture real-world packets of SMTP

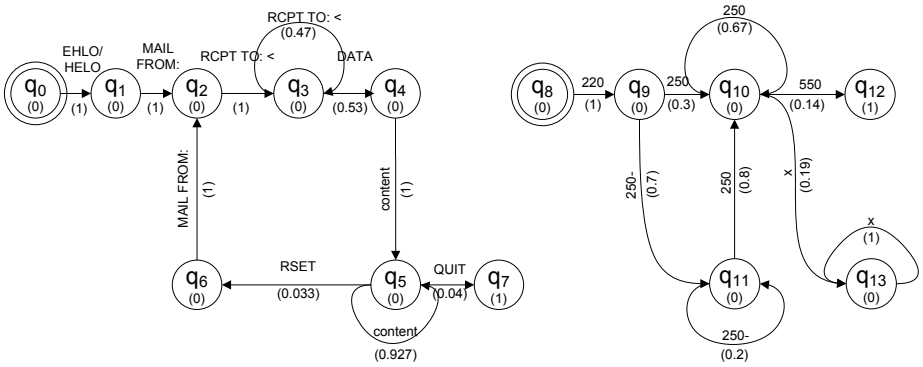

Fig. 6. Probabilistic protocol state machine of SMTP

protocol. In this paper, the data source of SMTP is real-world trace, which is obtained from a backbone router on TCP port 25.

In message clustering phase, as it is shown in Figure 5, the optimal cluster number k for SMTP is 12. Moreover, after several iterative clustering experiments, we find that EHLO and HELO messages are grouped into a cluster. And the probability of EHLO and HELO being the medoid of the cluster is equal. This is due to the fact that mail clients may send an EHLO or a HELO command to initialize a connection.

After state machine minimization, the final P-PSM of SMTP protocol we inferred is shown in Figure 6. As it is shown in Figure 6, the P-PSM of SMTP protocol contains two parts, one may be the state transition of client to server, and the other is the state transition of server to client. In addition, from State q_4 to State q_5, the state machines only carry on SMTP data transmission, which does not contain any state information. Furthermore, from State q_{10} to State q_{13}, unknown protocol state message is represented by x currently.

4.2 Binary Protocols

To test the validation of our system to the binary protocol, in this part we choose PPLIVE and XUNLEI, which are peer-to-peer and binary-based protocols.

Analysis on a P2P Streaming Video Application PPLIVE is a famous peer-to-peer streaming video application in China. The data source of PPLIVE protocol is obtained from our server which only runs an entertainment channel of PPLIVE on UDP port 3987. After state message inference phase, as it is shown in Figure 5, the optimal cluster number k for PPLIVE protocol is 8.

After state machine inference, the ultimate P-PSM of PPLIVE protocol is shown in Figure 7. Moreover, the set of protocol state messages are illustrated in Table 1.

Table 1. PPLIVE Protocol State Messages.

Sign	Protocol State Message
a	0xe9 0x03 0x62 0x01 0x98 0xab 0x01 0x02 0x01
b	0xe9 0x03 0x61 0x01 0x98 0xab 0x01 0x02 0x01
c	0xe9 0x03 0x63 0x01 0x98 0xab 0x01 0x02 0x01
d	0xe9 0x03 0x53 0x00 0x98 0xab 0x01 0x02 0x5b
e	0xe9 0x03 0x49 0x01 0x98 0xab 0x01 0x02 0x98
f	0xe9 0x03 0x51 0x01 0x98 0xab
g	0xe9 0x03 0x50 0x00 0x98 0xab 0x01 0x02 0x9b
h	0xe9 0x03 0x4a 0x01 0x98 0xab 0x01 0x02 0x01

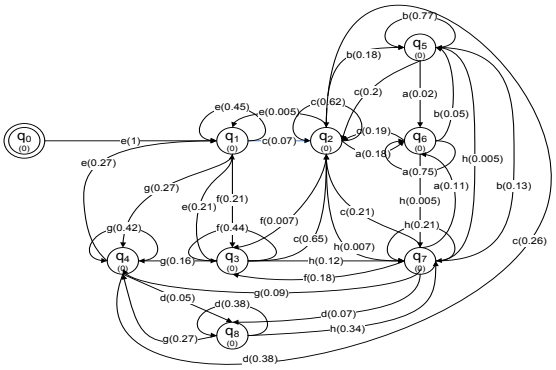

Fig. 7. Probabilistic protocol state machine of PPLIVE protocol.

Analysis on a P2P File-Sharing Application XUNLEI is a popular P2P application in China, and it holds a significant UDP Internet traffic. The data source of XUNLEI protocol is obtained from backbone routers on UDP port 15000. In message clustering phase, as it is shown in Figure 5, the optimal cluster number k for XUNLEI protocol is 10. However, 10 is not the final number of protocol state messages. In the next step, the system will construct DFA and find the minimal DFA that is consistent with it.

After state machine minimization, the ultimate P-PSM of XUNLEI protocol is shown in Figure 8. Moreover, the set of protocol state messages is illustrated in Table 2. Furthermore, sign f is not depicted in Figure 8 for the reason that state *type* pairs correspond with f are of very small probability. As far as we know, f is an old version of XUNLEI protocol state message. If we analyze flows correlated with f respectively, we will get a more comprehensive experiment result, which we do not show here.

4.3 Quality of Protocol Specification

In order to evaluate the quality of protocol specification inferred by *Veritas*, we make use of real-world network traces to test P-PSMs we inferred. In the

Table 2. XUNLEI Protocol State Messages.

Sign	Protocol State Message
a	0x32 0x00 0x00 0x00 0x06 0x00 0x00
b	0x32 0x00 0x00 0x00 0x07
c	0x32 0x00 0x00 0x00 0x08
d	0x32 0x00 0x00 0x00 0x11
e	0x32 0x00 0x00 0x00 0x12
f	0x32 0x00 0x00 0x00 0x09

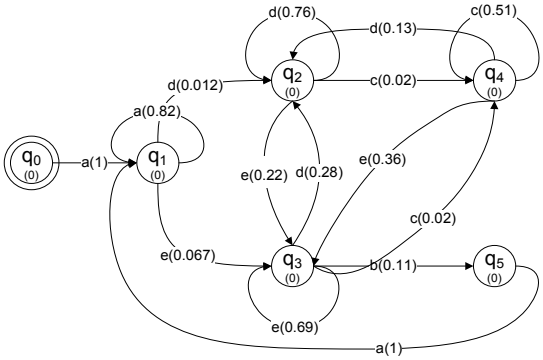

Fig. 8. Probabilistic protocol state machine of XUNLEI protocol.

following experiments, we will demonstrate that the P-PSMs we inferred are complete. Completeness is a measurement of protocol specifications accepting valid protocol sessions.

For SMTP, there are about 100,000 SMTP flows captured from the backbone router. Out of those flows, the SMTP protocol state machine are able to parse the state transitions of about 86% flows successfully. The remaining SMTP flows may use an encryption transmission, which we can not handle properly as one of the limitations of our system is its incompetence to deal with encrypted traffic.

PPLIVE peers always employ UDP packets to communicate and transmit data with each other. For the purpose of testing the quality of the PPLIVE specification, about 200,000 UDP flows of PPLIVE are captured from a server which runs a news channel of PPLIVE on September 9th, 2009. For PPLIVE flows, we are able to parse the state transitions of all flows successfully.

In order to test and verify XUNLEI protocol specification, there are about 500,000 UDP flows of XUNLEI obtained from a backbone router. For XUNLEI flows, we are able to parse the state transitions of about 90% flows successfully. The flows we parsed take up more than 99% XUNLEI protocol packets under analysis. Since our method is based on high probability sets, it will not be sensitive to the event of small probability.

From the above experiment results, we can find that the probabilistic protocol state machines we inferred are of good quality. The whole system can be employed as an auxiliary tool for analyzing unknown behaviors in real-world applications.

4.4 Summary

Our technique for inferring protocol state machine is based on a statistical model, and it is sensitive to states which are statistically significant. Therefore, maybe *Veritas* cannot cover all the paths of a protocol state machine. However, our method is suitable for analyzing critical paths in a protocol, which is very important in intrusion detection. Moreover, our experiment results show that the our inference method has a high degree of accuracy in practice.

5 Conclusions and Future work

Inferring protocol state machine from Internet traffic is a fundamental network security problem, solutions to which have many practical applications. In this paper, we presented a new solution to this problem. We proposed *Veritas*, a system that can automatically extract protocol state machine for stateful network protocols solely from Internet traffic. The input to *Veritas* is the real-world trace of a specific application, and the output is the protocol state machine of that application with a probabilistic description. Our technique proceeds mainly in the following steps. First, the real-world trace of a specific application is extracted from Internet traffic. Then, by analyzing each packet header, we capture the protocol message format from packet headers. Afterwards, by means of clustering algorithms, protocol state messages will be obtained. Based on the clusters, we assign a type to each packet of flows. Finally, we obtain the probabilistic protocol state machine. Our verification experiments show that *Veritas* is general and suitable for both text and binary protocols. The P-PSM inferred by our system reflects the actual applications with high degrees of accuracy.

For future work, we plan to work on semantic inference with *Veritas* for better understanding of protocol specifications. Moreover, *Veritas* can only deal with real-world network trace of a single application. In the future, we would like to make it fit for the multi-protocol environment.

References

1. Lim, Junghee and Reps, Thomas and Liblit, Ben, Extracting Output Formats from Executables, WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (2006)
2. Gringoli, F. and Salgarelli, Luca and Dusi, M. and Cascarano, N. and Risso, F. and claffy, k. c., GT: picking up the truth from the ground for internet traffic, SIGCOMM Comput. Commun. Rev. (2009)

3. Comparetti, Paolo Milani and Wondracek, Gilbert and Kruegel, Christopher and Kirda, Engin, Prospex: Protocol Specification Extraction, SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (2009)
4. C. Leita, K. Mermoud, and M. Dacier, Scriptgen: an automated script generation tool for honeyd, Annual Computer Security Applications Conference (2005)
5. Caballero, Juan and Yin, Heng and Liang, Zhenkai and Song, Dawn, Polyglot: automatic extraction of protocol message format using dynamic binary analysis, CCS '07: Proceedings of the 14th ACM conference on Computer and communications security (2007)
6. Cui, Weidong and Kannan, Jayanthkumar and Wang, Helen J., Discoverer: automatic protocol reverse engineering from network traces, SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (2007)
7. Haffner, Patrick and Sen, Subhabrata and Spatscheck, Oliver and Wang, Dongmei, ACAS: automated construction of application signatures, MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data (2005)
8. Kannan, Jayanthkumar and Jung, Jaeyeon and Paxson, Vern and Koksal, Can Emre, Semi-automated discovery of application session structure, IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (2006)
9. Ma, Justin and Levchenko, Kirill and Kreibich, Christian and Savage, Stefan and Voelker, Geoffrey M., Unexpected means of protocol inference, IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (2006)
10. Vidal, Enrique and Thollard, Franck and de la Higuera, Colin and Casacuberta, Francisco and Carrasco, Rafael C., Probabilistic Finite-State Machines-Part I, IEEE Trans. Pattern Anal. Mach. Intell. (2005)
11. Kendall, M. G. and Stuart, A. and Ord, J. K., Kendall's advanced theory of statistics, Oxford University Press, Inc. (1987)
12. L. Kaufman and P. Rousseeuw, Finding Groups in Data: An Introduction to Cluster Analysis, Wiley. (1990)
13. P. Jaccard, The distribution of the flora in the alpine zone, The New Phytologist. (1912)
14. Brumley, David and Caballero, Juan and Liang, Zhenkai and Newsome, James and Song, Dawn, Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation, SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium. (2007)
15. Juan Caballero and Shobha Venkataraman and Pongsin Poosankam and Min Gyung Kang and Dawn Song and Avrim Blum, FiG: Automatic Fingerprint Generation, Annual Network and Distributed System Security Symposium. (2007)
16. Dreger, Holger and Feldmann, Anja and Mai, Michael and Paxson, Vern and Sommer, Robin, Dynamic application-layer protocol analysis for network intrusion detection, USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium. (2006)
17. Nikita Borisov and David J. Brumley and Helen J. Wang, A Generic Application-Level Protocol Analyzer and its Language, Network and Distributed System Security Symposium (2007)
18. Dunn, J. C., Well separated clusters and optimal fuzzy-partitions, Journal of Cybernetics (1974)
19. Internet2 netflow statistics, http://netflow.internet2.edu
20. XUNLEI, http://www.xunlei.com/
21. PPLIVE, http://www.pptv.com/
22. How Samba Was Written, http://samba.org/ftp/tridge/misc/french_cafe.txt
23. Gaim Instant Messaging Client, http://gaim.sourceforge.net/