Title
Automatic Protocol Inference: Unexpected Means of Identifying Protocols

Authors
Ma, Justin
Levchenko, Kirill
Kreibich, Christian
et al.

Publication Date
2006-02-21

Peer reviewed

Automatic Protocol Inference: Unexpected Means of Identifying Protocols

Justin Ma* Kirill Levchenko* Christian Kreibich†
Stefan Savage* Geoffrey M. Voelker*

*Dept. of Computer Science and Engineering
University of California, San Diego, USA
{jtma,klevchen,savage,voelker}@cs.ucsd.edu

†University of Cambridge
Computer Laboratory, UK
christian.kreibich@cl.cam.ac.uk

Abstract

Network managers are inevitably called upon to associate network traffic with particular applications. Indeed, this operation is critical for a wide range of management functions ranging from debugging and security to analytics and policy support. Traditionally, managers have relied on application adherence to a well established global port mapping: Web traffic on port 80, mail traffic on port 25 and so on. However, a range of factors – including firewall port blocking, tunneling, dynamic port allocation, and a bloom of new distributed applications – has weakened the value of this approach. We analyze three alternative mechanisms using statistical and structural content models for automatically identifying traffic using the same application-layer protocol, relying solely on flow content. In this manner, known applications may be identified regardless of port number, while traffic from one unknown application will be identified as distinct from another. We evaluate each mechanism's classification performance using real-world traffic traces from multiple sites.

1 Introduction

The Internet architecture uses the concept of port numbers to associate services to end hosts. In the past, the Internet has relied on the notion of *well known* ports as the means of identifying the application-layer protocol a server is using [10]. However, in recent years, a number of factors have undermined the accuracy of this association.

In particular, the widespread adoption of firewalling has made some ports far easier to use than others (*i.e.*, the commonly "open" ports such as TCP port 80, used for HTTP traffic, TCP port 25, used for SMTP, and UDP port 53, used for DNS). Thus, to ensure connectivity, there is an increasing incentive to simply use *these* ports for arbitrary applications, either directly or using the native protocol as a tunneling transport layer. Other applications allocate ports dynamically to eliminate the need for application layer demultiplexing. For example, streaming media protocols, such as H.323 and Windows Media, Voice-Over-IP services such as SIP, and multi-player games like Quake routinely rendezvous on ports dynamically selected from a large range. The popular Skype service initializes its listening port randomly at installation, entirely abandoning the notion of well known ports for normal clients [2]. Finally, some applications use non-standard ports explicitly to avoid classification. Peer-to-peer (P2P) applications routinely allow users to change the default port for this purpose and some use combinations of tunneling and dynamic port selection to avoid detection [21]. We can expect this trend of unordered port use to increase further in the future.

Unfortunately, this transformation has created significant problems for network managers. Accurate knowledge of the spectrum of applications found on a network is crucial for accounting and analysis purposes and classifying traffic according to application is also a key building block for validating service differentiation and security policies. However, classification based on well known port numbers remains standard practice. While newer tools are being developed that exploit packet content in their analyses, all of these require ongoing manual involvement – either to create signatures or to label instances of new protocols.

In this paper we tackle the problem of *automatically* classifying network flows according to the application-layer protocols they employ. We do this "in the nude," that is, relying *solely* on flow content. While flow-external features such as packet sizes, header fields, inter-arrival times, or connection contact patterns can be used to aid classification, we argue that only the flow content itself can deliver unambiguous information about the application-layer protocols involved. We make the following contributions:

- We propose a generic architectural and mathematical framework for evaluating the performance of flow classifiers.
- We introduce three classification techniques for capturing statistical and structural aspects of messages exchanged in a protocol: product distributions, Markov models, and common substring graphs.
- We compare the performance of these classifiers using real-world traffic traces and highlight the individual strengths and weaknesses of the three methods.

Starting from the assumption that all traffic to the same destination host and listening port must have a common application-layer protocol[1], we first explore the problem space and position our work in Section 2. We introduce protocol inference in a generic way in Section 3 and show how our three classifiers fit in this problem space in Section 4. We have implemented the classifiers in a single framework, allowing side-by-side evaluation of the classifiers; this is recounted in Section 6. We present our evaluation in Section 7 and discuss the challenges we have observed in undertaking application-layer traffic classification using exclusively flow content, and conclude the paper in Section 9.

2 Background

Traditionally, network-level application analysis has depended heavily on identification via well known ports [3, 19, 6]. As new application patterns, particularly P2P use, undermined this assumption, measurement researchers began to identify the problem and seek workarounds. One class of solutions focuses on deeper structural analyses of communications *patterns*, including the graph structure between IP addresses, protocols and port numbers over time and the distribution of packet sizes and inter-arrival times across connections [12, 13, 24]. These approaches depend on the uniqueness of a particular communication structure within a particular application. While it has been shown to work well for particular application classes (*e.g.*, Mail vs P2P) it is most likely unable to distinguish between application instances (*e.g.*, one P2P system vs another).

Another line of research has focused on payload-based classification. Early efforts focused on using hand-crafted string classifiers to overcome the limitations of port-based classification for various classes of applications [11, 5, 21]. Thus, the Jazz P2P protocol could be recognized by scanning for ``X-Kazaa-*'' in transport-layer flows. Moore and Papagiannaki have shown how to further augment such signatures with causal inference to improve classification [16].

However, the manual nature of this approach presents several drawbacks. First, it presupposes the network manager *knows* what protocols she is looking for. In fact, new application protocols come into existence at an alarming rate and many network managers would like to be alerted that there is "a new popular application on the block" even if they have no prior experience with it. Second, even for well known protocols constructing good signatures is a delicate job – requiring expressions that have a high probability of matching the application and few false matches to instances of other protocols. The latter of these problems has recently been addressed by Haffner *et al.* [8], who automate the construction of protocol signatures by employing a supervised machine learning approach on traffic containing known instances of each protocol. Their results are quite good, frequently approaching the performance of good manual signatures.

Our work builds further upon this approach by removing the requirement that the protocols be known in advance. By simply using raw network data, our unsupervised algorithms classify traffic into distinct protocols based on correlations between their packet content. Thus, using no *a priori* information we are able to create classifiers that can then distinguish between protocols. The basic unit of communication between processes on Internet hosts, be it a large TCP connection or a single UDP packet, is a *session*. A session is a pair of *flows*, each a byte sequence consisting of the application-layer data sent by the *initiator* to the *responder* and the data sent by the responder to the initiator. Each session is identified by the 5-tuple consisting of initiator address, initiator port number, responder address, responder port number, and IP protocol number. Flows are identified by the same 5-tuple and the flow direction, either from the initiator to the responder or from the responder to the initiator. All sessions occur with respect to some *application protocol*, or simply *protocol*, which defines how session data are interpreted by the communicating processes. By observing the network we can identify communication sessions, but we typically cannot directly infer the session protocol. How we might do so is precisely the problem of protocol inference:

Problem: Given a session, identify its protocol.

We emphasize that a session consists only of the data exchanged between two ports on a pair of hosts during the session's lifetime; it does not include packet-level information such as inter-arrival time, frame size, or header fields.

Protocol inference can naturally be divided into two phases. The first is a "learning" or "training" phase, in which a description of the protocols is constructed, and the second is an operational phase, in which unknown sessions are classified. Before describing these two phases, we introduce some preliminary concepts.

2.1 Protocol Models

Any protocol inference algorithm must rely, explicitly or implicitly, on a *protocol model*, which is a set of assumptions about how a protocol manifests itself in a session. In this paper, we consider three protocol models: product distributions (Sec. 4.1), Markov processes (Sec. 4.2), and Common Substring Graphs (Sec. 5), which lead to three different protocol inference algorithms. All three models share the following two assumptions about protocols.

Assumption M1. A protocol is a distribution on sessions of length exactly n.

Assumption M1 tells us that a session protocol may be inferred from n bytes of the session flows; in our experiments, we fix n to be the first 64 bytes. Furthermore, Assump-

[1] Different flows may carry various additional protocols in case of tunneling.

tion M1 posits that there is an unchanging distribution according to which sessions of a protocol are drawn. It is this distribution that defines the protocol.

Assumption M2. Within each session, the data sent by the initiator to the responder is independent from the data sent by the responder to the initiator.

Assumption M2 is a much stronger assumption that, in general, does not hold in the real world. Consider, for example, the DNS protocol. A DNS query contains a 16-bit identification field that is copied into the reply, clearly violating the independence assumption. As a practical matter however, Assumption M2 greatly simplifies our algorithms. In particular, it allows us to treat each flow in the session independently.

Assumptions M1 and M2 do not, by themselves, admit an efficient description of a protocol, which is necessary for an inference algorithm. Each of our protocol models makes additional assumptions that allow the protocol to be efficiently represented within the protocol model.

2.2 *A priori* Information

The problem of protocol inference may be qualified by the type of information about protocols available *a priori*. We recognize three such variants of the problem:

Fully described. In fully described protocol inference, each protocol is given as a (possibly probabilistic) grammar. Identifying the protocol used by a session is a matter of determining which known description best matches the session.

Fully correlated. In fully correlated protocol inference, each protocol is assumed to be defined by some (possibly probabilistic) class of grammars, but the exact grammar is unknown. The grammar of each protocol must be learned from a set of session instances labeled with the protocol.

Partially correlated. In partially correlated protocol inference, each protocol is also assumed to be defined by some (possibly probabilistic) class of grammars, but the exact grammar is unknown. However unlike the fully trained case, only limited information is available about the which sessions have a common protocol.

The focus of this work is on partially correlated protocol inference, meaning that the training data consist of a set of unlabeled sessions with additional information of the form "Session A and Session B are using the same protocol." This auxiliary information is *partial* because not all sessions using the same protocol are identified as such, and only positive equivalences are given. In Section 6 we describe how such training data may be obtained using mild real-world assumptions about protocol *persistence* on host ports. Because all given correlations are positive (*i.e.*, information that two sessions share the same protocol) but partial, it is impossible to infer any negative correlation between cells through the absence of positive correlation (unlike the fully correlated case). This means that technically, identifying all sessions with a single protocol would be consistent with the provided data. For this reason, it becomes necessary for the protocol model to differentiate distinct protocols, a requirement we discuss in Section 3.

3 Protocol Inference

As mentioned previously, protocol inference consists of two phases. In the first phase, a compact description of each protocol is constructed from the training data. The correlation information in the training data allows us to group sessions into protocol equivalence classes consisting of sessions known to use the same protocol. We then construct a tentative protocol description, called a *cell*, in accordance with the protocol model. Because some cells may describe the same protocol, we cluster similar cells and merge them to create a more stable protocol description. The resulting cells define distinct protocols, and are used in the second phase to classify new sessions. To implement the above algorithm, a cell must support the following the following four operations.

Construct. Given a set of sessions of a protocol equivalence class, construct a protocol description in accordance with the protocol model.

Compare. Given two cells, determine their similarity, namely the degree to which we believe them to represent the same protocol.

Merge. Combine two cells believed to represent the same protocol. This operation should be the equivalent of constructing a new cell from the original protocol equivalence classes of the merged cells.

Score. Given a cell and a session, determine the likelihood that the session is using the protocol described by the cell.

3.1 Phase I: Training and Abstraction

Relying on the above operations, we can describe the first phase more rigorously.

1. Combine training data sessions into protocol equivalence classes based on the given correlations.
2. Construct a cell from each equivalence class.
3. Cluster similar cells together based on the result of the Compare operation between pairs of cells.
4. Merge clustered cells into a single cell.

The resulting cells form the protocol descriptions used in the second phase, described in Section 3.2.

Steps 1, 2, and 4 are fairly straightforward in view of the four cell operations described earlier. Step 3, however, requires further elaboration. The objective of step 3 is to correctly combine cells representing the same protocol into one. What reason have we to expect cells representing the same protocol to be similar with respect to the Compare operation and those representing different protocols to be dissimilar? We rely on the following assumption.

Assumption C3. There exists a session number threshold σ and a similarity threshold τ such that all cells constructed from protocol equivalence classes containing at least σ sessions have similarity greater than τ if the underlying protocols are the same and less than τ if the underlying protocols are different.

Assumption C3 tells us that protocol session distributions are fairly dissimilar with respect to the similarity measure defined implicitly by the Compare operation. Moreover, it tells us that the session correlations from which the equivalence classes are derived are sufficiently independent to make this dissimilarity observable. The degree to which Assumption C3 holds true depends both on the similarity measure and the types of correlations between sessions in the training data.

Relying on Assumption C3, step 3 can greedily cluster sufficiently large cells (that is, those constructed from protocol equivalence classes) together without fear that cells representing different protocols will be clustered together.

Assuming each protocol is represented by at least one large-enough protocol equivalence class, the result of Phase I is a set of cells where each protocol is represented by exactly one cell.

3.2 Phase II: Classification

Given a set of cells that we assume represent distinct protocols, classifying a new session is fairly straightforward using the Score operation. To determine the most likely choice of protocol for the session, we simply find the cell with the highest Score with respect to the session.

The result of phase two is a an association between a session and a protocol description (cell), which is not quite the same as saying that the session is using some known protocol, say HTTP. However since protocols are in one-to-one correspondence with cells, we can infer that two sessions are using the same protocol if and only if they are associated with the same cell.

4 Statistical Models

In this section we describe our first two protocol models. Assumptions M1 and M2 tell us that a protocol may be viewed as a pair of distributions on byte strings (flows) of length n. With this in mind, it is natural to view a protocol model entirely in a statistical setting. Before recasting cell operations in statistical terms, we introduce the concept of relative entropy and likelihood with respect to a distribution.

Definition. Let P and Q be two distributions[2] on some finite set U. The *relative entropy* between P and Q is

$$D(P|Q) = \sum_{x \in U} P(x) \log_2 \frac{P(x)}{Q(x)}.$$

Relative entropy is a measure of "closeness" between two distributions. However, it is not a true metric. For more information on relative entropy and some of its interpretations, see for example the text by Cover and Thomas [4]. In this paper, we use *symmetric relative entropy*, defined as

$$\begin{aligned} D(P,Q) &= D(P|Q) + D(Q|P) \\ &= \sum_{x \in U} (P(x) - Q(x)) \log_2 \frac{P(x)}{Q(x)}. \end{aligned}$$

There are other, semantically more natural ways of defining the distance between two distributions. However symmetric relative entropy turns out to be the easiest measure to compute for the special distributions defined by our two statistical protocol models.

We can now define cell operations defined from Section 3 in statistical terms. A cell consists of a pair of distributions $(\vec{P}, \overleftarrow{P})$, the first representing the flow distribution from initiators to responders and the second the flow distribution from responders to initiators within the protocol.

Construct. Given a set of sessions of a protocol equivalence class, create a cell $(\vec{P}, \overleftarrow{P})$ where $\vec{P}$ is the distribution of flows from initiators to responders in the set of sessions and $\overleftarrow{P}$ is the distribution of flows from responders to initiators in the set of sessions.

Compare. Given two cells $(\vec{P}, \overleftarrow{P})$ and $(\vec{Q}, \overleftarrow{Q})$, their distance is $D(\vec{P}, \vec{Q}) + D(\overleftarrow{P}, \overleftarrow{Q})$. Note that this treats each session direction as independent, per Assumption M2. Their similarity is simply the negation of their distance.

Merge. Given two cells as two pairs of distributions, the result of the Merge operation is the weighted sum of the distributions, equivalent to the result of a Construct operation on the protocol equivalence classes from which the original cells were constructed.

Score. Given a cell $(\vec{P}, \overleftarrow{P})$ and a session $(\vec{x}, \overleftarrow{x})$, the score is the probability that both flows of the session are drawn randomly from the pair of distributions defined by the cell. Since the flow distributions in each direction are independent (by Assumption M2), this is just $\vec{P}(\vec{x}) \cdot \overleftarrow{P}(\overleftarrow{x})$.

[2] P being a distribution on a finite set U means that $P(x) \geq 0$ and the sum of $P(x)$ over all x in U is 1.

Unfortunately explicitly representing a pair of flow distributions is not feasible (each distribution consists of 256^n points!), nor is it possible to reasonably learn such distributions approximately with a polynomial number of samples. We assume, therefore, that these flow distributions have a compact representation as a product of n independent byte distributions or as generated by a Markov process, as we describe next.

4.1 Product Distribution Model

The product distribution model treats each n-byte flow distribution as a product of n independent byte distributions. Each byte offset in a flow is represented by its own byte distribution that describes the distribution of bytes at that offset in the flow. Unfortunately, this assumes that each byte of the flow depends only on its offset from the beginning of the flow and not on any of the other bytes. The following example illustrates the significance of this assumption.

Product Distribution Example. For the sake of example, let $n = 4$ and consider the distribution on flows from the initiator to responder for the HTTP protocol. If the byte strings "HEAD" and "POST" have equal probability, then the strings "HOST" and "HEAT" must occur with the same probability; clearly this is an invalid assumption.

Despite relying on such an unrealistic assumption, the product distribution model has proven to be quite effective. It is also remarkably simple.

4.1.1 Construct

Each individual byte distribution is set in accordance with the distribution of bytes at that offset. For example, byte distribution i for the initiator to responder direction would represent the distribution of the i-th byte of the initiator to responder flow across the sessions in the protocol equivalence class.

4.1.2 Compare

The relative entropy of two product distributions $P_1 \times P_2$ and $Q_1 \times Q_2$ is just the sum of the individual relative entropies; that is,

$$D(P_1 \times P_2 \,|\, Q_1 \times Q_2) = D(P_1|Q_2) + D(P_2|Q_2).$$

In fact, this is why the relative entropy of two cells, which consist of two independent distributions on flows per Assumption M2, is just the sum of each direction's relative entropy.

4.1.3 Merge

The merge operation simply returns a convex combination of the underlying distributions. That is, if P_i is the i-th byte distribution in one flow direction of the first cell and Q_i is the i-th byte distribution in the same flow direction of the second cell, then the resulting cell's i-th distribution in that flow direction is $\lambda\, P_i + (1-\lambda)Q_i$, where λ is the number of sessions in the protocol equivalence class from which the first cell was constructed divided by the total number of sessions in the protocol equivalence classes of the first and second cells.

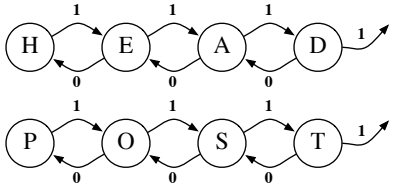

Figure 1: A Markov process for generating the strings "HEAD" and "POST" with each string chosen according to the value of H and P in the initial distribution. Irrelevant nodes have been omitted for clarity.

4.1.4 Score

Let $(\vec{P}_0 \times \cdots \times \vec{P}_{n-1}, \overleftarrow{P}_0 \times \cdots \times \overleftarrow{P}_{n-1})$ be a product distribution cell and $(\vec{x}, \overleftarrow{x})$ be a session. Then the probability of this session under the distribution defined by the cell is

$$\prod_{i=0}^{n-1} \vec{P}_i(\vec{x}_i) \cdot \prod_{i=0}^{n-1} \overleftarrow{P}_i(\overleftarrow{x}_i).$$

4.2 Markov Process Model

Like the Product Distribution Model, the Markov Process Model relies on introducing independence between bytes in order to reduce the size of the distribution. The Markov process we have in mind is best described as a random walk on the following complete weighted directed graph. The nodes of the graph are labeled with unique byte values, 256 in all. Each edge is weighted with a *transition probability* such that for any node, the sum of all its out-edge weights is 1. The random walk starts at a node chosen according an *initial distribution* π. The next node on the walk is chosen according to the weight of the edge from the current node to its neighbors, that is, according to the transition probability. These transition probabilities are given by a transition probability matrix P whose entry P_{uv} is the weight of the edge (u, v). The walk continues until n nodes (counting the starting node) are visited. The flow byte string resulting from the walk consists of the names (*i.e.*, byte values) of the nodes visited, including self-loops.

The probability distribution on length-n flows defined by the above Markov process is defined by the initial distribution π which consists of 256 values, and the transition probability matrix P which consists of 256^2 values. To better understand this distribution, consider the example used for the product distribution model.

Markov Process Example. Again, for the sake of example, let $n = 4$ and consider the distribution on flows from the initiator to responder for the HTTP protocol. Let the

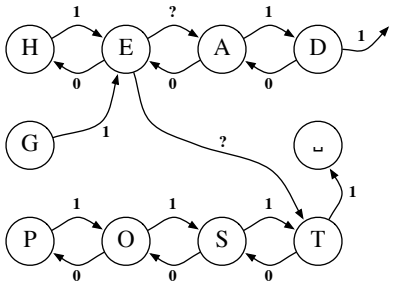

Figure 2: Attempting to add the string "GET" to a Markov process for generating the strings "HEAD" and "POST."

byte string "HEAD" occur with probability p and the byte string "POST" with probability q. The corresponding graph is shown in Figure 1, where the initial distribution is $\pi(\mathrm{H}) = p$, $\pi(\mathrm{P}) = q$, and $\pi(u) = 0$ for $u \neq \mathrm{H}, \mathrm{P}$.

It seems we have avoided the problem we had with the product distribution. However if we try to add the string "GET␣" we quickly run into problems (see Figure 2). Now the byte strings "GEAD" and "HET␣" are also generated by our process!

4.2.1 Construct

The initial distribution π for some flow direction is constructed in the straightforward manner by setting it to be the distribution on the first byte of all the flows (in the appropriate flow direction). The transition probabilities are based on the observed transition frequencies over all adjacent byte pairs in the flows (again, in the appropriate direction). That is, P_{uv} is the number of times byte u is followed by byte v divided by the number of times byte u appears at offsets 0 to $n-2$.

4.2.2 Compare

The relative entropy of two Markov Process Distributions is somewhat involved. For brevity, we omit the proof of the following fact. Let π and ρ be the initial distribution functions of two Markov Processes and let P and Q be corresponding transition probability functions. The relative entropy of length-n byte strings generated according to these processes is

$$\sum_u \pi(u) \log_2 \frac{\pi(u)}{\rho(u)} + \sum_{u,v} \xi(u) \cdot P(u,v) \log_2 \frac{P(u,v)}{Q(u,v)},$$

where

$$\xi(u) = \pi(u) + \sum_{i=1}^{n-2} \sum_{t_1..t_i} \pi(t_1) \cdot \prod_{j=1}^{i} P(t_{j-1}, t_j) \cdot P(t_i, u).$$

4.2.3 Merge

Just as in the case of the Product Distribution Model, the merge operation involves a convex combination of the initial distributions and the transition probability matrix of the two sessions in each of the two directions.

4.2.4 Score

To the probability of a string $x_0, \ldots, x_{n-1}$ according to some Markov process distribution given by initial distribution π and transition probability matrix P, is given by a straightforward simulation of the random walk, taking the product of the probability according to the initial distribution and the edge weights encountered along the walk:

$$\pi(x_0) \cdot \prod_{i=1}^{n-1} P(x_{i-1}, x_i).$$

5 Common Substring Graphs

We now introduce *common substring graphs* (CSGs). These differ from the previous two approaches in that they capture much more structural information about the flows they are built from. In particular, CSGs

- are not based on a fixed token length but rather use longest common subsequences between flows,
- capture *all* of the sequences in which common substrings occur, including their offsets in the flows,
- ignore all byte sequences that share no commonalities with other flows,
- track the *frequency* with which individual substrings, as well as sequences thereof, occur.

A common subsequence is a sequence of common substrings between two strings; a longest common subsequence (LCS) is the common subsequence of maximum cumulative length. We denote the LCS between two strings s_1 and s_2 as $L(s_1, s_2)$ and its cumulative length as $|L(s_1, s_2)|$.

The intuition for CSGs is as follows: if multiple flows carrying the same protocol exhibit common substrings, comparing many such flows will most frequently yield those substrings that are most common in the protocol. By using LCS algorithms, not only can we identify what these commonalities are, but we also expose their sequence and location in the flows. By furthermore comparing many of the resulting LCSs and combining redundant parts in them, frequency patterns in substrings and LCSs will emerge that are suitable for classification.

We will now formalize this intuition. A CSG is a directed graph $G = (N, A, P, n_s, n_e)$ in which the nodes N are labeled and the set of arcs A can contain multiple instances between the same pair of nodes: a CSG is a labeled multidigraph. P is the set of paths in the graph. We define a path $p = (n_1, ..., n_i)$ as the sequence of nodes starting from n_1 and ending in n_i in the graph, connected by arcs. $P(n)$ is the number of paths running through a node n. (If context doesn't make it clear which graph is being referred to,

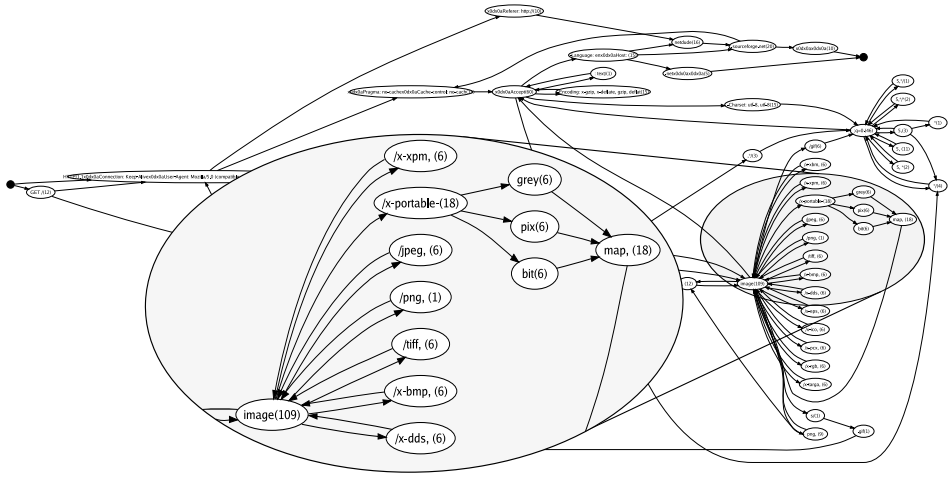

Figure 5: Example of a CSG for the HTTP requests comprising a single website download. The numbers in each node represent the number of paths going through it.

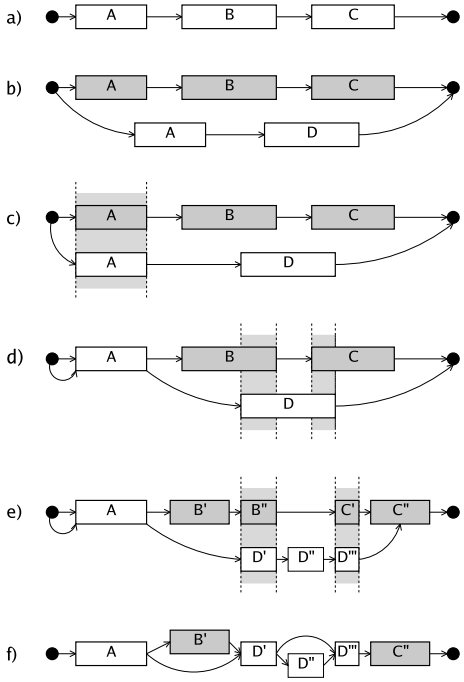

Figure 3: Constructing a CSG: introduction of a new path with subsequent merging of nodes. (a) A CSG with a single, three-node path. (b) An LCS (in white) is inserted as a new path. (c) New node A already exists and is therefore merged with the existing node. (d) New node D overlaps partially with existing nodes B and C. (e) Nodes B, C, and D are split along the overlap boundaries. (f) Identically labeled nodes resulting from the splits are merged. The insertion is complete.

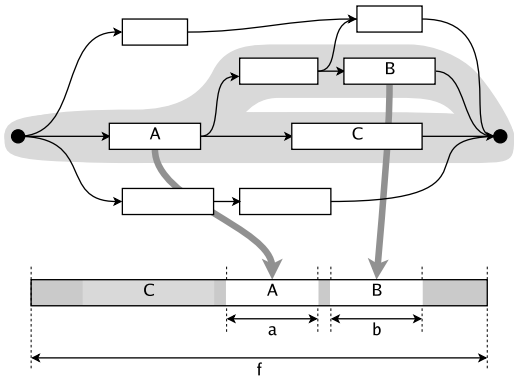

Figure 4: Scoring a flow against a CSG. The labels of nodes A, B, and C occur in the flow at the bottom. The shaded area in the graph indicates all paths considered for the scoring function. While the path containing A-C would constitute the largest overlap with the flow, it is not considered because A and C occur in opposite order in the flow. The best overlap is with the path containing A-B: the final score is $(a + b)/f$.

we will use subscripts to indicate membership, as in N_G, P_G, etc.) A CSG has fixed start and end nodes n_s and n_e. Each path originates from n_s and terminates in n_e, *i.e.*, $P_G(n_s) = P_G(n_e) = |P_G|$. We ignore these nodes for all other purposes; for example, when we speak of a path with a single node on it, we mean a path originating at the start node, visiting the single node, and terminating at the end node. Along the path, a single node can occur multiple times; that is, the path may loop. The node labels correspond to common substrings between different flows, and paths represent the sequences of such common substrings that have been observed between flows. CSGs grow at the granularity of new paths being inserted. For ease of explanation we liken nodes with their labels, thus for example when the say that a node has overlap with another node, we mean that their labels overlap, and $L(n_1, n_2)$ is the LCS of the labels of nodes n_1 and n_2. $|n_i|$ denotes the length of the label of node n_i. La-

bels are unique, *i.e.*, there is only a single node with a given label at any one time.

We make extensive use of a variant of the Smith-Waterman local alignment algorithm for subsequence computation [22]. Given two input strings, the algorithm returns the longest common subsequence of the two strings together with the offsets into the two strings at which the commonalities occur. Our software implementation of Smith-Waterman requires $O(|s_1| \cdot |s_2|)$ space and time given input strings s_1 and s_2. Significant speed-ups are possible by leveraging FPGAs or GPUs [17, 23]. We use linear gap penalty with affine alignment scoring and ignore the possibility of byte substitutions, *i.e.*, we compute only exact common subsequences interleaved with gap regions.

To fulfill the requirements of a cell $(\vec{P}, \overleftarrow{P})$, we put two CSGs into each cell, one per flow direction. We will now describe the realization of the four cell methods in CSGs.

5.0.5 Construct

Insertion of a flow into a CSG works as follows. A flow is inserted as a new, single-node path. If there are no other paths in the CSG, the insertion process is complete. Otherwise, we compute the LCSs between the flow and the labels of the existing nodes. Where nodes are identical to a common substring, they are *merged* into a single node carrying all the merged nodes' paths. Where nodes overlap partially, they are *split* into neighboring nodes and the new, identical nodes are merged. We only split nodes at those offsets that don't cause the creation of labels shorter than a minimum allowable string length.

For purposes of analyzing protocol-specific aspects of the flows that are inserted into a graph, it is beneficial to differentiate between a new flow and the commonalities it has with the existing nodes in a graph. We therefore have implemented a slightly different but functionally equivalent insertion strategy that uses *flow pools*: a new flow is compared against the flows in the pool, and LCSs are extracted in the process. Instead of the flow itself we then insert the LCSs into the CSG as a path in which each node corresponds to a substring in the LCS. The node merge and split processes during insertion of an LCS are shown in Figure 3.

Since many flows will be inserted into a CSG, state management becomes an issue. We limit the number of nodes that a CSG can grow to using a two-stage scheme in combination with monitoring node use frequency through a least recently used list. The list keeps the recently used nodes at the front, while the others percolate to its tail. A *hard limit* imposes an absolute maximum number of nodes in the CSG. If more nodes would exist in the graph than the hard limit allows, least recently used nodes are removed from the graph until the limit is obeyed. To reduce the risk of evicting nodes prematurely, we use an additional, smaller *soft limit*, exceeding of which can also lead to node removal but only if the affected nodes are not important to the graph's structure. In order to quantify the importance of a node n to its graph G we define as the *weight* of a node the ratio of the number of paths that are running through the node to the total number of paths in the graph:

$$W_G(n) = \frac{P_G(n)}{|P_G|}$$

We say a node is *heavy* when this fraction is close to 1. As we will show in Section 7.1, only a small number of nodes in a CSG loaded with network flows is heavy. Soft limits only evict a node if its weight is below a minimum weight threshold. Removal of a node leads to a change of the node sequence of all paths going through the node; redundant paths may now exist. We avoid those at all times by enforcing a uniqueness invariant: no two paths have the same sequence of nodes at any one time. Where duplicate paths would occur, they are suppressed and a per-path redundancy counter is incremented. We do not currently limit the number of different paths in the mesh because it has not become an issue in practice. Should path elimination become necessary, an eviction scheme similar to the one for nodes could be implemented easily.

5.0.6 Compare

In order to compare two CSGs, a graph similarity measure is needed. The measure we have implemented is a variant of feature-based graph distances [20]: the two features we use for the computation are the weights and labels of the graph nodes. Our intuition is that for two CSGs to be highly similar, they must have nodes that exhibit high similarity in their labeling while at the same time having comparable weight. We have decided against the use of path node sequencing as a source of similarity information for performance reasons: the number of nodes in a graph is tightly controlled, while we currently do not enforce a limit on the number of paths.

When comparing two CSGs G and H we first sort N_G and N_H by the length of the node labels, in descending order. Iterating over the nodes in this order, we then do a pairwise comparison $(n_i, n_j) \in N_G \times N_H$, finding for every node $n_i \in N_G$ the node $n_j \in N_H$ that provides the largest label overlap, *i.e.*, for which $|L(n_i, n_j)|$ is maximized. Let the LCS yielding n_i's maximum overlap with the nodes of N_H be denoted as $L_{max}(n_i, N_H)$. The sorting of the nodes allows us to abort the search once we are considering nodes that are shorter than the best match we have previously encountered, so this algorithm is in $O(|N_G| \cdot |N_H|)$. The score contributed by node n_i to the similarity is then the ratio of the best overlap size to the node label's total length, multiplied by $P_G(n_i)$ to factor in n_i's importance. The scores of all nodes are summarized and normalized, resulting in our

similarity measure $S(G, H)$ between two graphs G and H:

$$S(G,H) = \frac{\sum_{n_i \in N_G} P_G(n_i) \frac{|L_{max}(n_i, N_H)|}{|n_i|}}{\sum_{n_i \in N_G} P_G(n_i)}$$

5.0.7 Merge

The way the merge operation proceeds depends on whether the CSG that is being merged into another one needs to remain intact or not. If it does, then merging a CSG G into H is done on a path-by-path basis by duplicating each path $p \in P_G$, inserting it as a new LCS into H, and copying over the redundancy count. If G is no longer required, we can just unhook all paths from the start and end nodes, re-hook them into H, and make a single pass over G's old nodes to merge them into H.

5.0.8 Score

To be able to classify flows given a set of CSGs loaded with traffic, one needs a method to determine the similarity between an arbitrary flow and a CSG as a numerical value in $[0, 1]$. Intuitively we do this by trying to *overlay* the flow into the CSG as well as possible, using existing paths. More precisely, we first scan the flow for occurrences of each CSG node's label in the flow, keeping track of the nodes that matched and the locations of any matches. This is an exact string matching problem and many algorithms are available in the literature to solve it [7]. We are currently using a simple `memcmp()`-iterative approach. The union of paths going through the matched nodes is a candidate set of paths among which we then find the one that has the largest number of matched nodes *in the same order* in which they occurred in the input flow. Note that this gives us the exact sequence, location, and extent of all substrings in the flow that are typical to the traffic the CSG has been loaded with—when using a single protocol's traffic, we can expect to get just the protocol-intrinsic strings "highlighted" in the flow. Finally, to get a numerical outcome we sum up the total length of the matching nodes' labels on that path and divide by the flow length, yielding 1 for perfect overlap and 0 for no similarity. Figure 4 describes the process.

6 Classification Framework

In this section we present a cell-based framework for classifying traffic based on the notion and assumption of constructing protocol models as presented in Sections 2 and 3. Our purpose here is to describe in concrete terms how to implement a classification system based on our models. Moreover, the modularity of this framework allows us to evaluate different protocol models (*e.g.*, Product Distributions, Markov Processes, and Common Substring Graphs) while allowing them to share common components such as surrounding cell construction, clustering, and matching implementations. Figure 6 summarizes the overall operation of Phase I (see Section 3), training cells starting with processing input sessions to merging clusters of cells.

Equivalence Classes: We begin with the first step of grouping sessions into equivalence classes to construct cells, as illustrated in Figure 6a. For our implementation we assume that all communication sessions sharing the same *service key* belongs to the same protocol. Here, we define a service key as the 3-tuple (responder address, responder port, and transport protocol). We believe this key produces a sensible equivalence class because hosts typically communicate with servers at specified address-port combinations. In our experience, the granularity of this equivalence class is coarse enough to admit enough sessions in each class to form statistically significant models. Moreover, it is fine enough so that it does not approach the generality of more coarse (and potentially more inaccurate) equivalences such as treating all sessions destined for the same port as the same protocol—the very assumption that we argue is losing traction with today's protocols.

Augmenting Equivalence Classes with Contact History: We augment service key equivalence classes by making a real-world assumption about the protocol *persistence* between an initiating host and a responding port. In particular, we assume that within a short time period, if an initiating host contacts multiple responders at the same responder port, then the cells corresponding to those service keys must be using the same protocol. Thus, we keep a contact history table that maps initiator-address/responder-port pairs to cells, and merge under the following circumstance: whenever host A contacts the responder at $B : p$, and contacts another responder at $C : p$, then we merge the cells corresponding to service keys $B : p$ and $C : p$. This approach is partly inspired by previous work such as BLINC [14], although our application of external sources of equivalence information is relatively mild and not used during the classification process at all.

Cell Promotion, Comparison, and Merging: After inserting session into their respective cells, we need to *promote* them in preparation for clustering (Figure 6b). However, observing a single session within a cell is insufficient to accurately infer the underlying protocol. Thus, we find it useful to allow the cell to receive enough traffic to construct a reasonable model. For our implementation, we set the promotion threshold to a minimum of 500 flows (not sessions) per cell.

Finally, we perform clustering on the cells with the goal of forming compact descriptions of the observed protocols. We currently perform an agglomerative clustering to construct a hierarchy of cells, iteratively merging the closest pair of cells according to the comparison operation.

Summary: The Cell framework is a realization of the protocol inference approach described earlier, but it provides a modular platform for evaluating various aspects of the

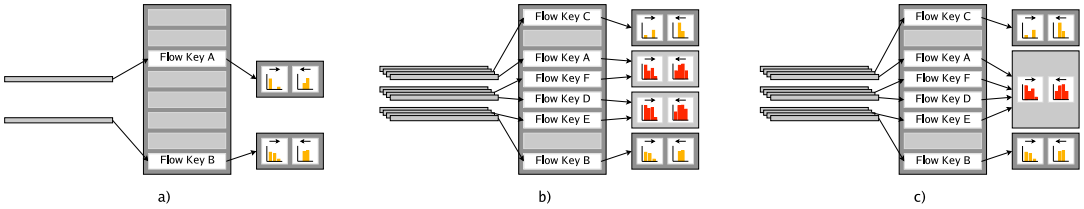

Figure 6: The Cell framework. (a) Flows are mapped to flow keys, stored in a hashtable. Each flow key points to a cell; the cells are only lightly loaded and have not yet been promoted. (b) More flows have been added, multiple flow keys now point to the same cells. The first cells have been promoted for merging. (c) Cells have begun merging.

traffic classification problem. Cell construction could benefit from more elaborate schemes of inferring equivalence classes. Moreover, the framework would provide us the flexibility to experiment with a variety of machine-learning approaches outside of agglomerative clustering to merge cells.

However, the most important aspect for this paper is that the framework allows us to flexibly evaluate the viability of Product Distributions, Markov Processes, and Common Substring Graphs as protocol models independent of the schema for constructing equivalence classes, or the clustering algorithms used after construction. We demonstrate in our evaluations that using Product Distributions, Markov Processes, and Common Substring Graphs with simple approaches such as contact graphs and agglomerative clustering yields promising classification results.

7 Evaluation

We implemented the cluster construction and flow matching components of the Cell framework in C++ using 3800 lines of code. The CSGs were simultaneously developed in the Cell framework and the Bro IDS [18] to allow for more flexible testing of input traffic. We ran all experiments on a dual Opteron 250 with 8 GB RAM running Linux 2.6.

We collected a several traces ranging from 30 minutes to 2.5 hours from a department backbone switch during the period of November 30, 2005 through February 7, 2006. In order to obtain session data out of raw packets, we reassembled TCP flows and concatenated UDP datagrams, using Bro. Session lifetimes are well defined for TCP through its various timeouts; for UDP we used a timeout of 10 seconds. Next we filtered out all flows containing no payload (essentially failed TCP handshakes) because we cannot classify them using a content-based flow classifier. We then used Ethereal 0.10.14 [1] as an oracle to provide a protocol label for each of the flows in the trace. Additionally, we filtered any flows that Ethereal could not identify because we want to compare our classifications to a ground truth provided by an oracle. Specifically, whenever Ethereal labeled a flow as simply 'TCP' or 'UDP,' we filtered it out of the trace. From the combined traces, flows labeled 'TCP' composed 0.6% of all flows. Moreover, although flows labeled 'UDP' comprised 25% of traffic, 88% of the UDP-labeled flows (22% of all traffic) were instances of the Slammer worm, and the remainder composed 3% of all flows.

After preprocessing, we stored the first k bytes of each reassembled flow in a trace ready for consumption by the Cell classifier. For this paper we set $k = 64$, as was done by Haffner *et al.* [9]. Note that at such short flow prefixes, correct UDP packet flow reconstruction is typically not an issue since a flow mostly consists of just one packet.

7.1 CSG Parameterization

CSGs have four parameters: soft/hard maximum node limits, eviction weight threshold, and minimum string length. We used a soft/hard node limit of 200/500 nodes, a minimum weight threshold of 10%, and 4-byte minimum string length. To validate that these are reasonable settings, we selected 4 major TCP protocols (FTP, SMTP, HTTP, HTTPS) and 4 UDP ones (DNS, NTP, NetBIOS Nameservice, and SrvLoc) and for each of them picked a destination service hosting at least 1000 sessions. We manually inspected the services' traffic to ensure we did indeed deal with the intended protocol. In three separate runs with minimum string lengths of 2-4 bytes, 8 CSGs were loaded with each session's first message while we recorded node growth and usage. Figure 7 shows the number of nodes in each graph during the construction. The protocols exhibit fairly different growth behaviors, but all of them tolerate the 200-node soft limit. HTTP repeatedly pushes beyond the limit but never loses nodes beyond the eviction weight threshold. Figure 8 shows the frequency distribution of each CSG's nodes after 1000 insertions. In all CSGs except for the FTP one, at least 75% of the 200 nodes carry only a single path. The FTP CSG only grew to 11 nodes in the 2-byte run, explaining the cruder distribution. Minimum string length seems to matter little. Thus, our CSG settings seem tolerant enough not to hinder natural graph evolution.

7.2 Classification Experiment

In our classification experiment, we examine how effective our three algorithms (Product, Markov process, and CSGs) are at classifying flows in a trace. We proceed in two phases. The first *clustering* phase accepts a training trace for training and produces a definitive set of clusters for describing protocols in the trace. The second *classification* phase then labels the flows in a testing trace by associating them with one of

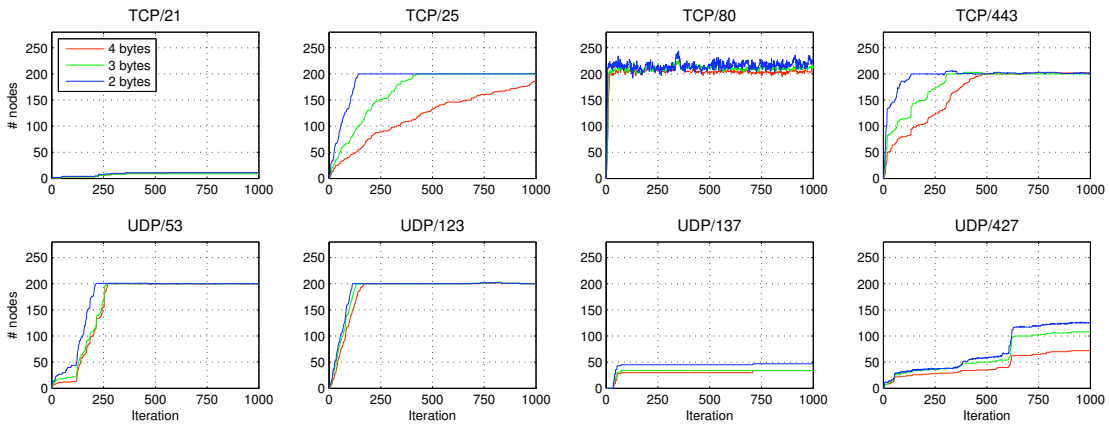

Figure 7: CSG node growth during insertion of 1000 sessions, for various minimum string lengths.

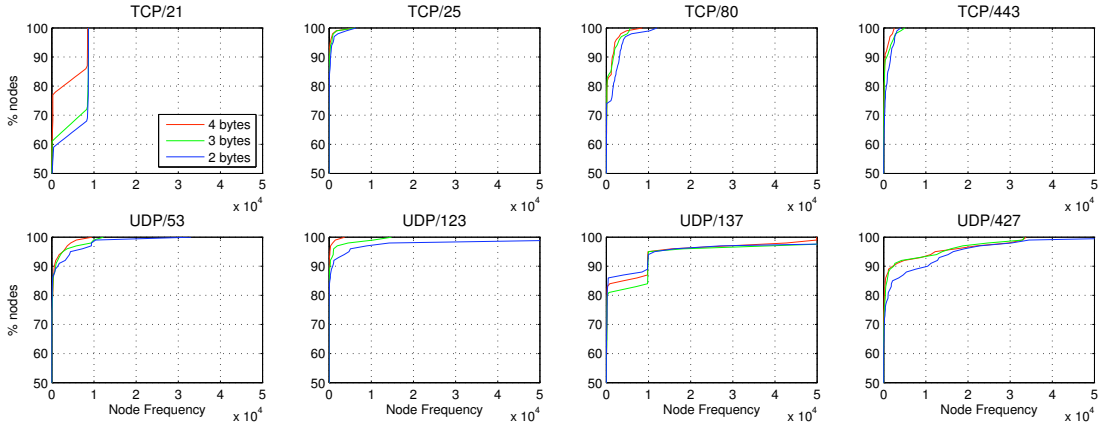

Figure 8: CSG node frequencies after 1000 insertions, for various minimum string lengths.

the definitive clusters produced in the first phase. The purpose of this experiment is to simulate the process by which a network administrator may use our system—by first building a set of cells to describe network traffic, and then classifying subsequent traffic using those cells. We describe each phase in more detail as follows.

7.2.1 Clustering Phase

Clustering is the process of producing a set of clusters (merged cells) that succinctly describes the traffic in a trace according to a clustering metric. In the current implementation, this involves *inserting* the input trace into cells and merging cells according to host contact patterns as described in Section 6. Then, the cell framework *promotes* cells that meet the promotion threshold, and prunes the rest from the cell table. In these experiments we promote cells that contain at least 500 flows. Afterward, we create a hierarchy of cell merges using a simple agglomerative (bottom-up) clustering. The distance metric is relative entropy for Product and Markov (Section 4), and approximate graph similarity for CSGs (Section 5.0.6). Each iteration of clustering merges the pair of cells with the lowest distance to produce a new cell, and recomputes the distance between the new cell and the others.

After producing a clustering hierarchy, we *replay* the merge operations to find the clustering that yields the lowest error according to a quality-of-clustering metric. There are various measures in the literature for evaluating the quality of a clustering [15]. However, for our purposes we choose a simpler metric that mimics the decision process of a constrained network administrator: *majority-first*. Computing the majority-first metric emulates the network administrator's task of labeling each cell with a protocol under the following constraints: (1) the administrator may only assign protocol X to a cell if the majority of flows belong to protocol X, and (2) the administrator can assign a protocol to at most one cell.

We calculate the majority-first misclassification rate by summing the number of correct classifications for each protocol X_i, subtracting that sum from the total number of flows, and then normalizing the difference by the number

of flows. More formally, let m_{ij} be the number of protocol X_i flows belonging to cell j if protocol X_i constitutes the majority of flows for that cell, but let $m_ij = 0$ if protocol X_i is not in the majority for cell j. Then the majority-first error rate e follows:

$$e = 1 - (\sum_i (\max_j m_{ij})/(total\ flows))$$

The resulting majority-first misclassification rate is the assignment of protocols to cells that maximizes the number of valid matches.

Under these constraints it is still possible to have protocols that we cannot assign to any cell, possibly overestimating the misclassification error compared to other quality-of-clustering metrics. For example, we may overestimate the classification error of clustering where cells c_1 = { 60 flows X, 40 flows Y } and c_2 = { 51 flows X, 49 flows Y}. According to majority-first, X is a valid protocol label for either cell, but we label c_1 with X because it contains more X flows than c_2. Because protocol Y is not in the majority for either cell, we cannot assign it. The resulting error would be $1 - 60/200 = 70\%$. A different metric such as the *clustering error* by Meilaă [15] would assign label Y to cell c_2, and thereby result in a misclassification error of $1 - 109/200 = 45.5\%$. Nevertheless, the important property of the majority-first metric is that it penalizes clusterings where a particular protocol is in the majority for multiple cells.

We use the merge threshold that yields the lowest majority-first error to produce the definitive set of clusters for the classification phase. Although we simulate a supervised learning process with this experiment in principle, the entire experiment runs from start to finish (including merge-threshold selection) requiring no manual intervention. Using other quality-of-clustering metrics to produce a best clustering is an area of future research.

7.2.2 *Classification Phase*

The goal of this phase is to associate each incoming flow with the cell that corresponds to the flow's protocol. To perform this classification, we load the definitive clusters back into the framework and label each test flow with the cluster that yields the best matching score.

After we label each flow with the protocol of the closest matching cell, we compute the misclassification rate. We note that the misclassification rate of this *classification* here should not be confused with the misclassification rate of the *clustering* described in Section 7.2.1.

7.3 Classification Results

Table 1 summarizes the misclassification rates for our framework under the three protocol models. Here we trained a 2.5 hour trace (7.2 million flows) while testing on a 1 hour trace (2.5 million flows). "Total" error encompasses all misclassified flows, including flows belonging to protocols that were absent from the training trace. "Learned" error represents the percent of all flows that were misclassified and belonged to a protocol present in the training trace. Finally, "unlearned" error is the percent of all flows that belonged to protocols absent from the training trace (not surprisingly, this last number stays consistent across all protocols). Table 2 shows the traffic composition for select test trace protocols that we focus on in our discussion.

	total	learned	unlearned
Product	5.66%	5.26%	0.39%
Markov	8.5 %	8.18%	0.39%
CSG	14.00%	13.61%	0.39%

Table 1: Misclassification for three protocol models training on a 2.5 hour trace and testing on a 1 hour trace.

Proto	Flows	%
DNS	1,285,718	50.8
NTP	306,031	12.1
HTTP	208,707	8.2
NBNS	185,569	7.3
SMTP	122,059	4.8
SNMP	67,078	2.7
YPSERV	40,525	1.6
Total	2,530,558	

Table 2: Select protocols from the 1-hour testing trace.

Tests with product distributions yielded the lowest classification error at 5.66% total error, while results for Markov processes were somewhat worse at 8.5% total error. Classification error for Content String Graphs was the largest at 14% total error (partly because we limited the maximum flows per cell to improve runtime performance). For all models except CSGs, classification results improve slightly with a longer training trace. Table 3 presents misclassification, precision, and recall rates for select protocols within the 1 hour test trace.

Product Distribution performed well over all protocols, particularly popular ones such as HTTP, NTP, NBNS. This model benefited strongly from the presence of invariant protocol bytes at fixed offsets within the flow. However, the largest number of misclassified flows resulted from false positive identifications for DNS. The precision and recall numbers are high for DNS because it composed the majority of flows in the 1 hour trace (1.3 million). Nevertheless, much of the misclassification error came from binary protocols being misidentified as DNS because of the uniformity of its byte-offset distributions (due in part to its high prevalence). Another interesting source of false positives was the YPSERV flow, whose content consisted primarily of NULL-bytes. Other protocols that heavily used NULL-bytes (*e.g.*, TLS and NFS) were frequently misidentified as YPSERV.

Protocol	Product			Markov			CSG		
	Misclass%	Precision%	Recall%	Misclass%	Precision%	Recall%	Misclass%	Precision%	Recall%
DNS	3.03	94.63	99.90	2.88	94.97	99.7	4.15	96.42	95.41
HTTP	0.17	98.54	99.94	0.41	97.71	97.6	0.35	97.54	99.89
NBNS	0.01	99.84	99.90	0.65	91.95	99.9	2.24	79.98	93.55
NTP	0.04	99.88	99.93	1.94	99.20	84.6	5.86	89.60	58.61
SMTP	0.09	98.62	99.57	0.02	99.78	99.9	0.02	99.85	99.79
SNMP	0.12	99.40	95.99	<0.01	99.99	100.0	0.02	99.40	100.00
YPSERV	1.66	51.81	100.00	1.90	0.00	0.00	2.16	44.01	99.30

Table 3: Misclassification, precision and recall rates of select learned protocols (those present in the training trace).

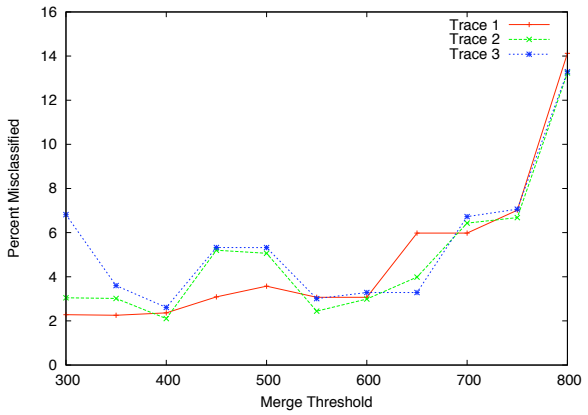

Figure 9: Misclassification rate for Product Distribution over merging threshold for training on three different hour-long traces, and testing on the 30-minute trace. The rate includes protocols that were not present in the training trace.

The misclassification rate for Markov is slightly better for DNS than Product Distribution. Nevertheless, the greatest weakness of Markov is misclassification of protocols containing many NULL-bytes such as YPSERV, more so than Product. Unlike Product, Markov cannot take advantage of byte offset information—hence protocols that contain long runs of NULL bytes, regardless of their offset, are undesirably grouped together.

Interestingly, CSGs performed well classifying protocols that Product Distributions misclassified, such as TLS (not shown). By contrast, CSGs incurred high misclassification rates for binary protocols that Product Distribution successfully classified, such as NBNS, NTP, and YPSERV. Nevertheless, the classification accuracy is respectable considering that we limited the number of flows in each CSG cell to 500.

7.4 Parameter Space

Finally, we answer the question of whether our approach is sensitive to the merge threshold—*i.e.*, is it possible to come up with a single threshold to apply to clustering traffic for different time periods. If classification errors remain consistent across different trace, this would suggest that pure online-clustering with a fixed merging threshold is viable with our system (as opposed to the previous experiment's approach of searching for the best threshold). Although our experience is limited, the results are promising.

Here we focus on Product Distributions and train on each of three hour-long traces. For a specified merge threshold, we merge cells until the minimum distance between any merged cell is greater than that threshold. This becomes the definitive set for the test phase. Then, we test the clusters against the 30-minute trace to measure misclassification rates.

Figure 9 shows that the total error rate (including misclassification on protocols absent from the training trace) remains relatively consistent across the test traces. The largest difference is roughly 4.5 percentage points between Trace 1 and Trace 3. The error rates afterward remain within 2 percent of each other.

8 Discussion

As we have shown in this paper, protocol inference using only flow content is a multi-faceted challenge. While product distributions have achieved the best over result, a look at individual protocol classification results suggests that each of the three methods has different strengths and weaknesses. Given that we only looked at the first 64 bytes of a flow, the results are quite promising, in particular since such short length means that the flow reassembly requirements are minimal even though we do content-based analysis. Another lesson learned is that binary protocols are not generally easy to classify because of sequences of null-bytes that are common multiple protocols, at similar locations. We note that the purpose of this work was not an investigation of the runtime performance of the three models, but the challenges they face in the classification task.

9 Conclusion

Protocol inference is as much a game of carefully chosen assumptions, like stepping stones across a stream, as it is one of clever algorithms and measurements. Our first contribution is to split the problem into three nearly-parts: session protocol correlation, protocol modeling, and cell clustering. Within this framework, we have presented a fairly simple mechanism for deriving protocol correlations from raw traffic data that is remarkably effective; we described and imple-

mented three protocol models, each revealed to have its own strengths and weaknesses, and a straightforward clustering algorithm for unifying protocol descriptions. The resulting system demonstrates that it is possible to drive a protocol inference system using only the partial information available in network traces without relying on additional data.

Acknowledgments

We would like to thank Vern Paxson for his helpful input on substring-based traffic analysis methods, Jim Madden and David Visick for their help understanding the UCSD network, Sameer Agarwal for insightful discussions on clustering, and both Michael Vrable and Michelle Panik for feedback on earlier versions of this paper. This work was supported by NSF grant CNS-0433668, Intel Research Cambridge, and the UCSD Center for Networked Systems.

References

[1] Ethereal: A network protocol analyzer. `http://www.ethereal.com`.

[2] S. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. Technical report, Columbia University, New York, NY, 2004.

[3] K. Claffy, G. Miller, and K. Thompson. The nature of the best: Recent measurements from an Internet backbone. In *Proc. of INET '98*, jul, 1998.

[4] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991.

[5] C. Dewes, A. Wichmann, and A. Feldmann. An Analysis of Internet Chat Systems. In *Proc. of the Second Internet Measurement Workshop (IMW)*, Nov 2002.

[6] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-level Traffic Measurements from the Sprint IP Backbone. *IEEE Network*, 17(6):6–16, 2003.

[7] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[8] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated Construction of Application Signatures. In *Proc. of the ACM SIGCOMM Workshop on Mining Network Data*, August 2005.

[9] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated construction of application signatures. In *Proceedings of the 2005 Workshop on Mining Network Data*, pages 197–202, 2005.

[10] IANA. TCP and UDP port numbers. `http://www.iana.org/assignments/port-numbers`.

[11] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos. Is P2P dying or just hiding? In *IEEE Globecom 2004 - Global Internet and Next Generation Networks, Dallas/Texas, USA*, Nov, 2004. IEEE.

[12] T. Karagiannis, A. Broido, M. Faloutsos, and K. Claffy. Transport Layer Identification of P2P Traffic. In *Proc. of the Second Internet Measurement Workshop (IMW)*, Nov 2002.

[13] T. Karagiannis, D. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. In *Proceedings of ACM SIGCOMM*, oct 2005.

[14] T. Karagiannis, D. Papagiannaki, and M. Faloutsos. BLINC: Multilevel traffic classification in the dark. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 229–240, 2005.

[15] M. Meilă. Comparing clusterings — an axiomatic view. In *Proceedings of the 22nd International Conference on Machine Learning*, 2005.

[16] A. Moore and D. Papagiannaki. Toward the Accurate Identification of Network Applications. In *Proc. of the Passive and Active Measurement Workshop*, mar 2005.

[17] T. Oliver, B. Schmidt, and D. Maskell. Hyper customized processors for bio-sequence database scanning on fpgas. In *FPGA '05: Proc. of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 229–237, New York, NY, USA, 2005. ACM Press.

[18] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23-24):2435–2463, 1998.

[19] D. Plonka. FlowScan: A Network Traffic Flow Reporting and Visualization Tool. In *Proc. of USENIX LISA*, jul, 2000.

[20] A. Sanfeliu and K. Fu. A Distance Measure Between Attributed Relational Graphs for Pattern Recognition. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13(3):353–362, 1981.

[21] S. Sen, O. Spatscheck, and D. Want. Accurate, Scalable In-network Identification of P2P Traffic Using Application Signatures. In *Proc. of the 13th International World Wide Web Conference*, may 2004.

[22] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147, 1981. `http://gel.ym.edu.tw/~chc/AB_papers/03.pdf`.

[23] G. Voss, A. Schröder, W. Müller-Wittig, and B. Schmidt. Using Graphics Hardware to Accelerate Biological Sequence Analysis. In *Proc. of IEEE Tencon*, Melbourne, Australia, 2005.

[24] S. Zander, T. Nguyen, and G. Armitage. Self-learning IP Traffic Classification based on Statistical Flow Characteristics. In *Proc. of the 6th Passive and Active Network Measurement Workshop*, March 2005.