Unexpected Means of Protocol Inference

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 that uses 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.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Algorithms, Measurement, Experimentation

Keywords

Application Signatures, Traffic Classification, Protocol Analysis, Sequence Analysis, Network Data Mining, Relative Entropy, Statistical Content Modeling

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 which application-layer protocol a server is using [9]. In recent years, however, 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 [20]. 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 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 *unsupervised* protocol inference.
- We introduce three classification techniques for capturing statistical and structural aspects of messages exchanged in a protocol: product distributions, Markov processes, and common substring graphs.
- We compare the performance of these classifiers using real-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IMC'06, October 25–27, 2006, Rio de Janeiro, Brazil.
Copyright 2006 ACM 1-59593-561-4/06/0010 ...$5.00.

world traffic traces in two use settings: semi-supervised post-hoc classification and new protocol discovery, highlighting the individual strengths and weaknesses of the three techniques.

We believe that the most significant impact of our work will be relieving network analysts from the need to classify unknown protocols or new protocol variants. We show that it is possible to automatically group protocols without *a priori* knowledge. Thus, labeling a single protocol instance is sufficient to classify all such traffic. In effect, we have substituted the painful process of manual flow analysis and classifier construction with the far easier task of recognizing a protocol instance.

The remainder of this paper is structured as follows. We first explore the problem space and position our work in Section 2. We introduce protocol inference in Section 3 and show how our three classifiers fit in this problem space in Sections 4 and 5. We have implemented the classifiers in a single framework, and describe this framework in Section 6. We present our evaluation in Section 7. We discuss our approach and results in Section 8 and conclude in Section 9.

2. RELATION TO EXISTING WORK

Traditionally, network-level application analysis has depended heavily on identification via well known ports [4, 7, 18]. New application patterns, particularly P2P use, undermined this assumption, leading measurement researchers to seek workarounds. One class of solutions focuses on deeper structural analyses of communication *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 [3, 11, 12, 15, 23]. These approaches depend on the uniqueness of specific communication structures within a particular application. While this approach has been shown to work well for separate 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 [6, 10, 20]. 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 [14].

However, the manual nature of this approach presents several drawbacks. First, it presupposes that network managers *know* what protocols they are 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. In this sense (i.e., of being unsupervised), our approach is similar in spirit to that of Bernaille et al. [3], who suggest using the sizes of the first six packets in a session as the protocol signature.

3. PROTOCOL INFERENCE

Below we define the problem background and basic terminology and then describe the foundation of our approach to protocol inference, namely statistical protocol modeling.

3.1 Background

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. 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.

All sessions occur with respect to some *application protocol*, or simply *protocol*, which defines how communicating processes interpret the session data. By observing the network we can identify communication sessions, but we cannot directly learn the session protocol. For this to be possible, sessions of different protocols must "look different" from each other. To formalize what this property means, we need the concept of a *protocol model*.

3.2 Protocol Models

Protocol inference relies, explicitly or implicitly, on a *protocol model*: a set of premises about how a protocol manifests itself in a session (i.e., a pair of flows, one from the initiator to the responder and one from the responder to the initiator). From the network view, a protocol is simply a distribution on sessions; that is, a protocol is described by the likelihood of each pair of flows.

To make the problem tractable, we restrict ourselves to finite distributions by bounding the length of a session. In other words:

Premise 1. A protocol is a distribution on sessions of length at most n.

Another way to think of Premise 1 is that we are assuming that the protocol to which a session belongs can be inferred from the first n bytes of a session; in our experiments, we fix n to be 64 bytes as in [8].

Unfortunately it is infeasible to work with n-byte session distributions as these consist of $256^{2 \cdot n}$ possible pairs of flows. To be useful, a protocol model must be simultaneously (1) sufficiently expressive to capture real-world protocols, and (2) compact so that it can be learned from a small number of samples and described efficiently.

Toward this end, we treat distributions on sessions as a pair of distributions on flows, rather than a distribution on pairs of flows:

Premise 2. A protocol is a pair of distributions on flows (one from the initiator to the responder and one from the responder to the initiator).

What we gain from Premise 2 is a drastic reduction on complexity, from $256^{2 \cdot n}$ to $2 \cdot 256^n$ values, to exactly describe a protocol. Unfortunately this is still not enough to satisfy requirement (2), and therefore it is necessary to further restrict the class of distributions used by our classification models. For the statistical models (Section 4), this class is given explicitly; for Common Substring Graphs (Section 5), this class is implicit in the data structure.

3.3 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, a protocol is also assumed to be defined by some (possibly probabilistic) class of grammars, but the exact grammar is unknown. Unlike the fully trained case, however, only limited information is available about 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. Since 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 sessions through the absence of positive correlation (unlike the fully correlated case). In the absence of negative correlation, there may be several hypotheses that are consistent with the training data, ranging from all absent correlations being negative (maximum number of distinct protocols) to all absent correlations being positive (a single protocol for all sessions). We describe how to distinguish between these two cases, as well as the cases in between, next.

3.4 Protocol Construction

Constructing a protocol description requires some session instances of each protocol extracted from the training data. The correlation information in the training data allows us to group sessions into protocol equivalence groups 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. As multiple 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 three operations.

Construct. Given a set of sessions of a protocol equivalence group, 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 groups of the merged cells.

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

1. Combine training data sessions into equivalence groups based on the given correlations. Each group consists of sessions using the same protocol.
2. Construct a cell from each equivalence group.
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.

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. This objective requires that cells of the same protocol be "similar" and cells of different protocols "dissimilar." This premise is central to our work, so we state it formally.

Premise 3. For some size threshold σ and similarity threshold τ, cells constructed from protocol equivalence groups containing at least σ sessions must have similarity greater than τ if the underlying protocols are the same, and less than τ if the underlying protocols are different.

We base our protocol inference algorithms on this premise with parameters σ and τ determined empirically. Note that we do not claim that Premise 3 is always true in the real world, only that it is a useful assumption for designing protocol inference algorithms. Each of our three protocol models (Sections 4 and 5) defines its own similarity measure.

3.5 Cells as Classifiers

Because in our model protocols are described by distributions, we can classify unknown sessions by matching them with the maximum-likelihood distribution (cell). This is captured by the following cell operation:

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

We use this as the basis for our classification experiment presented in Section 7.

4. STATISTICAL MODELS

In this section we describe our first two protocol models. Premises 1 and 2 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[1] 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 "dissimilarity" between two distributions. However, it is not a metric in the strict mathematical sense. For more information on relative entropy and some of its interpretations, see for example the text by Cover and Thomas [5]. 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 is the easiest measure to compute for the special distributions defined by our two statistical protocol models, and has provided excellent results in practice.

We can now describe cell operations defined in Section 3 in statistical terms. A cell consists of a pair of distributions $(\overrightarrow{P}, \overleftarrow{P})$, 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 group, create a cell $(\overrightarrow{P}, \overleftarrow{P})$ where $\overrightarrow{P}$ 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 $(\overrightarrow{P}, \overleftarrow{P})$ and $(\overrightarrow{Q}, \overleftarrow{Q})$, their distance is $D(\overrightarrow{P}, \overrightarrow{Q}) + D(\overleftarrow{P}, \overleftarrow{Q})$. 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 groups from which the original cells were constructed.

Score. Given a cell $(\overrightarrow{P}, \overleftarrow{P})$ and a session $(\overrightarrow{x}, \overleftarrow{x})$, the score is the probability that both flows of the session are drawn randomly from the pair of distributions defined by the cell. This is $\overrightarrow{P}(\overrightarrow{x}) \cdot \overleftarrow{P}(\overleftarrow{x})$.

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. Instead, we compactly represent flow distributions by introducing independence assumptions into our models for an exponential reduction in space. Next we describe two approaches, one that represents flow distributions as a product of n independent byte distributions and a second that represents them as being generated by a Markov process.

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. For this reason, we expect this model to be successful at capturing binary protocols, where distinguishing features appear at fixed offsets. Being a product of byte distributions means that each byte offset is treated independently, as the following example illustrates.

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 not generally the case. Fortunately, this is only a problem if if the strings "`HOST`" and "`HEAT`" occur in *another* protocol, which would cause it to be confused with HTTP.

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 group.

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 property is why the relative entropy of two cells, which consist of two independent distributions on flows per Premise 2, is just the sum of each direction's relative entropy.

4.1.3 Merge

The merge operation simply returns a weighted average 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 group from which the first cell was constructed divided by the total number of sessions in the protocol equivalence groups of the first and second cells.

4.1.4 Score

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

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

4.2 Markov process Model

Like the product distribution model, the Markov process model relies on introducing independence between bytes 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 to 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

[1] 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.

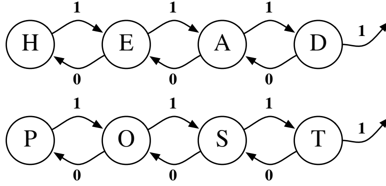

Figure 1: A Markov process for generating the strings "HEAD" and "POST" with each string chosen according to the probability of H and P in the initial distribution. Irrelevant nodes have been omitted for clarity.

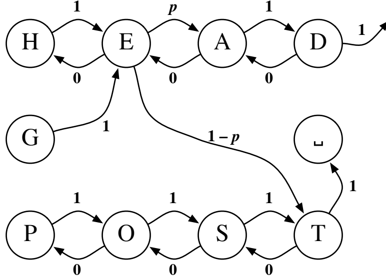

Figure 2: Attempting to add the string "GET␣" to a Markov process for generating the strings "HEAD" and "POST."

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.

In general, for a fixed sequence of adjacent nodes (corresponding to a sequence of bytes), there may be paths of different lengths ending with this sequence of nodes. This means that the process can capture distinguishing strings that are not tied to a fixed offset. As such, the Markov process model may be well-suited for text protocols.

The probability distribution on length-n flows defined by the above Markov process is described 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 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 $\pi(\mathtt{H}) = p$, $\pi(\mathtt{P}) = q$, and $\pi(u) = 0$ for $u \neq \mathtt{H}, \mathtt{P}$.

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 "GET␣" 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

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). This representation differs from the previous two approaches in that it captures much more structural information about the flows from which it is built. 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

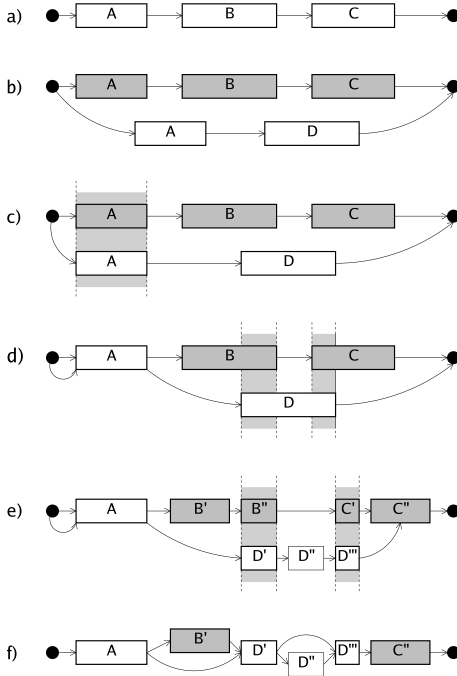

Figure 3: Constructing a CSG: introduction of a new path with subsequent merging of nodes. (a) A CSG with a single, three-node path. (b) An LCS (in white) is inserted as a new path. (c) New node A already exists and is therefore merged with the existing node. (d) New node D overlaps partially with existing 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.

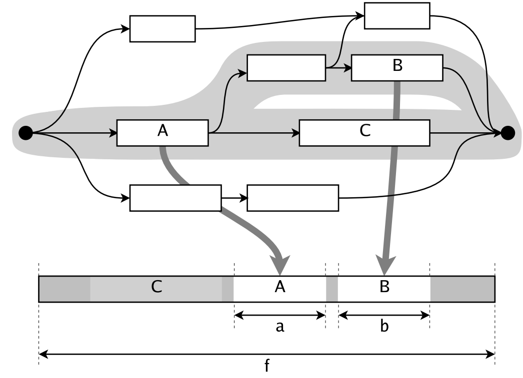

Figure 4: Scoring a flow against a CSG. The labels of nodes A, B, and C occur in the flow at the bottom. The shaded area in the graph indicates all paths considered for the scoring function. While the path containing A-C would constitute the largest overlap with the flow, it is not considered because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

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 does not make it clear which graph is being referred to, 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. Labels 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 [21]. 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 [16, 22]. 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 $(\overrightarrow{P}, \overleftarrow{P})$, we put two CSGs into each cell, one per flow direction. We will now describe the realization of the four cell methods in CSGs.

5.1 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, this completes the insertion process. 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 do not 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. Figure 3 shows the node merge and split processes during insertion of an LCS.

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. 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 until the limit is obeyed. To reduce the risk of evicting nodes prematurely, we use an additional, smaller *soft limit*, exceeding of which leads to node removal only if the affected nodes are not important to the graph's structure. 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. 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 occur, they are suppressed and a per-path redundancy counter is incremented. We do not currently limit the number of different paths in the graph 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.

5.2 Compare

To compare two CSGs, a graph similarity measure is needed. The measure we have implemented is a variant of feature-based graph distances [19]: 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 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 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:

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

5.3 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.

5.4 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. 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. By carefully numbering each path's links we can do this without actually walking down each candidate path. 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 notions and premises of constructing protocol models as presented in Section 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 5 summarizes the overall operation of our protocol description construction algorithm, from training cells starting with processing input sessions to merging cell clusters.

Equivalence Groups. We begin with the first step of assembling sessions into equivalence groups to construct cells, as illustrated in Figure 5a. For our implementation we assume that all communication sessions sharing the same *service key* belong 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 group because hosts typically communicate with servers at specified address-port combinations. In our experience, the granularity of this equivalence group is coarse enough to admit enough sessions in each group 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 Groups with Contact History. We augment service key equivalence groups 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 are 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 $C : p$,

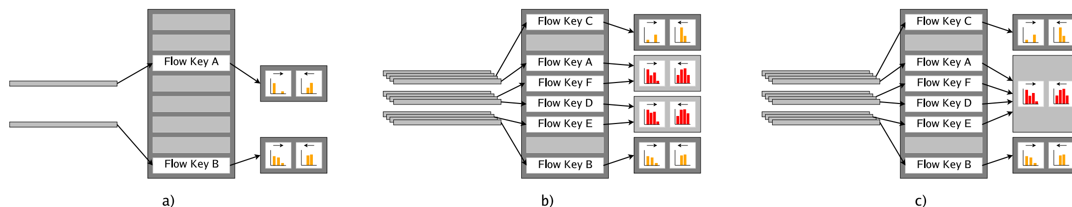

Figure 5: The Cell framework. (a) Flows are mapped to flow keys, stored in a hash table. Each flow key points to a cell; the cells are only lightly loaded and have not yet been promoted. (b) More flows have been added, multiple flow keys now point to the same cells. The first cells have been promoted for merging. (c) Cells have begun merging.

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 [12], although our application of external sources of equivalence information is relatively mild and not used during the classification process.

Cell Promotion, Comparison, and Merging. After inserting sessions into their respective cells, we need to *promote* them in preparation for clustering (Figure 5b). 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 sufficient 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 (bottom-up) clustering to construct a hierarchy of cells, and build larger cells by iteratively merging the closest pair of cells according to the classifier Compare operation.

Summary. The Cell framework is a realization of the protocol inference approach described earlier, providing a modular platform for evaluating various aspects of the traffic classification problem. Cell construction could benefit from more elaborate schemes of inferring equivalence groups. Moreover, the framework would provide us the flexibility to experiment with a variety of machine-learning approaches outside of agglomerative clustering to merge cells.

In the context of this paper, the framework allows us to flexibly evaluate the viability of product distributions, Markov processes, and Common Substring Graphs as protocol models independently of the schema for constructing equivalence groups or the clustering algorithms used after construction.

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 [17] to allow for more flexible testing of input traffic and because we anticipate using CSGs for other uses than traffic classification. We ran all experiments on a dual Opteron 250 with 8 GB RAM running Linux 2.6.

We used three traces for our experiments, each representing different network locations and traffic mixes. The "Cambridge" trace includes all traffic of the Computer Laboratory at the University of Cambridge, UK, over a 24-hour period on November 23, 2003. "Wireless" is a five-day trace of all traffic on the wireless network in the UCSD Computer Science and Engineering building starting on April 17, 2006. Finally, the "Departmental" trace collects over an hour of traffic from a UCSD department backbone switch at noon on May 23, 2006.

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. We chose this value for two reasons: first, it is the default setting that Bro uses, and second, smaller timeouts translate into more sessions to analyze. Note that erring on the early side only makes the classification task harder since we will pick up a mid-stream session as a novel one. 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 generically as just "TCP" or "UDP," we filtered it out of the trace. From the combined traces, flows labeled "TCP" comprised 1-6% over the three traces. Flows labeled "UDP" comprised 5% of the Cambridge traffic, 34% of the Wireless traffic, and 14% of the Departmental traffic. We attempt to classify excluded flows for the Departmental trace in Section 7.4.

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. [8].

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 four major TCP protocols (FTP, SMTP, HTTP, HTTPS) and four UDP protocols (DNS, NTP, NetBIOS Name service, and SrvLoc). For each of them, picked a destination service hosting at least 1000 sessions. We then 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, eight CSGs were loaded with each session's first message while we recorded node growth and usage. We have found that in no case was the hard limit insufficient and the soft limit was violated only by HTTP and NTP. We also measured 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

	Cambridge			Wireless			Departmental		
	total	learned	unlearned	total	learned	unlearned	total	learned	unlearned
Product	1.68%	0.50%	1.18%	1.78%	1.28%	0.51%	4.15%	3.03%	1.12%
Markov	3.33%	2.15%	1.18%	4.26%	3.75%	0.51%	9.97%	8.85%	1.12%
CSG	2.08%	0.90%	1.18%	4.72%	4.21%	0.51%	6.19%	5.06%	1.12%

Table 1: Misclassification for the three protocol models over the Cambridge, Wireless, and Departmental traces.

Protocol			Product			Markov			CSG		
		%	Err.%	Prec.%	Rec.%	Err.%	Prec.%	Rec.%	Err.%	Prec.%	Rec.%
①	DNS	26.28	0.09	99.94	99.78	0.61	97.89	99.97	0.45	98.82	99.52
	HTTP	12.24	0.07	100.00	99.99	0.09	100.00	99.98	0.74	99.91	99.99
	NBNS	44.89	0.35	100.00	99.25	0.40	99.82	99.31	0.17	99.71	99.99
	NTP	5.29	0.00	100.00	100.00	1.19	99.96	77.84	0.25	99.83	95.65
	SSH	0.22	0.14	68.39	100.00	1.10	17.39	100.00	0.05	99.22	100.00
②	DNS	23.14	0.04	99.88	99.93	0.29	98.88	99.99	1.97	94.37	97.59
	HTTP	0.67	0.27	76.02	97.54	0.09	90.68	99.93	0.22	76.87	99.38
	NBNS	6.94	0.00	100.00	100.00	1.96	78.06	100.00	0.81	90.34	99.97
	NTP	0.57	0.01	99.95	99.72	0.51	100.00	11.29	0.40	86.65	48.76
	SSH	0.44	0.17	75.28	100.00	0.00	99.63	100.00	0.00	99.99	100.00
③	DNS	54.78	0.26	99.90	99.95	1.90	97.13	99.98	1.43	98.47	99.15
	HTTP	9.17	0.38	97.46	99.62	0.33	97.21	99.72	1.21	95.14	97.19
	NBNS	7.03	0.01	100.00	99.81	1.25	85.66	99.81	0.33	96.04	99.45
	NTP	6.70	0.02	99.99	99.94	5.39	78.07	29.61	0.36	99.82	96.58
	SSH	0.08	0.08	68.81	81.82	0.09	0.00	0.00	0.03	95.40	82.01

Table 2: Error, precision and recall rates of select protocols. The second column is the proportion of the protocol in the entire trace. Trace key: ① = Cambridge (226,046 flows), ② = Wireless (403,752 flows), ③ = Departmental (1,064,844 flows).

three models (product distribution, Markov process, and CSG) 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 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 clusters to describe distinct protocols in the traffic, and then using those clusters to construct classifiers that recognize subsequent instances of the same protocols. Thus, if the system functions as intended, it is sufficient for the network administrator to label an *instance* of each protocol and thereafter all future traffic will be labeled correctly. We describe the automated phases of this experiment in more detail below.

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 agglomerative clustering (described in Section 6), and stop merging when the distance between all remaining clusters is greater than the pre-specified *merge threshold*. The distance metric is the weighted relative entropy for Product and Markov (Section 4), and approximate graph similarity for CSGs (Section 5.2). For our experiments, we set the merge threshold to 250 for Product, 150 for Markov, and 12% graph similarity for CSGs.

If more than one cell has a majority of flows with a Protocol X, then we include the cell with the largest number of Protocol X flows in the final set of cells and delete the others where X is in the majority. This accounting ensures that only one cell represents a protocol, and makes this experiment more challenging. (Again, recall that we apply labels to the flows *after* merging.)

7.2.2 Classification Phase

The goal of this phase is to associate each flow from the test trace with the cell that corresponds to the flow's protocol. To perform this classification, we take the clustered cells produced in Section 7.2.1, classify each flow with the protocol label of the closest matching cell, and compute the classification error rate.

7.2.3 Classification Results

Table 1 summarizes the misclassification rates for our framework under the three protocol models. For each of the Cambridge, Wireless and Departmental traces, we trained on the first half (in terms of duration) and tested on the second half. "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).

Overall, product distributions yielded the lowest total misclassification error across the traces (1.68–4.15%), while Markov Processes had the highest (3.33–9.97%) and CSGs fell in the middle (2.08–6.19%).

Table 2 presents misclassification, precision, and recall rates for select protocols within the three test traces. The product distribution model performed well over all protocols, particularly popular ones such as DNS, 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 comprised the majority of flows across the traces. 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).

The greatest weakness of Markov is misclassification of binary protocols such as NTP and NBNS. Unlike Product, Markov cannot take advantage of information related to multiple byte offsets—hence protocols that contain runs of NULL bytes, regardless of their offset, are undesirably grouped together.

CSGs also struggled slightly with binary protocols that product distribution successfully classified, such as NTP, but did best overall on SSH.

7.3 Unsupervised Protocol Discovery

The purpose of the protocol discovery experiment is to determine whether we can use our technique to automatically identify new and unknown protocols. Premise 3 suggests that new protocols will emerge as clusters that are distinct from any known protocol. To test this hypothesis, we perform the following experiment.

We split the trace into two equal parts, choose a protocol that is present in both parts, remove it from the first part, cluster both halves independently, and then match clusters in the two halves greedily. By greedily, we mean that the algorithm matches the closest pair of clusters between the two halves, removes them from consideration, and iterates on the remaining clusters. Finally, we assign protocol labels to each of the clusters *afterwards* to evaluate the effectiveness of the matching phase. Thus, if our system works perfectly, it will correctly place the missing protocol into a single homogeneous cluster that could then be labeled by the network administrator. Unlike the classification experiment we do not eliminate multiple cells labeled with the same protocol.

We expect that clusters of the same protocol straddling the two halves will match each other within the matching threshold. By contrast, we expect the protocol that we withheld from the first half would stand out, i.e., the withheld protocol present in the second trace would not have a close match from the first half.

Table 3 shows the results of the protocol discovery experiment over the two halves of the Departmental trace. We modeled protocols using product distributions, and withheld HTTP from the first half. The promotion threshold was 500 flows per cell, and the merge threshold was a weighted relative entropy of 250 (the same parameters as the classification experiment).

Our results indicate that there is a robust matching threshold for which nearly all protocols from the first trace match nearly all protocols from the second trace (covering 99% of all flows), while the *new protocol is left unmatched*. Specifically, the distance between the first incorrectly matched cells (RTSP and the previously excluded HTTP at 131.3) is three times larger than the distance between the last correctly matched cells (SNMP-SNMP at 44.5). An unintentional result was that two protocols that were not present in the first half, RIPv1 and RADIUS, appeared in the second half with a very distant match from the remaining cells of the first half (more than 300).

7.4 Classifying Excluded Traffic

For the experiments in Sections 7.2 and 7.3, we used Ethereal as an oracle to identify the protocols used by the flows in the clusters created by our models. Ethereal is not a perfect oracle, however, and there were flows that it could not identify, classifying them generically as "TCP" or "UDP". In the above experiments, we excluded those flows from the analyses because we could not compare clusters against a ground truth.

As a demonstration of the utility of our methodology for identifying flows using unknown protocols, next we identify these excluded flows. Specifically, we used product distributions to build protocol models over the excluded traffic using a promotion threshold of 500 and a merge threshold of 250 (as in the previous section). We then examined the clusters and manually identified several protocols unknown to Ethereal, including SLP (Service Location Protocol) Advertisements, game traffic, an implementation of Kademlia [13], HTTP over SSL, as well as various Web exploits. The flows using each of these protocols fell into separate clusters, once again showing that (1) our methodology can identify flows from one unknown application as distinct from another, and (2) it is sufficient for the network administrator to identify an instance of each protocol rather than all flows using a protocol.

First Half w/o HTTP				Second Half		
Cum. %	Ind. %	Protocol	Dist.	Protocol	Ind. %	Cum. %
0.49	0.49	Slammer	0.000	Slammer	0.43	0.43
1.07	0.58	ISAKMP	0.250	ISAKMP	0.44	0.87
9.01	7.93	NBNS	0.300	NBNS	7.12	7.99
9.66	0.65	TFTP	0.300	TFTP	0.42	8.41
70.52	60.87	DNS	0.399	DNS	56.46	64.87
71.37	0.85	SMB	0.595	SMB	0.72	65.59
71.40	0.03	SSDP	0.616	SSDP	0.03	65.62
72.22	0.82	SNMP	1.235	SNMP	0.80	66.42
76.22	4.00	SMTP	1.315	SMTP	4.00	70.41
76.52	0.30	SMB	1.548	SMB	0.27	70.68
77.02	0.50	DCERPC	2.011	DCERPC	0.47	71.15
77.12	0.10	SNMP	4.166	SNMP	0.09	71.24
78.08	0.96	BROWS.	4.168	BROWS.	0.82	72.06
78.15	0.08	Mssgr.	4.551	Mssgr.	0.43	72.49
78.47	0.32	KRB5	4.867	KRB5	0.29	72.78
79.39	0.92	DHCP	4.972	DHCP	0.78	73.56
79.52	0.13	LDAP	5.136	LDAP	0.11	73.67
86.39	6.87	SSL	5.900	SSL	5.72	79.39
86.47	0.08	SSL	6.127	SSL	0.04	79.43
87.39	0.91	YPSERV	6.509	YPSERV	0.82	80.25
87.49	0.11	SRVLOC	6.785	SRVLOC	0.09	80.35
89.00	1.50	POP	7.024	POP	1.24	81.58
89.19	0.19	SSL	8.136	SSL	0.06	81.65
89.49	0.30	SNMP	13.321	SNMP	0.28	81.93
89.55	0.06	SSH	15.871	SSH	0.07	82.00
89.60	0.05	KRB5	16.613	KRB5	0.03	82.03
89.64	0.04	IMAP	18.535	IMAP	0.04	82.08
89.85	0.20	CLDAP	19.496	CLDAP	0.15	82.23
89.91	0.06	Syslog	24.436	Syslog	0.06	82.29
98.01	8.10	NTP	38.452	NTP	6.84	89.13
98.16	0.15	NFS	44.493	NFS	0.04	89.18
99.52	1.36	SNMP	44.510	SNMP	1.23	90.40
99.58	0.06	RTSP	131.352	**HTTP**	9.46	99.87
99.88	0.30	SNMP	312.578	*RIPv1*	0.03	99.89
100.00	0.12	DAAP	470.046	*RADIUS*	0.11	100.00

Table 3: Unsupervised Protocol Discovery: Matched cells using Product Distributions between two halves of the Departmental trace where HTTP (in bold) was removed only from the first trace, ranked by match distance (strongest match to weakest). Protocol labels for cells were *not* available until after the matching was performed. Each row summarizes information for each pair of matching cells including their distance, their cumulative and individual percentage of flows in their respective halves, and the protocol label for the cell. The italicized protocols were also absent from the first half, although by coincidence and not by construction.

8. DISCUSSION

We have presented a systematic framework for unsupervised protocol inference—protocol inference based on partial correlations which are derived from unlabeled training data using simple assumptions about protocol behavior. In the remainder of this section, we discuss our experience with this framework, what we have learned in evaluating three different protocol inference algorithms, and how we envision our approach would be applied in practice.

Framework. While from an engineering point of view, a net-

Offset (Initiator to Responder)																			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
S	**S**	**H**	**-**	**2** 84 **1** 16	**.**	**0** 84 **9** 16 **5** <1	**-** 84 **9** 16	**O** 53 **-** 16 **P** 14 **S** 10 **h** 5	**p** 53 **3** 15 **u** 14 **e** 8 **t** 5	**e** 53 **T** 16 **.** 16 **c** 8 **t** 5	**n** 53 **2** 15 **T** 14 **u** 8 **p** 5	**S** 53 **.** 15 **Y** 14 **r** 8 **:** 5	**S** 53 **9** 10 **e** 8 **_** 7 **-** 7	**H** 53 15 **R** 14 **C** 8 **/** 5	**_** 53 **S** 15 **e** 14 **R** 8 ␣ 3	**3** 35 **4** 18 **S** 15 **l** 14 **T** 8	**.** 53 **H** 15 **e** 14 **_** 8 ␣ 3	**8** 28 15 **a** 14 **2** 9 **5** 9	**.** 40 **s** 16 **S** 15 **0A** 12 **p** 10
Offset (Responder to Initiator)																			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
S	**S**	**H**	**-**	**2** 53 **1** 47	**.**	**0** 53 **9** 47	**-** 53 **9** 47	**O** 52 **-** 47 **E** <1	**p** 52 **O** 47 **E** <1	**e** 52 **p** 47 **C** <1	**n** 52 **e** 47 **S** <1	**S** 52 **n** 47 <1	**S** 100 **3** <1	**H** 52 **S** 47 **.** <1	**_** 52 **H** 47 **2** <1	**_** 47 **3** 37 **4** 15 **.** <1	**.** 52 **3** 30 **4** 18 **9** <1	**.** 48 **9** 28 **2** 8 **1** 7	**p** 40 **1** 18 **9** 15 **7** 11

Figure 6: Individual byte distributions for the first 20 bytes of SSH (based on the Wireless trace) in both initiator-to-responder and responder-to-initiator directions. Non-printable characters are shown in hexadecimal. When more than one byte value occurs frequently at a given offset, the relative frequency (percent) is shown. The SSH Transport Layer Protocol specification (RFC 4253) dictates: "When the connection has been established, both sides MUST send an identification string. This identification string MUST be `SSH`–*protoversion*–*softwareversion* `SP` *comments* `CR LF`."

Offset (Initiator to Responder)

```
SReleCRenSecur Sec| 9A IE\ C2 1F D9 D3 B1 RT 0D C1 ED BD A4 X EF "2.1 0A 5) SSSenSH- 87 EF ureCAN-0-
SSSenSSH_RTY-2.065) SH_3 0A E6 icoreCA D3 18 E7 98 C8 12 forinSH Wil F2 = B9 C1 95 C0 A6 FE 01 CE nShe
SH-Recur 1D BrecunSH-2.0gz D4 FA { A1 htpe_4-1.9- 95 E0 B0 peC EE 0B L FF y AA C6 C2 B0 4.56 0A 10 92 B2 FF C7 Y
SenSH_2 0A = B5 \$:_3.0-SSSH-2.1.1 0A 8A -2SH-1+-2.065.5 C5 C8 E3 1E BB C0 F0 E1 17 k 83 89 F5 1B AC ~G&d^
SSSSSH_4.1 1D AF 3.8.cur 1.2.3.3.1uT_3 8C ] 0D 0A : A0 D4 A5 -3.1 0A C4 RTY-2.9-Opelendor
```

Offset (Responder to Initiator)

```
SSH-8 DenSH-1 0A r B6 F8 s EC 92 \$1l-Op1 0A 97 09 ED A0 89 3.0-1.0-OpebianSH_4.0z 89 9E 01 BA 4 0A A6 D4 92
SH_3.1 0A A4 AC _4H_4.0-2.9p1 B7 J A8 1B DB BE 1.1 0A .9pe.1 0A kG-1 0A AD 4.0-Op 08 Op1p1 0A B4 = C0 11 8B
SSH-OpenSSH_3.0-2 0A 82 ?VsanSSh B6 l 13 EE DF 85 CF C4 1C | 0F FA J 82 F7 C6 ) 17 LgenSSH-OpenSH_4.9
SH_3.8.0-2p1 Den-2.1 C4 08 L C7 81 8F A5 >) Y 80 91 E9 DE ubianSSSSSH_4.9-1.0-8. 1D + EC ) 86 l AA
SH-Open 12 dY FA 6w 83 13 1L99-2 0A A3 14 05 CD FF .1 0A E8 D2 B3 16 C3 0.1 0A J 8D * 80 DenSH_3.0-Op1.1 ( A5 ' D7 ^
```

Figure 7: Five samples of a Markov process distribution of both directions of SSH. Sequences like "`SH`" and "`SSSSSH`" appear because transitions `S` → `S` and `S` → `H` occur with roughly the same probability in the process.

work protocol is an intricate set of rules dictating the interaction between two processes, to the network, a protocol is just a distribution on byte sequences induced by real implementations and usage. It is on this ability to learn and distinguish these distributions that our protocol inference system is built. Doing so requires not only that protocol distributions differ measurably, but also that we be able to represent these distributions compactly while remaining completely oblivious to their design. This is the main challenge of designing protocol models. In our three models we have explicitly or implicitly relied on two techniques for overcoming this challenge and reducing the size and complexity of session distributions.

The first is to introduce independence assumptions into the model, as in the Naive Bayes assumption. For example, rather than treating the 64-byte distribution as a whole, we factor it into a product of 64 individual byte distributions—the product distribution model. At the cost of discarding correlations between bytes, we achieve an exponential reduction in space. Our product distribution model shows that this technique is remarkably effective, suggesting that the presence of certain attributes (rather than their correlation) is sufficient to distinguish protocols.

The second technique is to ignore infrequent features. In other words, the model assumes that distinguishing features are also frequent features. This plays a central role in the CSG model, which preserves more correlation by focusing on common high-frequency substrings.

Models. Our most successful model is product distribution. Because it is offset-based, we expected it to perform well on binary protocols where protocol "anchors" (i.e., byte sequences with low variance) occur at fixed offsets. Indeed, it did quite well on protocols such as NTP, and, surprisingly, also textual protocols like HTTP and SMTP. It turned out that these text protocols have a small number of distinguishing initial sequences that induce well-discriminating byte distributions. Figure 6 shows the individual byte distributions for SSH.

Our Markov process model was expected to capture "floating" strings — those not tied to a particular offset — as might appear in text protocols such as HTTP and SMTP. Unfortunately, a first-order Markov process cannot capture such strings perfectly. Moreover, it turned out that for many protocols we saw, the distinguishing strings *do* occur at fixed offsets (e.g., "`GET␣`" for HTTP or "`SSH`" for SSH). Figure 7 shows some samples of the Markov process distribution of SSH.

CSGs offer the unique benefit of providing protocol-intrinsic substrings in their entirety and with precise information about the location of their occurrence along with their frequencies. CSG's main strength, the focus on common substrings, is also its main weakness: only substrings that were observed during training can later be used for classification. Binary protocols make the presence of such strings less certain, though the fact that we used a minimum string length of 4 bytes shows that this is not a fundamental hurdle. Figure 8 shows a CSG model for the SSH protocol.

Applications. We envision two usage scenarios for partially-correlated protocol inference. The first, modeled by the classification experiment (Section 7.2), is one in which protocol models are learned in a semi-supervised manner. Rather than labeling training instances, as in the case of fully-supervising learning, only the constructed protocol models need to be labeled — a dramatic reduction in complexity. Our results show that our approach is competitive with existing supervising techniques (i.e., Haffner et al. [8]).

In the protocol discovery experiment, we explored an aspect of partially-correlated protocol inference not possible using supervised techniques, namely the discovery of new protocols. In this scenario, a new protocol would appear as a new cluster distinct from

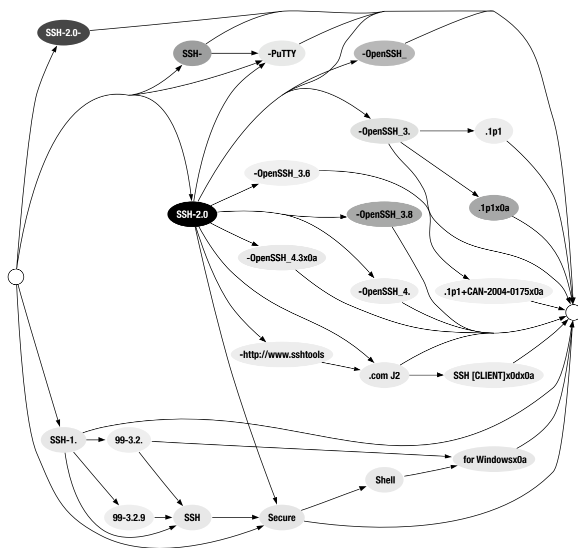

Figure 8: The CSG model for SSH, showing nodes with at least 50 paths. Redundant paths are aggregated into single arcs; darker nodes carry more paths.

known protocols (whether labeled or not). This allows for rapid discovery and description of new protocols. Our results show that our approach is effective for this problem as well.

Putting these together, consider a scenario in which a large increase in traffic is caused by a new peer-to-peer file sharing application and a new network worm outbreak. Using our approach, a network administrator could automatically determine that 70 percent of this otherwise "unknown" traffic belonged to one protocol and 30 percent to another. Moreover, the generated classifiers would be sufficient to discriminate future traffic using these protocols. Thus, after examining a single instance of each protocol and determining its associated application for labeling purposes, the administrator would once again have a comprehensive description of the traffic carried on their network.

Future Work. Our work naturally leads to several future directions of research. Perhaps most immediately, our models can be further improved to be more accurate and space-efficient. Although performance was not our primary objective, both the product distribution and Markov process models may be suitable for on-line processing; we plan to explore the performance aspect in future work. CSG's higher complexity makes operating at line-speeds challenging. However, we believe the unique strengths of CSGs can be put to use in related but less time-critical settings.

9. CONCLUSION

Identifying application-layer protocols has become an increasingly manual and laborious task as the historical association between ports and protocols deteriorates. To address this problem, we propose a generic architectural and mathematical framework for unsupervised protocol inference. We present three classification techniques for capturing statistical and structural aspects of messages exchanged in a protocol: product distributions of byte offsets, Markov models of byte transitions, and common substring graphs of message strings. We compare the performance of these classifiers using real-world traffic traces from three networks in two use settings, and demonstrate that the classifiers can successfully group protocols without *a priori* knowledge. Thus, labeling a single protocol instance is sufficient to classify all such traffic. In effect, we have substituted the painful process of manual flow analysis and classifier construction with the far easier task of recognizing a protocol instance.

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, Andrew Moore for access to the Cambridge trace, 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.

10. 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] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian. Traffic classification on the fly. *ACM SIGCOMM Computer Communication Review*, 36(2):23–26, April 2006.

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

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

[15] A. W. Moore and D. Zuev. Internet traffic classification using bayesian analysis techniques. In *Proceedings of the 2005 Conference on Measurement and Modeling of Computer Systems*, pages 50–60, 2005.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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`.

[22] 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.

[23] 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.