ACAS: Automated Construction of Application Signatures

Patrick Haffner, Subhabrata Sen, Oliver Spatscheck, Dongmei Wang

AT&T Labs-Research, Florham Park, NJ 07932
{haffner,sen,spatsch,mei}@research.att.com

ABSTRACT

An accurate mapping of traffic to applications is important for a broad range of network management and measurement tasks. Internet applications have traditionally been identified using well-known default server network-port numbers in the TCP or UDP headers. However this approach has become increasingly inaccurate. An alternate, more accurate technique is to use specific application-level features in the protocol exchange to guide the identification. Unfortunately deriving the signatures manually is very time consuming and difficult.

In this paper, we explore automatically extracting application signatures from IP traffic payload content. In particular we apply three statistical machine learning algorithms to automatically identify signatures for a range of applications. The results indicate that this approach is highly accurate and scales to allow online application identification on high speed links. We also discovered that content signatures still work in the presence of encryption. In these cases we were able to derive content signature for unencrypted handshakes negotiating the encryption parameters of a particular connection.

Categories and Subject Descriptors

I.5.4 [**Computing Methodologies**]: Pattern Recognition—*Applications*

General Terms

Algorithms, measurement

Keywords

Application signatures, application-level filter, machine learning

1. INTRODUCTION

A range of network operations and management activities benefit from the ability to gather per-application measurements in the middle of the network. These include traffic engineering, capacity planning, provisioning, service differentiation, performance/failure monitoring and root-cause analysis, and security. For example, enterprises would like to provide a degraded service (via rate-limiting, service differentiation, blocking) to P2P and extranet web traffic to ensure good performance for business critical applications, and/or to enforce corporate rules regarding access to certain types of applications and content; broadband ISPs would like to limit the P2P traffic to limit the cost they are charged by upstream ISPs; network engineers need to develop workload characterizations and traffic models for emerging applications, for network capacity planning and provisioning. All this requires the ability to accurately identify the network traffic associated with different applications. Some uses of application identification such as performance monitoring and service differentiation require online classification early in the connection. This paper explores the potential of automating the construction of signatures for accurate real-time application traffic identification early in the connection. We first outline the key issues and challenges and then present our contributions.

SIGCOMM'05 Workshops, August 22–26, 2005, Philadelphia, PA, USA.

1.1 Application identification

Application identification in IP networks, in general, can be difficult. Ideally, a network system administrator would possess precise information on the applications running inside the network, with unambiguous mappings between each application and its network traffic (e.g., port numbers used, IP addresses sourcing, and receiving the particular application data) etc. However, such information is rarely available, up-to-date or complete, and identifying the application-to-traffic associations is a challenging proposition. The traditional ad-hoc growth of IP networks, the continuing rapid proliferation of applications of different kinds, and the relative ease with which almost any user can add a new application to the traffic mix in the network with no centralized registration, are some factors contributing to this "knowledge gap". For instance, in the Virtual Private Network (VPN) world, it is not uncommon that while an operator is aware that a certain application (e.g., Lotus Notes) is being used in the enterprise, she possesses only partial information about the hosts and servers generating traffic for this application.

A traditional approach to traffic identification has been to use the TCP or UDP server port number to identify the higher layer application, by simply identifying which port is the server port and mapping this port to an application using the IANA (Internet Assigned Numbers Authority) list of registered[1] or well known ports. However port-based application classification has been shown to be unreliable for a variety of reasons. There is substantial empirical evidence that an increasing number of applications use random port numbers to communicate. A recent study [11] reports that the default port accounted for only 30% of the total traffic for the popular Kazaa P2P protocol, the rest being transmitted on non-standard ports. It also explains in detail what causes this trend. To summarize those findings (i) applications use non-standard ports mainly to traverse firewalls, circumvent operating system restrictions or hide from detection. (ii) In some cases server ports are dynamically allocated as needed; For example, FTP allows the dynamic negotiation of the server port used for the data transfer, (iii) port numbers can have limited fidelity - the same standard port can be used to transmit multiple applications. For

[1] `http://www.iana.org/assignments/port-numbers`

example, Lotus Notes transmits both email and database transaction traffic over the same ports, and `scp` (secure copy), a file transfer protocol, runs over `ssh` (secure shell), which is also used interactively on the same port (TCP port 22).

Instead of port-numbers, one possibility is to use specific features present in the application traffic to guide the identification. In particular this *signature-based application classification* approach parses packets for application-level information and tries to match the content of a TCP/UDP connection against common signatures found in the target application.

Following are some key challenges in developing the application signatures. First is the lack of openly available reliable, complete, up-to-date and standard protocol specifications for many applications. This is partly due to developmental history and partly a result of whether the protocols are open or proprietary. For some protocols (e.g., Gnutella), there exists some documentation, but it is not complete, or up-to-date. In addition, there are various implementations, such as Gnutella clients which do not comply fully with the specifications in the available documentation. For an application classifier to be accurate, it is important to identify signatures that span all the variants or at least the dominantly used ones. At the other end of the spectrum is an application like SONY's popular Everquest gaming protocol, which is developed by a single organization and therefore exhibits a more homogeneous protocol deployment, but is a proprietary protocol with no authoritative protocol description openly available. Finally, note that the application signatures are not fixed, but may change with time as the corresponding applications and their protocols evolve. The signature construction process has to be applied repeatedly to keep up with the change dynamics.

Existing approaches to application signature identification (e.g., [11, 4, 7]) involved a labor-intensive process combining information from available documentation with information gleaned from analysis of packet-level traces to develop potential signatures, and using multiple iterations to improve the accuracy and computation overhead. Such a painstaking manual approach will not scale if it has to be applied individually to the growing range and number of diverse Internet applications.

1.2 Contributions

In this paper, we explore the feasibility of automatically developing accurate signatures for individual applications. We use the term *signature* to refer to a set of conditions defined over a set of features in the application traffic. A *classifier* is used to classify network traffic using such a signature – it marks all traffic matching the conditions in the signature as belonging to the corresponding application.

Among key requirements for a signature composition algorithm, the resultant signatures (i) must be accurate, i.e., have low misclassification error rates, (ii) have low evaluation overheads to make it practicable to use the corresponding classifier for online real-time classification on high speed links, (iii) allow the classifier to identify the application early in the connection, (iv) be robust to asymmetric routing effects, and (v) must have good accuracy properties over an extended period of time. Finally, the algorithm should have wide applicability and be capable of developing accurate, efficient signatures for a range of different applications. The motivation for requirements (i) and (ii) are obvious considering the goal of identifying applications in real time on high speed links. Requirement (iii) is motivated by the fact that in some cases these types of signatures will be used to react quickly. For example, in a QoS enabled VPN this signature might be used to change the class of service (CoS) for the flow. Therefore, if the flow is classified late the benefits of such an approach would be small. This requirement does have one drawback in that a connection for which the initial part was not captured might not be classified correctly. We believe that in practice this will be a rare case in an ongoing measurement setup and, therefore, the benefits of early classification outweigh this drawback. Requirement (iv) stems from the fact, that a measurement point may capture only one direction of a bidirectional communication due to the prevalence of asymmetric routing in the Internet; therefore it is important that the derived signature is able to identify the application with high accuracy, irrespective of which direction it sees. Requirement (v) is very important, because the network traffic mix (and relative proportions) even at a given link changes with time, even if the target application itself is unchanged. To be practicable, the developed signature should be robust to these effects, and remain accurate over extended time periods.

Machine learning has been widely used in data analysis. It has been demonstrated to be an efficient technique for classifying texts, including filtering spam messages and classifying documents based on a list of feature words inside them (e.g. [1, 12]). Although machine learning has been used for network traffic classification, existing studies mostly considered statistical network flow attributes (e.g. [14, 13, 9]) such as packet size distributions. To our knowledge, this is the first work that explores the applicability of statistical machine learning techniques for identifying signatures based on application-level content. Specifically, we select three popular learning algorithms – Naive Bayes [8], AdaBoost [10] and Maximum Entropy [5], and use them to develop the application signatures.

We compare the performance of the different algorithms through extensive experiments. The key metrics for accuracy comparison are *error rate*, *precision* and *recall*. They are defined in Section 2.

To demonstrate the wide applicability, we use the three algorithms to develop signatures for a variety of network applications: ftp control, smtp, pop3, imap, https, http and ssh. Our evaluations of the resulting signatures, using real traffic traces from a tier-1 ISP, show that across the applications, the automatically constructed signatures exhibit (i) good accuracy properties (low error rate as well as high precision, and recall) based on inspecting only a modest initial portion of a communication, (ii) low evaluation overheads, and (iii) maintain good accuracy across several months.

The remainder of the paper is organized as follows. Section 2 describes how we model the application identification problem and its requirements (described above), and transform it to a statistical classification problem. Section 3 provides a high level description of the three machine learning algorithms that we apply to this classification problem. We present the experimental results in Section 4. Finally, Section 5 concludes the paper.

2. THE CLASSIFICATION PROBLEM

Before we introduce our approach let us define the problem we want to solve first:

Goal: Determine which application a flow belongs to by inspecting application layer information only (above the TCP/UDP header), where a IP flow is defined by a (protocol ,srcIP,destIP,srcPort,destPort) tuple.

Using this definition it is clear that our classification approach is resilient to: (i)The use of random port numbers; (ii)Changes in the network characteristics of an application such as average packet size, packet timing etc; (iii) Changes in the communication patterns of a client.

The only persistence this approach requires to correctly identify an application is the persistence in the applications signature itself. As we will show (Section 4) we found such persistent signatures for a range of applications. While in theory encryption techniques can be used to prevent or thwart content-signatures, in practice we were able to extract signatures for encrypted communication protocols such as ssh and https since both protocols perform an initial

handshake in the clear. Still, more generally, we expect that future identification techniques will rely on a composite approach combining different information such as statistical characteristics, content, and communication patterns etc. In this paper we explore the potential of one source of information- the application content.

As outlined in the introduction we use well-known machine learning algorithms to automatically derive the application signature. These algorithms require a training phase during which a set of pre-classified feature vectors (training set) is processed by the algorithm. At the end of this phase a classifier is returned which can be used to determine a probability that a feature vector for which the class assignment is not known belongs to a particular class. We map this functionality onto our classification goal as follows:

- Single Application Signature: We derive application signatures to decide if a particular flow belongs to an application or not. Therefore, we have two classes (application, non-application). If multiple applications need to be classified we will treat them as individual classification problems, with the classification of one application versus all other applications. Defining multicategory classification as a combination of *one-vs-other* binary classifiers has been shown to be attractive in terms of performance and simplicity [6].
- Raw Data: We encode the raw application level data as a feature vector. An alternate approach would be to extract features such as ASCII words from the data stream before presenting them as a feature vector to the learning algorithm. The drawback of a feature extraction phase is that it selects what is presented to the learning step, and structures which are not extracted as features are not visible to the learning algorithm. The potential drawback of the raw data approach is that the number of raw features can grow very large. However this is not a problem for the classifiers considered in this paper which are *large margin* or *regularized* thus able to generalize well with a large number of features (see next section). The raw data approach seems more suited to our particular problem domain where the actual useful features are unknown and can have a very wide range of structures depending on the protocol, its encoding etc.
- Initial data only: Our algorithm only considers the first n-Bytes of a data stream as features.There are three main motivations for this choice: (1) We want to identify traffic as early as possible. (2) For most application layer protocols, it is easy to identify application layer headers at the beginning of a data exchange. (3) This allows us to limit the amount of data the machine learning algorithms have to process. In case of a TCP connection, if TCP reassembly is required, we consider the first n-Bytes of the reassembled TCP data stream. Note that since we identify flows and not connections we treat each TCP connection as two independent reassembled TCP flows. This per-flow identification strategy was selected to facilitate developing a single per-application signature that is robust to asymmetric routing effects. i.e., that can identify the application with high accuracy by monitoring just one direction of the bidirectional communication.
- Discrete byte encoding: We encode the first n-Byte of a flow with a feature vector v with $n * 256$ elements. All components of v are initialized to 0. Then for each byte in the input stream, the component i*256+c[i] is set as 1, that is: $v[i * 256 + c[i]] = 1$, where i represents the position of a byte with value $c[i]$ in the reassembled flow. Therefore, the feature vector v has n non-zero components. This binary vector v is used as input by the machine learning algorithms and classifiers. The reason for this choice is that classifiers studied here are known to work well on binary vectors.

Another important property of the discrete representation is that byte values are *equidistant* (based on Euclidean distance). A binary classifier draws a separation hyperplane in the space where the examples live. If some examples were to be closer in this classifier input space, they would be considered as more similar. In other words, two examples with a smaller Euclidean distance are not separated easily, and tend to belong to the same class. For example, consider three byte streams c_1, c_2, c_3 of length one with $c_1[0] = [A], c_2[0] = [B], c_3[0] = [Z]$. If we would encode each byte in the Byte stream as a single integer the distance between c_1 and c_2 would be 1 whereas the distance between c_1 and c_3 would be 25. Therefore, the classifier has a harder time separating c_1 from c_2 than from c_3. This is counter-productive if for example, we want to identify all flows starting with an $[A]$. Using our feature vector encoding the distance between two feature vectors of equal length will always be identical removing this unwanted bias.

Using these mappings, we can directly apply the machine learning algorithms to our application classification problem.

Last we provide definitions of the three metrics we shall use to measure signature accuracy. Given a dataset of size S consisting of application and non-application flows, if a non-application flow is identified as an application flow according to the constructed signatures, we call it *false positive misclassification* and denote FP as the total number of false positive misclassifications. Similarly, if an application flow is identified as non-application according to the constructed signatures, we call it a *false negative misclassification* and denote FN as the total number of false negative misclassifications. Define the True Positive TP to be the total number of application flows that are correctly identified by the constructed signature. Then, the *error rate* is defined as $(FP + FN) * 100/S$. The *precision* is the number of actual application flows as a fraction of the total number of flows identified as belonging to the application, that is, $TP/(TP + FP)$. The *recall* is the the number of actual application flows as a fraction of the total number of flows in the set, that is, $TP/(TP + FN)$.

3. MACHINE LEARNING ALGORITHMS

This section offers a brief overview of the linear classifiers we use, their optimization processes, which we call learning, and their efficient implementation. We focus on three classes of algorithms: Naive Bayes [8], AdaBoost [10], and Maximum Entropy or Maxent [2]. In the case of Maxent, we use the Sequential L1-regularized Maxent algorithm (SL1-Max) [5]. These algorithms were chosen because of the scalability of their learning processes and the fact that their runtime implementation can be very efficient. They give us three ways to train a linear classifier[2] using very different frameworks. These algorithms can be interpreted as follows:

- **Naive Bayes** models, for each feature independently, a discrete distribution that gives its conditional probability given the class. Assuming these distributions are independent, the probability to observe an example given a class is the product of the probabilities to observe each feature given the class.
- **AdaBoost** incrementally refines a weighted combination of weak classifiers. In the process, the importance of examples that are still erroneous is "adaptively boosted".
- **Regularized Maximum Entropy** looks for a distribution over training samples with maximum entropy that satisfies a set of user-defined constraints. Regularization implies that we allow some slack in the satisfaction of these constraints.

Naive Bayes is provided as a baseline. While it does not minimize explicitely the training error and while the independence assumption it relies on is very far from the truth, it often provides reasonable performance [8]. In addition , the learning procedure is very simple and *incremental*: a learning example just has to be processed once to be learned, it does not need to be stored, and it can be used to further

[2]The runtime implementation of this linear classifier, which is the dot product between the feature vector and the weight vector, is independent of which of the algorithms is used.

train an deployed system. A system based on Naive Bayes can be continuously updated as new labeled data becomes available, this is why it is widely used for applications such as SPAM filtering [1].

The rest of this section briefly introduces AdaBoost and SL1-Max for Maximum Entropy, where better control over performance is obtained at the expense of a more complex and non-incremental training procedure. Both algorithms have shown excellent performance on other large scale problems [6]. In that study, which focused on spoken language classification, these algorithms have a learning time that scales as $O(M)$ where M is the number of examples in the training set. They are particularly efficient when the data is *sparse*, that is, the proportion of non-zero input features is small. This is the case in the present study, as only 1 feature out of 256 is non-zero.

AdaBoost was introduced as a sequential algorithm [10]. It selects at each iteration a feature k and computes, analytically or using line search, the weight w_k that minimizes a *loss function*. In the process, the importance of examples that are still erroneous is "adaptively boosted", as their weight in a distribution over the training examples that is initially uniform is increased. It was only later that this greedy sequential optimization process was shown to be guaranteed to converge and to be more efficient than other methods minimizing the same loss function [3]. Given a training set associating a target class y_i to each input vector $\mathbf{x}_i$, AdaBoost sequential algorithm looks for the weight vector $\mathbf{w}$ that minimizes the exponential loss (which is shown to bound the training error):

$$\mathcal{C} = \sum_{i=1}^{M} \exp(-y_i \mathbf{w}^T \mathbf{x}_i) \quad (1)$$

AdaBoost also allows a *log-loss* model (used in this paper), where the goal is now to maximize the log-likelihood of the training data $\log(\prod_i P(y_i|\mathbf{x}_i))$. The posterior probability to observe a positive example is $P(y_i = 1|\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T \mathbf{x}_i)}$.

Maxent relies on probabilistic modelling. Suppose we solve our classification problem by looking for the class which maximizes a distribution $argmax_y P(y|\mathbf{x})$. What is a reliable way to estimate this distribution on the training data without overfitting? First, how well this distribution matches the training data is represented by constraints which state that features must have the same means under the empirical distribution (measured on the training data) and under the expected distribution (obtained after the training process). Second, this distribution must be as simple as possible. This can be represented as a constrained optimization problem: find the distribution over training samples with maximum entropy that satisfies the constraints. Using convex duality, we obtain as a loss function the Maximum Likelihood. The optimization problem is applied to a Gibbs distribution, which is exponential in a linear combination of the features:

$$P(\mathbf{x}) = \frac{\exp(\mathbf{w}^T \mathbf{x})}{Z} \quad (2)$$

with $Z = \sum_{i=1}^{M} \exp(\mathbf{w}^T \mathbf{x}_i)$. A description of the derivation of this function from the constrained optimization problem is beyond the scope of this paper and can be found elsewhere [2]. SL1-Max is one of the fastest and most recent algorithms to estimate Maxent models. It offers a sequential-update algorithm which is particularly efficient on *sparse* data and allows the addition of L1-regularization to better control generalization performance.

An implementation of AdaBoost and SL1-Max that is optimized using *partial pricing* strategies is provided in the LLAMA software package [6]. AdaBoost, which is large-margin and implicitely regularized, and SL1-Max, which is explicitely regularized, have also been shown, both in theory and experimentally, to generalize well in the presence of a large number of features. Regularization favors simple model by penalizing large or non-zero parameters. This property allows the generalization error, i.e. the error on test data, to be bounded by quantities which are nearly independent of the number of features, both in the case of AdaBoost [10] and SL1-Max [2]. This is why a large number of features, and consequently a large number of classifier parameters, do not cause the type of overfitting (i.e. learning by heart the training data) that used to be a major problem with traditional classifiers.

Because of the present focus on linear classifiers, comparisons using non-linear Support Vector Machines, whose classification capacity is more powerful, but whose learning time can be slower ($O(M^2)$), will be kept for a future study.

Dataset	Training			Testing	
	8hr	4hr	1hr	8/2004	3/2005
ftp ctrl	6,678	3,255	824	7,152	490
smtp	343,744	172,647	43,987	363,062	208,399
pop3	100,472	49,913	13,376	103,150	43,583
imap	1,512	545	240	2,183	535
https	48,763	26,747	6,812	59,060	27,604
http	547,813	263,876	76,308	649,074	260,441
ssh	797	759	9	60	341
Total	1,242,515	614,773	165,815	1,381,533	1,065,018

Table 1: Number of training and test vectors in data sets

4. EVALUATION

To evaluate the performance of our approach we focus on building application layer signatures for ftp control, smtp, pop3, imap, https, http and ssh. We chose these applications for multiple reasons: (1) They cover a wide range of application classes in today's Internet including interactive, large file transfer, transactional, and encrypted communication-based applications; (2) It is easy to build the required preclassified training and test sets for these applications since they are mainly still using their default ports. We therefore used default port numbers to build the training and test sets.

In practice we would expect that the training sets would be built by other means which would allow us to include applications which do not use default port numbers. Possible approaches in constructing the training sets include the ones described in[7] as well as monitoring of traffic on a VPN type network in which traffic can be mapped to applications based on server IPs. In either case the resulting signatures could then be utilized in other locations. We are still investigating the best way for constructing such training sets.

4.1 Experimental Setup

To build the training and test sets we collect more than 100GByte of packet traces on a high speed access network serving more than 500 residential customers. The training data was collected in August 2004, the test data was collected in August 2004 and March 2005. For training we use training sets covering a total of 1,4, and 8 hours of traffic. Each set consists of 4 partitions which are equally spaced within a 24 hour period to account for variations in the applications and traffic mix during a daily cycle (for example each 8 hour set consists of four 2-hour long data partitions spaced 6 hours apart).

After compiling these data sets we processed them as follows: (i) Reassemble every TCP connection into two unidirectional flow records; (ii) Determine the server port of each flow by analyzing the TCP handshake; (iii) Generate a feature vector using the algorithm described in Section 2 while only considering at most the first 64 and 256 Bytes of a reassembled TCP flow (less if the connection did not have that many bytes in a particular direction); (iv) Build a class file for each application. A feature vector for a flow is classified as belonging to the class if the server port of the flow matches the default server port for the application to be classified.

Application	Training set	Training User Time	Algorithm	Error Rate in %	Precision	Recall	$\|\mathbf{w}\|$
ftp control	8hr	4h53m17.86s	AdaBoost	0.016	0.996	0.971	612
smtp	8hr	7h33m58.07s	AdaBoost	0.031	0.998	0.999	480
pop3	8hr	5h44m36.53s	AdaBoost	0.039	0.995	0.999	356
imap	8hr	12m2.16s	AdaBoost	0.000	1.000	0.999	189
https	8hr	7h28m39.37s	AdaBoost	0.258	0.992	0.946	271
http	8hr	1h0m17.06s	Maxent	0.508	0.990	0.999	5666
ssh	8hr	20m54.00s	AdaBoost	0.001	1.000	0.866	74

Table 2: Best classification results for each application considering the 8/2004 test set.

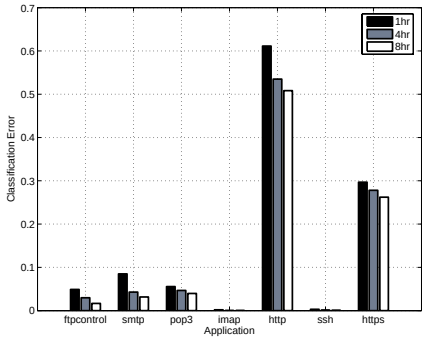

Figure 1: Application Error Rate for different training set sizes.

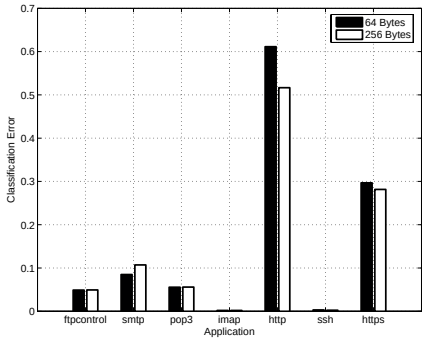

Figure 2: Application Error Rate for different vector lengths.

Table 1 shows the summary statistics of the resulting data sets.

4.2 Accuracy Results

In our first experiment we trained one classifier for each application using all three algorithms and all three training sets. The feature vectors in the training sets represent the first 64 Bytes of each TCP flow. We then evaluated which algorithm and training set resulted in the best error rate for a particular application in the 8/2004 test set. After that, we apply the resulting classifier for each application to classify the flows in the test set. For each flow, the classifier returns a probability value to indicate whether the flow belongs to an application. We classify a flow belonging to the application if the probability for the flow is at least 0.5 [3].

As shown in Table 2 for all applications the longest training set yielded the best result and for all applications but one AdaBoost performed the best. Maxent performed slightly better for http (less than an improvement of 0.005% in the Error Rate). An explanation for this good performance of AdaBoost is that the data is clean and contains very few outliers (because of the way it was generated: this is confirmed by the very low error rate). AdaBoost may not perform so well if the data is noisier [6]. Overall our error rate is below 0.51% for all applications we considered, that is, we correctly classify more than 99% of all flows using our approach. In addition the precision is above 0.99 for all applications, indicating very low levels of classification noise. The Recall which indicates how many connections we would have found in the test set is above 0.94 for all application with the exception of ssh. The poor performance of ssh is most likely caused by the fact that we only had 797 ssh training vectors in our training set with more than 1.2 million training vectors.

As stated above, for each classifier of a particular application we considered each vector to be classified as belonging to the application if the returned probability is at least 0.5. To evaluate the impact of this choice we varied the threshold in 0.01 increments from 0 to 1 and calculated the best error rate for each application using the optimal threshold for our test set. We know that a preclassified test set should not be used to learn a threshold, however, using this approach we can compute an upper bound on how much improvement we could expect if another threshold is used. For all application we found that the improvement in error rate over the 0.5 threshold was less than 0.03%. Therefore, it is safe to use 0.5 in practice.

Since Naive Bayes classifiers are well known and extremely simple one interesting question to ask is how much improvement do we get from using more sophisticated machine learning algorithms on our application layer data. Compared to the best AdaBoost classifier on the 8/2004 test set, the error rate obtained with Naive Bayes is 4 to 12 times larger, which is a considerable difference. Note that we used default smoothing probabilities, and that their choice could be critical for Naive Bayes performance. However, none of the other algorithms benefited from any task specific tuning, as we expect machine learning to avoid any form of human intervention. So clearly a more sophisticated approach provides substantial payoffs.

4.3 Choosing the Parameters

Another aspect to examine is how large a training set is required to achieve acceptable results. Figure 1 shows how the error rate differs for the different applications using AdaBoost and considering the first 64Byte of each flow as the classification vector. Even though the absolute error rate is small even for a 1 hour training set the error does improve with longer training sets. In particular the improvement from a one to a four hour training set is noticeable for all applications. In practice the training set size is limited by the memory available during the learning process (see next subsection).

A similar question to consider is how many bytes at the beginning of each flow should be considered. To evaluate this we computed the error rate for the different applications using AdaBoost and the 1 hour training set varying the Bytes considered from 64 to 256. Figure 2 shows the results of this experiment. Overall the results seem quite stable. In fact, considering more data slightly decrease the Error Rates of http and https while increasing the Error Rate of smtp. This indicates that for the applications considered there is enough information in the first 64 Bytes of each flow to identify the application. In fact more data leads to overfitting in the case of smtp.

[3]Naive Bayes, AdaBoost and Maximum Entropy outputs correspond to estimates of the conditional probability $P(y = 1|\mathbf{x})$ where $y \in \{-1, +1\}$ is the class to be recognized. Class $+1$ is recognized if $P(y = +1|\mathbf{x}) \geq P(y = -1|\mathbf{x})$, that is $P(y = 1|\mathbf{x}) \geq 0.5$.

4.4 Signature Durability

Another important question is for how long a signature can be used. Obviously the answer to this questions depends on many factor such as the release schedule of new versions of a particular application as well as changes in user behavior. To understand how stable the signatures shown in Table 2 are over a longer period of time we compute the result for the algorithms and training sets shown in the table for 3/2005 set instead of the 8/2004 set shown. Comparing those results to the earlier test set shows that the error rates for ftp-control, smtp, pop3 and https actually improve slightly. Whereas the error rates for ssh and imap increase to approximately 1% and the http error rate reaches 2.2%. In this set Precision is above 0.94 for all applications, whereas Recall is above 0.91 with the exception of imap which has a recall of 0.82. Overall this indicates that our classifiers maintain quite good performance on test data 7 months newer than the training data. An important practical implication is that the signatures can be used for long time periods without recourse to frequent retraining which can be an expensive process.

4.5 Performance Overheads

In this section we consider both the performance and memory requirements of building a classifier as well as the performance and memory requirements of executing it on a data feed which needs to be classified. The training times for the best classifiers are shown in Table 2. As shown the training time varies from slightly more than one hour to less than 8 hours of user time on a single 900Mhz UltraSparcIII processor using AdaBoost for all but the http application for which we use Maxent. Generally the Maxent algorithm terminates within less than 61 minutes whereas this version of AdaBoost takes longer (an Adaboost implementation at least as fast as Maxent is under deployment). Considering that the training phase is performed infrequently off line and that we used slow processors in our experiment the training times are acceptable for the task. The system we used had 192GByte of main memory which in theory limited our training data to 750M examples assuming a training vector which is based on 64 bytes of data. Therefore, the memory is more than enough for our experiment. In fact, all results presented in this paper are computed with less than 4GB of memory requirements.

The memory and performance requirements for real time classification depend on the feature vector $\mathbf{x}$, the weight vector $\mathbf{w}$ (which represents the classifier parameters) and the algorithm chosen to compute the dot product of the two vectors. This algorithm depends on whether these vectors are represented with their *full* list of values (including the zeros) or a *compacted* list of index/values, where the zeros have been removed. As we encode the first 64 bytes of each TCP flow into a feature vector, the full feature vector representing a TCP flow has $64*256 = 16384$ components, but its compacted version only requires 64 entries (see Section 2). Similarly, the classifier weight vector $\mathbf{w}$ has $|\mathbf{w}|$ none-zero elements and can also be compacted. We show the number of such elements for our classifiers in Table 2. The largest such classifier contains 5666 non-zero elements with the typical containing less than 1000 non-zero elements. Again we consider those vectors to be sparse. Computing the dot product of two sparse vectors is most easily done by storing each vector as an ordered list (by offset) of non-zero elements. Then we can compute the dot product by traversing both lists (similar to a merge operation). This has an overhead of $O(|\mathbf{w}| + |\mathbf{x}|)$ compare and add operations where $|\mathbf{x}| = 64$ is the number of non-zero elements in the feature vector.

For example, if we consider the classifier with the largest $\mathbf{w}$ in our experiments we need $5666 * 8 = 45,328$ Bytes of memory to store $\mathbf{w}$ which is small enough to fit into first level cache of most modern CPUs. Classifying a feature vector representing 64Bytes of a TCP flow then requires at most $5666 + 64 = 5730$ compare and add operations with most classifiers requiring less than 600 such operations (see Table 2). This overhead is clearly low enough to facilitate application layer classification at high speed links.

5. CONCLUSION

In this paper we explored the feasibility of automatically identifying application signatures. In particular, we applied three machine learning algorithms to automatically identify the signatures for a wide range of applications. We demonstrated that two sophisticated machine learning algorithms work well, that is, the automatically constructed signatures can be used for online application classification with low error rate, high precision and high recall, based on examining a modest amount of information at the beginning of the communication. The resulting signatures remain highly accurate to traffic variations on the time scale of several months. We also evaluated the computational complexity of applying the constructed signatures for realtime application classification. Our results show that the constructed signatures are suitable for online application classification on high-speed links.

6. ACKNOWLEDGMENTS

The authors would like to thank Lee Breslau, Jennifer Rexford and Robert E. Schapire for many helpful discussions.

7. REFERENCES

[1] I. Androutsopoulos, J. Koutsias, K. Chandrinos, G. Paliouras, and C. Spyropoulos. An evaluation of naive bayesian anti-spam filtering. In *Proceedings of the Workshop on Machine Learning in New Information Age*, Barcelona, Spain, 2000.

[2] A. L. Berger, S. A. Della Pietra, and V. J. Della Pietra. A Maximum Entropy Approach to Natural Language Processing. *Computational Linguistics*, 22(1):39–71, 1996.

[3] M. Collins, R. E. Schapire, and Y. Singer. Logistic Regression, AdaBoost and Bregman Distances. In *Proceedings of COLT'00*, pages 158–169, Stanford, CA, 2000.

[4] C. Dewes, A. Wichmann, and A. Feldmann. An analysis of internet chat systems. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, October 2003.

[5] M. Dudik, S. Phillips, and R. E. Schapire. Performance Guarantees for Regularized Maximum Entropy Density Estimation. In *Proceedings of COLT'04*, Banff, Canada, 2004. Springer Verlag.

[6] P. Haffner. Scaling Large Margin Classifiers for Spoken Language Understanding. In *Accepted for Publication in Speech Communication*, 2005.

[7] A. Moore and K. Papagiannaki. Toward the accurate identification of network applications. In *Passive & Active Measurement Workshop*, Boston, USA, March 2005.

[8] I. Rish. An empirical study of the naive bayes classifier. In *Proceedings of IJCAI-01 workshop on Empirical Methods in AI"*, pages 41–46, Sicily, Italy, 2001.

[9] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-service mapping for qos: A statistical signature-based approach to tp traffic classification. In *Proceedings of ACM SIGCOMM Internet Measurement Conderence (IMC'04)*, Sicily, Italy, October 2004.

[10] R. E. Schapire. The boosting approach to machine learning: An overview. In *MSRI Workshop on Nonlinear Estimation and Classification*, 2002.

[11] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proceedings of World Wide Web Conference*, NY, USA, May 2004.

[12] S. Souafi-Bensafi, M. Parizeau, F. Lebourgeois, and H. Emptoz. Bayesian networks classifiers applied to documents. In *Proceedings of ICPR*, Québec, Canada, 2002.

[13] S. Zander, T. Nguyen, and G. Armitage. Self-learning ip traffic classification based on statistical flow characteristics. In *Passive & Active Measurement Workshop*, Boston, USA, March 2005.

[14] D. Zuev and A. Moore. Traffic classification using a statistical approach. In *Passive & Active Measurement Workshop*, Boston, USA, March 2005.