Networking 2013 1569698247

SANTaClass: A Self Adaptive Network Traffic Classification System

Alok Tongaonkar, Ram Keralapura, Antonio Nucci
Narus Inc., USA
{alok, rkeralapura, anucci}@narus.com

***Abstract*—A critical aspect of network management from an operator's perspective is the ability to understand or classify all traffic that traverses the network. The failure of port based traffic classification technique triggered an interest in discovering signatures based on packet content. However, this approach involves manually reverse engineering all the applications/protocols that need to be identified. This suffers from the problem of scalability; keeping up with the new applications that come up everyday is very challenging and time-consuming. Moreover, traditional approach of developing signatures once and using them in different networks suffers from low coverage. In this work, we present a novel fully automated packet payload content (PPC) based network traffic classification system that addresses the above shortcomings. Our system learns new application signatures in the network where classification is desired. Further more, our system adapts the signatures as the traffic for an application changes. Based on real traces from several service providers, we show that our system is capable of detecting (1) *tunneled* or *wrapped* applications, (2) applications that use random ports, and (3) new applications. Moreover, it is robust to routing asymmetry, an important requirement in large ISPs, and has a very high (>99.5%) detection rate. Finally, our system is easy to deploy and setup and performs classification in real-time.**

I. Introduction

A critical aspect of network management from an operator's perspective is the ability to understand or classify all traffic that traverses the network. This ability is important for traffic engineering and billing, network planning and provisioning as well as network security. Rather than basic information about the ongoing sessions, all of the aforementioned functionalities require accurate knowledge of what is traversing the network in order to be effective.

Network operators typically rely on *deep packet inspection* (DPI) techniques for gaining visibility into the network traffic [11], [15], [18]. These techniques inspect packet content and try to identify application-level protocols such as Simple Mail Transort Protocol (SMTP) and Microsoft Server Message Block (SMB) protocol. In this paper, we refer to application-level protocols, with a distinct behavior in terms of communication exchange, simply as *applications* (or sometimes as protocol) for ease of understanding. In commercial world, DPI based techniques commonly use application signatures in the form of regular expressions to identify the applications. Signatures for each application are developed manually by inspecting standards documents or reverse engineering the application.

However, use of DPI based approaches in large network shows that the coverage is usually low, i.e., a large fraction of traffic is unknown. The main reason for the low coverage in commercial solutions is the lack of signatures for many applications. Many applications like online gaming and p2p applications do not publish their protocol formats for general use. Reverse engineering the several hundred new p2p and gaming applications that have been introduced over the last 5 years requires a huge manual effort. As a consequence, keeping a comprehensive and up-to-date list of application signatures is infeasible.

Recent years have seen an increasing number of research work that aim to automatically reverse engineer application message formats. These techniques work well when they are used for targeted reverse engineering i.e., they have access to either the binaries for the application [6] or the network traffic belonging to an application [9], [21]. There are two main drawbacks of these approaches that severely limit the use of these techniques for automatic application signature generation. First, these techniques are unable to handle *0-day applications*, i.e., applications that are seen for the first time in the network. Clearly, it is impractical for a network operator to obtain the binaries belonging to all the applications that all the network users ever install on their machines. On the other hand, network traffic based techniques are also impractical for 0-day application signature generation as they require the ability to identify the application in the first place in order to group all the application flows together.

The second problem that the automatic reverse engineering techniques fail to deal with is the *variations* in the application message formats. These variations may be due to the *evolution of applications* which may lead to addition or modification of features. For example, many SMTP servers now support newer extensions such as the use of keyword EHLO instead of HELO. If the signature for SMTP does not account for this, it will fail to match flows originating from clients using the extensions. Another common reason for the variation is the differences in the underlying OS. Many text-based network applications use *newline* as a delimiter. However, newline is represented by carriage return (CR) and linefeed (LF) on Windows and only by linefeed on Unix. The signatures need to account for such differences as well.

In this work, we present a novel approach for network traffic classification that overcomes the above shortcomings by learning the application signatures on the network where the

classifier is deployed. Our approach, which we call *Self Adaptive Network Traffic Classification*, aims to eliminate the manual intervention required to develop accurate payload based signatures for various applications such that they can be used for real-time classification. We built a Self Adaptive Network Traffic Classification system, called SANTaClass, that combines a novel automated signature generation algorithm with a real-time traffic classifier. Our system can be plugged into any network and it *automatically* learns application signatures tailored to that network. The signature generation algorithm is based on identifying *invariant patterns* and can handle text-based and binary-based as well as encrypted applications in a uniform way. Moreover, our system uses *incremental learning* to adapt to the changing nature of the network traffic by generating signatures for applications which were not seen before as well as newer versions of applications for which we have already extracted signatures. The main contributions of this work are:

• We propose and evaluate a novel methodology that *automatically* learns signatures for applications on any network without any manual intervention. These signatures reflect the applications seen on the deployed link and the signature set evolves as and how new applications traverse the link.

• We built an efficient system which combines automated signature generation process with real-time traffic classification such that the current set of signatures that are extracted are utilized to classify traffic in the future in a transparent fashion. We have currently deployed this system in more than 6 different Internet service providers and enterprise networks.

• Our experiments with real traffic from multiple ISPs shows that our methodology

- increases coverage by identifying new applications
- handles variations in applications due to varying implementations or application evolution
- has high accuracy when compared to the state of the art DPI systems
- adapts to changing network traffic without user intervention
- can extract signatures for several encrypted applications
- is robust to routing asymmetry.

• Finally, the *learn-on-the-fly* philosophy behind our system is a major paradigm shift from existing classification systems which use pre-loaded application signatures.

The rest of the paper is organized as follows. In Section II we describe the system design. Section III describes the complete system implementation. We present the experimental results in Section IV. We discuss related work in Section V. Finally, we conclude the paper in Section VI.

II. System Overview

SANTaClass is a completely automated network traffic classification system that involves real-time classification and unsupervised signature generation. The input to our system are full packets.

1 CS: EHLO MAIL.LABSERVICE.IT
2 SC: 250-IMTA01.WESTCHESTER.COMCAST.NET HELLO ...
3 CS: MAIL FROM:<DAGA@LABSERVICE.IT> ...
4 SC: 250 2.1.0 <DAGA@LABSERVICE.IT> SENDER OK

Fig. 1. SMTP Session 1

1 CS: EHLO QMTA03.COMCAST.NET
2 SC: 250-MAIL.LABSERVICE.IT SAYS EHLO TO ...
3 CS: MAIL FROM:<> ...
4 SC: 250 MAIL FROM ACCEPTED

Fig. 2. SMTP Session 2

A. Application Signatures

The key insight in generating signatures is that flows belonging to an application contain certain invariant parts such as keywords in text-based applications and fields like session identifiers in binary-based applications. These invariant parts can form the building blocks of application signatures. In this paper, we focus on text-based applications and omit binary-based applications due to lack of space. We note that the binary-based technique, which is based on identifying invariant bit patterns, is similar to the text-based technique that utilizes invariant string patterns as signatures.

Consider the flows belonging to two different sessions of SMTP shown in Figures 1 and 2. Client-to-server payloads are indicated by CS and server-to-client ones with SC. If we consider the client-to-server flow for session 1, it consists of payloads in step 1 and 3 concatenated together. Similarly, for session 2, client-to-server consists of payloads from step 1 and 3. It is clear that they share some common strings such as “EHLO” and “MAIL FROM:”. These common parts may or may not contain application keywords. We are interested in identifying such invariant parts and not necessarily identifying all application keywords since our goal is not to reverse engineer the application message formats but to generate signatures that can be used to identify the applications. We use “term” to refer to strings of arbitrary length. Terms which are present in multiple flows are referred to as “common terms”. The question that we try to address in this work is “*how can we generate application signatures from common terms?*”

A straightforward approach to using the terms as signatures is to use the presence of common terms in flows for classification. A simple scheme for weighting can reward terms that occur frequently in an application term set and penalize terms that are present in multiple term sets [24]. Such an approach is light-weight and depends only on the weighted terms. However this approach introduces false positives [17]. The problem of false-positives can be significantly reduced by providing additional context in the signatures.

The signatures can be augmented with context by considering the *sequence* (or *ordering*) of terms in flow content instead of considering the terms independently. This allows us to reduce the false-positives due to overly general or *loose* signatures. For example, the signature “application is X only if a flow contains term A followed by term B” is tighter than the signature “application is X if a flow contains terms A and B”. The latter signature will match a flow content that

has terms A and B in the order B followed by A, which may not be possible in application X. Such ordering relation can be represented as a Prefix Tree Acceptor (PTA), which is a trie-like deterministic finite state automata, i.e., it has no back edges [16]. Figure 3 shows the PTA for SMTP client-to-server. Note that each of the nodes has a self loop which allows arbitrary characters to be matched between terms. We omit these self loops from the figures in this paper for ease of understanding. We can see that the starting node in the PTA has two outgoing transitions corresponding to “EHLO” and “HELO”. This is because some SMTP flows on the network contain the older “HELO” keyword and others use “EHLO” which is supported in extended SMTP. In this paper, we use PTA and state machines interchangeably to refer to the representation of signatures as shown in Figure 3.

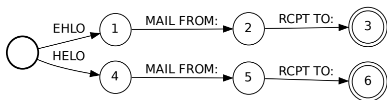

Fig. 3. PTA for SMTP c-to-s

Handling Encrypted Traffic. Any payload content based classifier will face certain limitations when the payload is encrypted. However, we observe that this is not a severe limitation due to the way applications use encryption. Typically, most applications have a clear-text part at the start of a session for negotiating parameters and such. This clear text contains invariant strings that helps us generate signatures for applications that use encryption. Since our technique relies on traffic from multiple communicating entities (explained in Section III-C), we can also use invariants such certificates within encrypted payload as application signatures. Moreover, the invariant bit patterns in encrypted payloads are captured as binary signatures (which we do not discuss in this paper). We note that the combination of these factors does not guarantee that we are be able to identify all the encrypted traffic but still allowed us to identify a large fraction (≈80%) of real-world encrypted traffic.

B. Design

Figure 4 shows the architecture of the system with blocks having real-time constraint shown in Green color. When SANTaClass receives full packets as input, the flow reconstructor module first reconstructs flows. Then the classifier tries to label these flows using any existing packet content based signatures. If the classifier successfully labels a flow, then the result is recorded in a database. The classification process for the flow ends. However, if the classifier cannot label the flow, then the flow is sent to the flow-set constructor which tries to group together flows belonging to each application into flow-sets. The signature generator extracts one signature for each flow-set. Finally, the distiller module distills any newly extracted signature by consolidating with existing signature for the application, and eliminating redundancies.

Given the above design, we can see that the SANTaClass system can easily tolerate false-negatives (flows that do not get labeled despite having a signature), but cannot tolerate false-positives (flows that are misclassified). The reason for this is the following. The proposed signature generation is an incremental process, i.e., the signature for an application is generated as and how the signature generator sees the flows belonging to the application. The system starts with no signatures in the database. When the first set of application flows enter the system, a new signature for the application is generated and populated in the database. Now the system has one signature. Henceforth, all the flows that belong to the application are classified and thus do not enter the automated training phase. Now, if the signature is not very accurate, then several flows that do not belong to the application may get misclassified as belonging to the application. These misclassified flows (i.e. false-positives) will never be available for training in the future and the errors in classification will continue to increase. Hence, in the signature generator, the goal is to ensure that if the system has to err then it should err on the false negative side and not the false positive side.

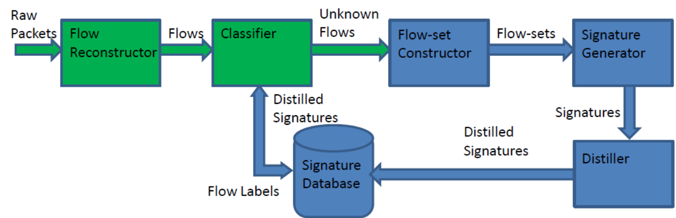

Fig. 4. SANTaClass Architecture

III. Implementation

In this section, we describe each of the system components in detail.

A. Flow Reconstructor

Flow reconstructor captures all packets flowing through a link and performs IP defragmentation and TCP reassembly. It maintains session information for every session. Each session is composed of two flows: client-to-server (henceforth referred to as c-to-s) and server-to-client (s-to-c). We consider each flow of the session independently since in the backbone one of the two directions is often missing due to routing asymmetry. Each flow is constructed by concatenating the transport layer payload from all packets in the given direction. What this means is that we ignore all headers up to and including transport layer (TCP/UDP). Maintaining the complete payloads in all packets in a given direction causes dramatic increase in space requirements as well increases the latency in classification. To overcome this, we store a specific number of bytes in each direction. This is a reasonable compromise as we expect most common terms to be present in the application layer headers, which are present at the start of the flow. In our implementation, we store a maximum of 1024 bytes of application data in each direction as we empirically found this value to be good for extracting strong signatures without causing noticeable performance degradation. Note that for the flows that terminate before producing 1024 bytes of payload,

all of the application data will be stored. From here on, we refer to the contents of a flow as payload. We convert the payload to upper case in order to improve the efficiency of the classifier since case sensitive string matching incurs space/time overhead compared to case insensitive matching.

B. Classifier

The classifier is responsible for matching every incoming payload against all signatures in the database and identifying all the matching signatures. The classifier can naively do this by iterating over the signatures and traversing the PTA by performing string searches for the outgoing transitions in the payload. The above approach is very inefficient as string search, which is an expensive operation, may be performed multiple times. At each state in a PTA, the payload is scanned multiple times for each of the terms on the outgoing transitions. Moreover, this search is repeated across multiple states (possibly belonging to different signatures) even if the terms are same. To overcome these redundancies, we developed a two phase classification system that uses efficient multi-pattern search to identify all terms present in a payload in a single scan and then use these terms to match only the signatures which contain these terms.

In the first phase, we use Aho-Corasick [3] as follows. We create a trie-like structure with failure links, called Aho-Corasick Trie (ACT), from all the terms present in all the signatures. This ACT helps us identify all the matching terms in a payload, ordered according to their offsets in the payload, and the set of signatures that contain each term, in a single scan of the payload. In the second phase, we iterate over each of the signatures that have at least one term matched by ACT, and match their PTA as follows. We maintain a pointer in the ordered list of terms that matched in a payload, called *current term pointer*, and a corresponding pointer to current state, called *current state pointer*. Starting from the *current term pointer*, we pick the first term in the matched term list that has an outgoing transition in the *current state*. We move the *current term pointer* to this term and take the transition by moving the *current state pointer* to the end state of the transition. If the new *current state* is a matching state, we can announce a match for the signature but continue matching to see if we can get a stronger match (i.e., match at a state which has a longer path length from start state). If no such term is found, then we can make no progress and stop this process. In this case, based on whether *current state* is accepting (or not) we announce success (or failure). We note that the ACT has to be reconstructed every time the signatures in the database change. However, the new ACT can be built in a background thread and hot-swapped with the old one to prevent any performance degradation.

C. Flow-Set Constructor

A critical component in the overall system is the flow-set constructor. The main goal of this component is to organize the incoming flows into buckets (or flow-sets) such that each bucket represents a particular application. The facts that an application can run on multiple ports, and multiple applications can run on the same port, make this problem hard to solve. If a bucket contains flows from multiple applications, then the signature extracted by the downstream component will result in inaccurate classification. Hence, it is critical to devise a strategy to accurately bucketize applications.

In this work, we perform bucketization in two steps. The first step is to use DNS information corresponding to the data flows that need to be bucketized. Most of the applications today (except for some p2p applications) rely on DNS to provide the name to ip-address resolution. In other words, a DNS query and response precedes an actual application flow. In this approach, we correlate the DNS information (i.e., the server-ip, client-ip, and domain name) with the data flow to identify the corresponding domain name. for example, a flow generated by Google Mail will be correlated to the domain name mail.google.com or gmail.com. We use the complete domain name as the "key" for the bucket and place the application flow into the bucket. If the bucket with the current "key" did not exist before, then we will create a new bucket and put the application flow as the first element in the bucket. For more details about the algorithm and the implementation, please refer to [4]. The bucket is considered full and sent to the signature generator based on a simple threshold.

For a flow that comes into the flow-set constructor, we first try to put it into a bucket based on the corresponding DNS domain name as described above. If we can successfully bucketize the flow, then we are done. If not, we proceed to the second step of bucketization. In the second step, we bucketize the flow using the following three values as the "key": the layer-4 protocol, the server port number, and the flow direction (i.e., s-to-c or c-to-s). Obviously a strategy like this will introduce flows from several applications into a single bucket. We counter this by using two steps: (i) Ensuring that the bucket is good statistical representation of the flows on the port, and (ii) Sophisticated clustering algorithm that groups various flows based on the similarity of their payloads.

To ensure we have a good statistically diverse set of flows inside a flow-set, we use several user configurable parameters while constructing flow-sets. A valid flow set should satisfy the following constraints: (a) *Server Diversity.* The total number of server ip-addresses in the flow set should be greater than a threshold (say, s_{th}). This ensures that the signature extracted is not specific to one server hosting a service. (b) *Number of Flows per Server.* To help reduce the impact of one server on the extracted signature, we bound the maximum number of flows that we consider for each server IP-address by a threshold (say $nmax_{sth}$). (c) *Total Flows Per Flow-Set.* The total number of flows in the flow set should be greater than a threshold (say $nmin_{th}$) to ensure that a flow set contains enough number of flows to represent a statistically good subset of application flows.

The flow-set formed using the above process is subject to a two-dimensional clustering process based on the similarity of the flow payloads. The algorithm that we use is similar to [23]. We will omit the details of the algorithm here, but mention

that the output from this algorithm will result in a flow-set with extremely cohesive set of flows.

D. Signature Generator

We developed a novel system for extracting the signatures from the flow-sets. It is composed of the following three components: (i) Common Term Set Extraction (ii) Common Term Set Refinement (iii) PTA Generation.

1) Common Term Set Extraction: The input to this component is a flow-set that contains the application payload content of each flow in the flow-set. Extracting common terms requires pairwise comparisons of the application payload content of all the flows in the flow-set. In other words, if there are n flows in a flow-set, then this operation requires $O(n^2)$ payload comparisons. To further increase the complexity, each payload comparison involves all common substring extraction - an operation that has the complexity $O(ab)$, where a and b are the lengths of the two payload strings that are being compared. Hence, the overall complexity of extracting all common substrings for a flow set has the complexity $O(n^2m^2)$, where n is the total number of flows in the flow-set and m is the average length of the payload strings in the flow-set. If we assume that a flow-set consists of a few thousand flows and the average payload length is 1000 bytes, the common substring extraction algorithm requires more than a million string comparisons - an impractical operation.

Hence, we first split a given flow set, F, into several smaller subsets[1], and extract common terms in each of these subsets independently. For every pair of payloads in each of the subsets we extract all the common terms and insert them in common term set CTS. Note that CTS contains only unique terms and hence, duplicates are eliminated.

2) Common Term Set Refinement: As noted before, we extract terms (i.e., the longest common substrings) by comparing two flow payloads with each other. The quality of the extracted terms could affect both the quality of final signatures and the efficiency of real-time classification. To ensure a high quality of extracted terms, we enforce a set of rules that accepts *good* terms and rejects *bad* terms. Here we present these rules.

Remove short terms. When multiple payloads are compared with each other, many short terms are extracted. However, these short terms add little value in determining whether a particular flow belongs to given application or not. We eliminate all terms that are shorter than a threshold, T_{len}. In our experiments, we found that a value of 4 for T_{len} is good for retaining important terms while discarding shorter terms like $OK+$.

Remove terms unrelated to applications. Typically, a flow originating from any application has certain fields, such as the *date/time* field, that always occur but do not have any relevance to the application. Hence we remove strings that identify day/month/year, such as "MON", "MONDAY", "JAN", "2010", "2011", and those identifying specific domains on the Internet, such as ".com", ".edu", etc.

[1]The upper bound on the number of flows in a sub-flow set can be controlled using a user specified parameter.

Identify and remove bad terms. Most of the flows that we see in the data traces carry several different parameter values that are usually numeric values. We eliminate any terms that does not contain at least two alphabetic characters, such as "2E00","/0/0/0/", "0.001", etc.

Remove low frequency terms. If the number of terms in the common term set is large it can potentially lead to PTAs with a large number of states and paths. To reduce the number of terms that we consider in the common term set, we define two thresholds: term probability threshold, P and the number of terms threshold, N. The term probability threshold selects only those terms that occur with a probability greater than P in the flow-set . The number of terms threshold selects at most the top-N terms with the highest probabilities. The terms that pass both of the above constraints are retained in the common term set and the rest are discarded.

Handle substrings. If we find that one term is a substring of another term, then we retain the term that has a higher probability and eliminate the other. If the probabilities happen to be the same, then we retain the term that is longer.

Add mutually exclusive terms. A problem that will be introduced by the above thresholds is that several important terms might be eliminated. For example, consider the popular HTTP protocol. There are several methods that can be used in this protocol like GET, POST, HEAD, PUT, DELETE, etc. Each of these methods might not have a high probability of occurrence; however when analyzing many http flows all of these methods can occur in the flows. If we set the term probability threshold, P to be high, then all of the terms representing these methods will get eliminated. To counter this problem we introduce *mutually exclusive term grouping* - a process by which terms are grouped together when two conditions are satisfied: (1) The terms that belong to the same group do not occur in the same flow payload, i.e., the terms occur mutually exclusively from each other, and (2) The combined probability of all the terms in a group should be at least equal to the term probability threshold, P. Note that the combined probability of a mutually exclusive term group is simply the sum of the probabilities of all the terms in the group. We add all the terms in the mutually exclusive group into the set of eligible terms.

3) PTA Generation from Terms: The inputs to this component are all the flows in a flow set and the common term set for the flow set. First, for every flow in the training set, we sort the common terms in the order of occurrence in the payload. We iterate through each of these terms in the order of occurrence in the flow payload and build the state machine starting from state 0 every time. If the transitions (i.e. the terms) are already part of the state machine, then the pointer to the current state is just forwarded. However, if the transition and states do not exist, then they are added to the existing state machine. If the term that is being examined is the last one in the sorted sequence in the flow payload, then we make the next state an accepting state.

E. Distiller

The signature generation module presented in previous section generates signatures from a given flow-set independent of other flow-sets, which may result in redundancies in the signatures. We have developed a distiller module to *distill* all the current signatures by resolving conflicts (i.e., overlaps), identifying and eliminating duplicates, and optimizing state machines. In the distiller module, we mainly accomplish the following tasks:

1) Eliminate Redundancy: The distiller is responsible for eliminating redundancy in the state machines as follows.

Identify and merge redundant state machines. Many applications, such as p2p, do not use a single standard port but can run on any one (possibly user configured) of a range of port numbers. This leads to the presence of same application in different flow-sets. If the state machines that are extracted in the signature generation module are identical, it indicates that a particular application could be running on many different ports. The distiller eliminates such duplicate state machines and tags the first extracted one with a label that indicates the application. Moreover, since our system may generate multiple state machines for the same application in different iterations, the distiller merges these state machines belonging to the same application.

Handle overlaps between state machines Several applications, although significantly different from each other, can share paths in their state machines. This typically occurs when different applications share some common message formats, such as the ones used for user authentication at the start of a session. These paths, when traversed by a flow, could lead to multiple labels which may or may not be conflicting with each other. The distiller identifies these overlaps and extracts them (i.e., the overlapping paths) to create new state machines with multiple labels (concatenation of labels from all the overlapping state machines) associated with them.

2) Optimize PTA: Our signature algorithm generates a trie-like automaton. The advantage of this is the ease of construction and sharing of states whenever the prefix of two paths are common. A disadvantage of this approach is that there is redundancy when paths share suffixes. In the distiller, we identify such redundancies and merge suffixes to generate directed-acylic-graph-like (DAG-like) automata that has the same matching semantics as that of the trie-like automaton. This optimization reduces the size of the automaton drastically for many of the signatures, which translates to a large reduction in the memory footprint of the classifier. Figure 5 shows the optimized PTA corresponding to the PTA shown in Figure 3.

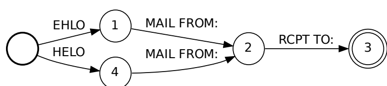

Fig. 5. Optimized PTA for SMTP

3) Assign Confidence Scores: A flow may match multiple state machines in the classifier. We developed a metric, called *confidence score*, that helps us resolve ambiguity and assign a unique label in case of conflicts. Intuitively, confidence scores are values associated with the signatures that represent the confidence that we have about how good a given signature is for accurately identifying the application. If a flow matches multiple state machines, we assign the label of the state machine that has the higher confidence score. Since not all the paths in a state machine are equally good for identifying an application, we assign a confidence score for the state machine and another for each path within the state machine. The *state machine* confidence score is directly proportional to the number of flows considered for signature generation. The intuition here is that we can have a higher confidence on state machines that are extracted from a higher number of flows.

We have developed three confidence scores based on path characteristics which can be used independently or in combination (along with the state machine confidence score). Here we explain these confidence scores in more detail.

Path lengths. Longer path lengths are typically better signatures than shorter path lengths. One of the confidence scores that the distiller module assigns is based on the path length where longer paths get higher scores.

Transition probabilities along a path. If a lot of flows in a flow set matches a particular path in the state machine, then we can consider the path to be a *good* path. To capture this notion we use a confidence score based on transition probabilities. The transition probabilities are computed after the state machines are constructed using the percentage of flows from the flow set that traverse a particular transition. The transition probabilities are weakly decreasing along a path. Hence, we consider the probability of the last transition on a path as the representative (lowest) probability of the path being taken. We assign a high confidence score for paths that have large values of last transition probability.

Term Frequency Inverse Document Frequency (TFIDF). Term Frequency Inverse Document Frequency (tf-idf) [22] is a weight commonly used in information retrieval and text mining to evaluate how important a word is to a document in a collection. The importance of a word increases with its frequency in a document but reduces with an increase in the number of documents containing that word. Intuitively, a high tf-idf word indicates that the word is good for identifying a document in a collection. We use a similar notion in the distiller module to score the term sets that we use for signature generation. A state machine represents a document and the set of all state machines represents the overall collection. We compute the tf-idf value of each term with respect to a state machine. To compute the confidence score of a match along a path, the distiller computes the maximum[2] tf-idf of all terms along a path and assigns it to the accepting state. Using this methodology, we assign a high score to paths which have terms with high tf-idf values, thus helping us to distinguish an application from the entire set of applications.

[2]Note that we can use the average, median or any other metric feasible in this context.

Note that none of the above measures will be very accurate in all scenarios. For example, the signature for BitTorrent protocol, shown in Figure 7, has a path length of 1. If we only use the path length based confidence score, then we will ignore it whenever we have matches with other signatures of longer path lengths. On the other hand if we use $tfidf$ confidence score, then this is a very strong signature. Hence, we wish to point out that an ideal approach is to use all the confidence scores together and make a decision based on all the scores.

IV. Evaluation

In this section, we provide details about the setup that we have used for our experiments and the results that we obtained using the system that we presented.

A. Setup

Data Set. To evaluate our algorithms, we used 2 different traces from large ISPs. The first trace is from a tier-1 backbone network, while the second trace is from a cellular service provider. The first trace is about 10 minutes long collected from a OC-192 link while the second trace is about 30 minutes long collected on a 1 GigE link. Both the traces are raw packets including the complete packet payload. The details of the traces are shown in Table I.

Name	Collection Time	Duration	Link	Flows
ISP1	Aug 2010	10 mins	OC-192	1.2M
ISP2	Mar 2011	30 mins	1 GigE	5.2M

TABLE I
Data Traces Used in Experiments

System Settings. Before we present the state machines, it is important to understand all the parameters that we use in our algorithms when running our experiments. Due to space limitations we will not discuss how we tune these parameters in our experiments. However, Table II shows all the parameters used. Using these parameter settings we extracted 36 state machines for ISP1 traces and 158 state machines for ISP2 traces. Some of the key observations are presented below.

Parameter Name	Value
Min. Flows in a Flow-Set, $nmin_{th}$	50
Min. Num. of Servers, s_{th}	5
Max. Num. of Flows Per Server, $nmax_{sth}$	200
Min. Num. of Total Flows	1000
Max. Num. of Total Flows	5000
Min. Term Length, T_{len}	4 bytes
Term Probability Threshold, P	0.8
Max Num of Term Threshold, N	Unused

TABLE II
Parameter Settings for SANTaClass System

B. Results

1) PTAs Generated by SANTaClass: Here, we will show several different PTAs that we were able to extract automatically from the two traces. We will focus mainly on PTAs that we generated using the ISP2 (the cellular service provider) traces. However, we will use some of the PTAs from the other ISP to compare and contrast between the PTAs.

Some PTAs are trace dependent. Figure 6 shows the state machine that our algorithm automatically extracted for SMTP in ISP2 trace. This PTA has has just one path with three terms (`HELO`, `MAIL FROM:`, `RCPT TO:`). However, the same application has a different PTA in ISP1 traces (Figure 5). We can see that there is an additional path, starting with a transition on `EHLO`. The main take away point here is that some of the applications like SMTP (which typically runs on TCP port 25) can behave a little differently in different networks depending on the implementation both on the server and the client sides. Hence, it is a good choice to extract signatures for such applications on the network where we intend to classify traffic since signatures carried over from one network to another might not work very well.

Signatures for new applications. We examined the set of PTAs that were output by our system and found that there were several PTAs that we could not easily associate with a well known application. We show a couple of examples for this here. The first application is "nginx", a http and a reverse proxy server that is very popular with the users in ISP2. The PTAs (for both the directions) for this service are shown in Figures 8 and 9. Note that some of the terms in the PTA can help us identify the application/service easily. The second application that we want to point out is "LindenLab" (Figure 10). This is a gaming application (on the same lines as "second life"). The main take away point here is that our approach is capable of identifying new applications without any human intervention.

The terms in a state machine can reveal a lot about the application. Another important characteristic of our signature generation process is that the PTA and the term set associated with it can reveal a lot of information about the behavior of the application. For example, consider the RTSP protocol that is running on TCP port 554 in the ISP2 trace. Figures 11 and 12 show the state machines for this protocol in the client to server and server to client direction respectively. These PTAs reveal a number of important aspects: (1) The first term `DESCRIBE` is a request method in the RTSP protocol. (2) The term `YOUTUBE` occurs with a probability of over 90%. This shows that Youtube is an extremely popular site among ISP2 customers and more than 90% of RTSP requests are intended for Youtube streaming. (3) The term `.3GP` reveals the video file format that is being requested. (4) The term `RTSP/1.0` reveals the version of RTSP being used and `CSEQ` that a sequence number is being specified for a request/response. (5) The term `MVOG` tells us that the cellular service provider network is using a mobile video optimization gateway (MVOG) to optimize multimedia traffic.

Fig. 6. PTA for SMTP in ISP2

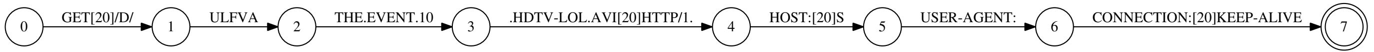

Fig. 8. PTA for <TCP,182,c-to-s> in ISP2

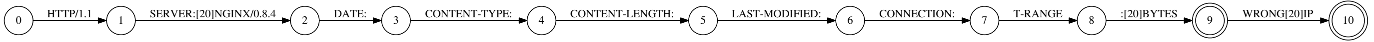

Fig. 9. PTA for <TCP,182,s-to-c> in ISP2

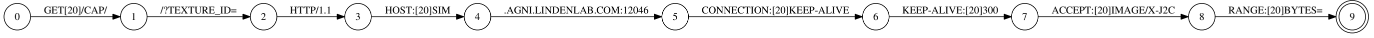

Fig. 10. PTA for <TCP,12046,c-to-s> in ISP2

Fig. 11. PTA for <TCP,554,c-to-s> in ISP2

Fig. 7. PTA for Bittorent

	ISP1	ISP2
Total Flows	1.2M (100%)	5.2M (100%)
Total Classified	1.19M (99.56%)	5.17M (99.54%)
Total Missed	5.3K (0.44%)	23.8K (0.46%)

TABLE III
FINAL CLASSIFICATION RESULTS

C. Classification Results

Measuring coverage. For each of the traces, we passed the complete trace through the system to generate signatures and then passed the trace to the system again to evaluate the classifier recall. Table III shows the classification results for both the traces using the PTAs generated using the same trace. We can see that in both the traces, we are able to classify over 99.5% of all traffic. The small fraction of traffic that our classifier does not identify is mainly due to the fact that we do not extract signatures for applications for which there are not enough flows.

Measuring accuracy. As mentioned before, our system can not tolerate false positives. To measure false positives we need ground truth about the actual application in a flow. One way to obtain the ground truth is to use a traditional DPI system. Unfortunately, DPI systems typically can recognize only a limited number of applications and can not label new applications. Even for known applications, the DPI may label the flow incorrectly. Hence, we have a chicken and egg problem, where it is difficult to obtain the ground truth. However, to get an idea about the false positives in our system, we considered all the flows for which a commercial DPI gave some label. We manually inspected the flows that had a mismatch to check whether the labels generated by our system were incorrect. In our experiments we found no false positives as the mismatches were due to the more fine-grained labeling in our system.

Below we present some interesting findings from our experiments on the trace from ISP2 which highlights the accuracy of our system compared to other approaches.

Well-known applications running on non-standard port. We found 41 SMTP flows on TCP port 110 which is typically reserved for POP3. In addition, we found a HTTP flow on port 110. Traditional port-based classification approach would have classified these flows as POP3.

Tunneled applications. Many applications tunnel traffic inside other applications. Traditional approaches label such tunneled flows with the label from either the outer application or the inner application, but not both. In contrast, our approach presents multiple labels corresponding to both the inner and the outer application. We compared our results with a commercial DPI system. In our experiments, we identified the following tunneled applications based on the labels:

• We found 400 BitTorrent flows within HTTP. The commercial DPI solution labeled these flows as just BitTorrent while our system labeled these flows as both HTTP and BitTorrent.

• For many of the flows labeled as HTTP by DPI system, we had additional labels:

- 3282 flows were labeled as Real Time Message Protocol (RTMP) and HTTP. Inspection of these flows revealed keywords such as HTTP, Shockwave, and Flash, clearly indicating that the flows were carrying RTMP within HTTP.
- We obtained HTTP and LindenLab labels for 5115 flows. A manual inspection revealed that these were LindenLab gaming flows tunneled inside HTTP.
- We identified 30080 flows that were running Torrent inside HTTP. These flows revealed that "Azureus" client was being used for tunneling torrents within HTTP.

Applications using random ports. We see that BitTorrent and other torrent signatures match flows on many different ports. This is expected since clients for these applications do not require a fixed port and end up selecting random port in every session based on user preference.

V. RELATED WORK

Failure of port based traffic classification systems led to a growing interest in deep packet inspection solutions [1], [2], [8], [19]. However, laborious manual step required to reverse engineer protocols to develop signatures make these solutions non-scalable. Many techniques use network traffic

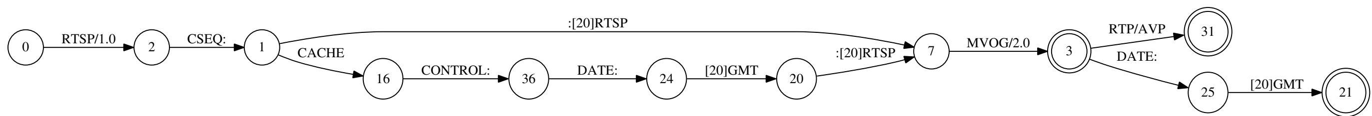

Fig. 12. PTA for <TCP,554,s-to-c> in ISP2

to reverser engineer protocols [9], [20], [21] but their focus is on extracting the message formats and not application identification. They require the flows belonging to an application to be grouped a priori. This limits the use of these techniques in the real-world for generating signatures for new applications.

The LASER [17] and CUTE [24] systems try to automatically extract application signatures based on longest common substrings in application flows. However, these systems have high false-positive and false-negative rates due to the lack of context in signatures. ACAS [11] is another technique intended to automatically extract application signatures using the first 200 bytes of the flow payload content with the intent of classifying traffic [11]. Although this work is novel from a pure conceptual perspective, the practicality of such framework is questionable since it has been tested only on a very few and well-known applications such as FTP, POP3, and IMAP. Moreover, it does not provide a way to recognize new applications; it is mainly intended to find signatures of applications that an operator is aware of. ACAS expects the operator to manually group the flows that belong to an application and provide it as input. Ma et al [14] developed techniques for unsupervised learning for traffic classification using common substrings. However, they do not show the practicality of such techniques in recognizing applications in the wild. Automatic worm detection is another area where researchers have studied signature extraction in an automated fashion [12], [13], [16]. However, these techniques also require the flows, for which signature is to be extracted, to be grouped together a priori.

Another drawback of DPI systems is that it is not always possible or legal to access full payload data. This fact combined with the inability of DPI in handling encrypted traffic led to the development of techniques that use flow statistics (i.e., L4 data) [5], [7], [10], [15], [25], [26]. Some of these techniques have been shown to achieve high accuracy. However, in general, these results are from controlled experiments and do not translate to equivalent high accuracy when dealing with applications in the wild.

VI. Conclusions

In this work we presented SANTaClass, an automated signature generation and traffic classification system based on the Layer-7 (packet content) data. We proposed algorithms for signature generation and distilling the generated signatures, and showed that the generated signatures are practical for real-time classification in the real-world.

References

[1] CloudShield Technologies. http://www.cloudshield.com.

[2] L7 filter. http://l7-filter.sourceforge.net/.

[3] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, June 1975.

[4] I. Bermudez, M. Mellia, M. Munafo, R. Keralapura, and A. Nucci. DNS to the Rescue: Discerning Content and Services in a Tangled Web. In *ACM Internet Measurement Conference*, 2012.

[5] L. Bernaille, R. Teixeira, and K. Salamatian. Early Application Identification. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2006.

[6] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis. In *ACM Conference on Computer and Communications Security*, 2007.

[7] J.Y. Chung, B. Park, Y.J. Won, J. Strassner, and J.W. Hong. Traffic Classification Based on Flow Similarity. In *IEEE Workshop on IP Operations and Management*, 2009.

[8] Allot Communications. http://www.allot.com/.

[9] W. Cui, J. Kannan, and H. Wang. Discoverer: automatic protocol reverse engineering of input formats. In *Usenix Security Symposium*, 2007.

[10] J. Erman, M. Arlitt, and A. Mahanti. Traffic Classification using Clustering Algorithms. In *ACM SIGCOMM Workshop on Mining Network Data*, 2006.

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

[12] H. A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, 2004.

[13] Z. Li, M. Sanghi, Y. Chen, and M. Y. Kao. Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *IEEE Symposium on Security and Privacy*, 2006.

[14] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. Voelker. Unexpected Means of Protocol Inference. In *ACM Internet Measurement Conference*, 2006.

[15] A. Moore and K. Papagiannaki. Toward the Accurate Identification of Network Applications. In *Passive and Active Measurements*, 2005.

[16] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symposium on Security and Privacy*, 2005.

[17] B. Park, Y. J. Won, M. Kim, and J. W. Hong. Towards Automated Application Signature Generation for Traffic Identification. In *IEEE/IFIP Network Operations and Management Symposium*, 2008.

[18] S. Sen, O. Spatscheck, and D. Wang. Accurate, Scalable In-Network Identification of P2P Traffic Using Application Signatures. In *International Conference on World Wide Web*, 2004.

[19] TSTAT. http://www.telematica.polito.it/public/project/tstat-traffic-monitoring-and-classification.

[20] Y. Wang, Y. Xiang, W. Zhou, and S. Yu. Generating regular expression signatures for network traffic classification in trusted network management. *Journal of Network and Computer Applications*, May 2012.

[21] Y. Wang, X. Yun, M. Z. Shafiq, L. Wang, A. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo. A Semantics Aware Approach to Automated Reverse Engineering Unknown Protocols. In *IEEE International Conference on Network Protocols*, 2012.

[22] Wikipedia. http://en.wikipedia.org/wiki/tfidf.

[23] G. Xie, M. Iliofotou, R. Keralapura, M. Faloutsos, and A. Nucci. SubFlow: Towards Practical Flow-Level Traffic Classification. In *IEEE International Conference on Computer Communications (Mini-Conference)*, 2012.

[24] S. Yeganeh, M. Eftekhar, Y. Ganjali, R. Keralapura, and A. Nucci. CUTE: traffic Classification Using TErms. In *IEEE International Conference on Computer Communications and Networking*, 2012.

[25] S. Zander, T. Nguyen, and G. Armitage. Automated Traffic Classification and Application Identification using Machine Learning. In *IEEE Conference on Local Computer Networks*, 2005.

[26] S. Zander, T. Nguyen, and G. Armitage. Self-Learning IP Traffic Classification Based on Statistical Flow Characteristics. In *Passive and Active Measurements*, 2005.