Towards Self Adaptive Network Traffic Classification

Alok Tongaonkar[a], Ruben Torres[a], Marios Iliofotou[a], Ram Keralapura[a], Antonio Nucci[a]

[a] *Narus Inc.*

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, the 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. Furthermore, 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 high precision ($> 97\%$). Finally, our system is easy to deploy and setup and performs classification in real-time.

Keywords: Traffic Classification, Network Monitoring

1. 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. These techniques inspect packet content and try to identify application-level protocols such as Simple Mail Transfer Protocol (SMTP) and RTP Control Protocol (RTCP). 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 the 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, the use of DPI based approaches in large network shows that the coverage or the fraction of traffic that is known, is usually low (this statement will be substantiated with our evaluation using an open source state-of-the-art traffic classification tool). 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 aims 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 [1] or the network traffic belonging to an application [2, 3]. 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. Similarly, RTCP messages contain a version field. The latest

version is 2 but the field may contain the value 1 if the older version is being used. The signature for RTCP needs to account for these variations in the value of the version field. 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 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 novel automated signature generation algorithms with real-time traffic classifiers. Our system can be plugged into any network where it can *automatically* learn 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 when new applications traverse the link.
- We built an efficient system which combines automated signature generation with real-time traffic classification. The set of signatures that are extracted is utilized to classify traffic in the future in a transparent fashion.
- Our experiments with real traffic from multiple ISPs shows that our methodology:
  - increases coverage by identifying new applications. We were able to increase coverage by up to 30% for TCP flows and 23% for UDP flows in one of our datasets.
  - 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 underlying 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 2 we describe the system design. Section 3 describes the complete system implementation. We present the experimental results in Section 4. We discuss related work in Section 5. Finally, we conclude the paper in Section 6.

2. 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.

2.1. 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.

Text-based and binary-based protocols differ in how information is encoded in the fields which necessitates developing different kinds of signatures for them. In text-based protocols, fields contain plain text which is human readable. Typically, the fields can occur at different offsets within application flows and may also have variable lengths. On the other hand, binary-based protocols use fields with values which can be interpreted as binary values by a machine. Typically, the fields occur at fixed offset and usually are of fixed length. Below we describe the signatures that we generate for text-based and binary-based protocols respectively.

2.1.1. Text-based Applications

In this section we discuss the signature for SMTP which is a text-based protocol. 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 [4]. Such an approach is

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

Figure 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

Figure 2: SMTP Session 2

light-weight and depends only on the weighted terms. However this approach introduces false positives [5]. 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 [6]. 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.

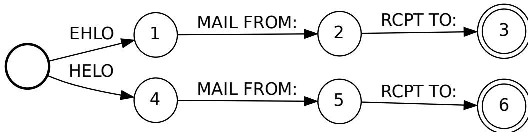

Figure 3: PTA for SMTP c-to-s

2.1.2. Binary-based Applications

In this section we discuss the signature for RTCP which is a binary-based protocol. The intuition for binary-based application signatures is same as the one for the text-based cases, i.e., we can construct signatures from the invariant parts of the protocol flows.

Consider, for instance, the RTP Control Protocol (RTCP), which is used to provide statistics and control information of Real-time Transport Protocol (RTP) flows and runs over UDP. The protocol header, obtained from RFC 3550 [7], is described in Figure 4. We identified the following three types of fields in the header. (i) Low entropy fields (i.e. fields where the value rarely changes): In our datasets, the value of the packet type (PT) field is always 200, which corresponds to the Sender Report (SR) packet type. Similarly, the value of the version (V) field is always two, since this is the latest version of the RTCP protocol. (ii) Medium entropy fields (i.e. fields where the value changes sometimes): In our datasets, the value of the padding (P) field is zero most of the time, because padding is more commonly used in some encryption algorithms. Similarly, the reception count (RC) shows the number of reception reports embedded in the packet, which as we will see later, is typically only one report per packet. Finally, the length field, which represents the length of this packet content (i.e. RTCP header and payload) in 32-bit words, is typically small and most of the 16-bit values assigned to the higher order bits of this field are zero. (iii) High entropy fields (i.e. fields where the value changes very often): These cases include the timestamp and the ID (SSRC) of the sender. Our intuition is that those fields with values that rarely change can serve as the building blocks in our signature.

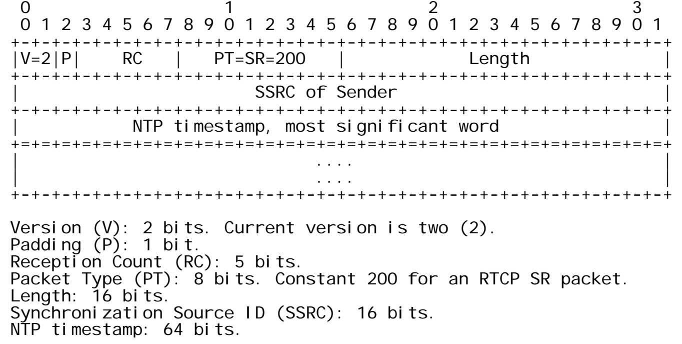

Figure 4: Header of the RTCP protocol

A clear difference between binary-based and text-based applications is that the invariant parts for binary-based cases are at a fixed offset within each flow. Moreover, the invariant parts may be less than 8 bits whereas for the text-based protocols typically they are in multiples of 8 bits. This is because of text-based protocols using encodings like ASCII or UTF-16 which have characters that are multiples of 8 bits. We call the offset within a payload and the value at the offset as *feature*. The problem of generating binary signature then reduces to one of finding the right size for features and then finding features that are common across flows belonging to a protocol. We found empirically that 4 bits is a good feature size that captures invariants across many different binary protocols. The question that we try to address in this work is "*how can we generate application signatures from common features?*".

Figure 5 shows the most commonly occurring values for the first 24 offsets (12 bytes of payload) in the client to server direction of RTCP flows in a large ISP dataset. Note that since we are using 4-bit features, the value of each feature will range from 0 to 15. On the x-axis, we denote each feature as F_i =

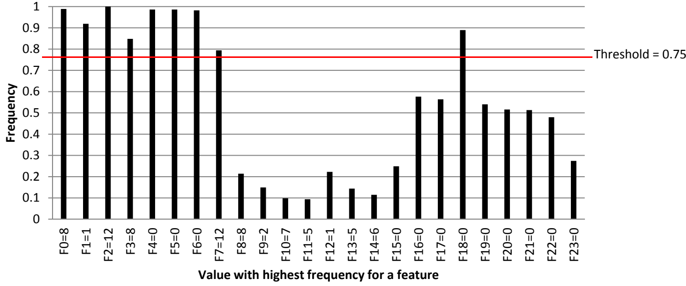

Figure 5: Frequency of features for RTCP c-to-s. In this example, any feature that repeats in more than 75% of the flows, is selected for the final RTCP signature.

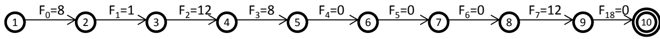

Figure 6: PTA for RTCP c-to-s

val, where i represents the offset which varies from 0 to 23 and val is the value at that offset. We draw several observations. First, we notice that constant values from the protocol header show up as dominant values for the offsets in the 4-bit groups that we analyze. For instance F_0 is 8, or in binary 1000. The first two bits correspond to $V = 2$ (i.e. the expected RTCP version), the next bit to $P = 0$ (i.e. padding is not used) and the last bit to the most significant bit of RC which equals zero (i.e. small number of reception reports in this packet). Thus, $F_0 = 8$ can be used as a feature in our signature for RTCP. We form the binary signatures as vectors of features that occur commonly. In Figure 5, features with frequency over 0.75 are selected for the final RTCP signature. For ease of understanding, we represent the binary signatures as state machines as shown in Figure 6. Note that transitions in this state machines are on features which are combination of offsets and the values at the offsets.

2.1.3. 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. Many applications use certificates which are same for all communication. Since our technique relies on traffic from multiple communicating entities (explained in Section 3.3), these certificates may show up as invariant parts of the communication and can be used in generating application signatures for the application using encryption. Moreover, the invariant bit patterns in encrypted payloads are captured as binary signatures. 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 of real-world encrypted traffic in many cases.

2.2. Design

Figure 7 shows the architecture of the system with blocks having real-time constraint shown in lighter color (Green). 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. Since we have no way of identifying whether a flow belongs to text-based protocol or binary-based protocol, we first use the text-based classifier. If it is not able to label the flow we pass it to the binary classifier. If either of the classifiers successfully labels a flow, then the result is recorded in a database. The classification process for the flow ends. However, if the classifiers 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 both signatures for each flow-set. The signature generator tries to extract both text-based and binary-based signatures for each flowset since it does not know a-priori the type of the protocol. 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 when 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.

3. Implementation

In this section, we describe each of the system components in detail. Table 1 summarizes the list of parameters used by our system and the recommended values based on our evaluation.

3.1. 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

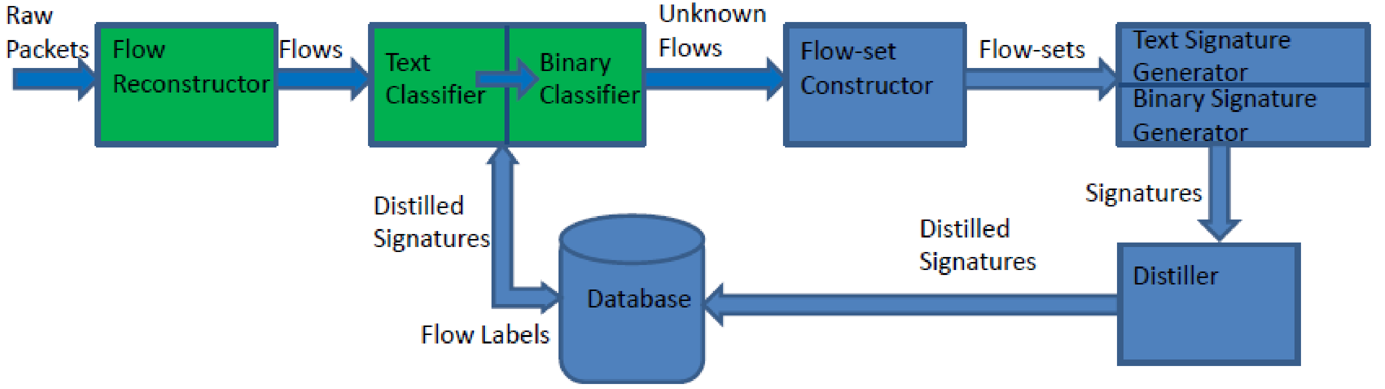

Figure 7: SANTaClass Architecture

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
Number of bytes considered, B	24
Feature size, $fsize$	4 bytes
Minimum Probability of Occurrence, t_p	0.6
Minimum number of Features, f_{min}	5
Minimum number of non-zero values, nz	1

Table 1: Parameter Settings for the SANTaClass System

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 as 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.

3.2. *Classifier*

The classifier is responsible for matching every incoming payload against all signatures in the database and identifying all the matching signatures. The classifier has two different components: (i) text-based classifier (ii) binary-based classifier. Any incoming payload is provided to the text-based classifier and only if it does not match any signature, then it is provided to the binary-based classifier.

3.2.1. *Text-based Classifier*

The classifier can naively match a payload 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 [8] 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 the 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 the 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.

3.2.2. Binary-based Classifier

The binary classifier is quite straightforward. Each path in the state machine representation of the binary signature is treated as a vector of features. For each feature the classifier checks whether it is present in the incoming flow. This involves matching the value at the offset corresponding to the feature in the payload against the value in the feature. If it matches then we consider the feature to match and if all the features match along a path, then the classifier considers the path, and correspondingly the signature to match. The classifier iterates over all the signatures and matches the payload against them. We note here that this matching is restricted to the first 24 bytes and involves simple equality test on 4 bits. Hence, the matching is efficient and does not require sophisticated data structures for matching as in the text-based case.

3.3. 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 fact 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 [9]. 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 a 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 [10]. 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.

3.4. Signature Generator

We have developed a novel system for extracting signatures from the flow-sets. Our system extracts the text-based and the binary-based signatures independently. In this section, we describe both the signature generation algorithms in detail.

3.4.1. Text-based Signature Generator

The text-based signature generator is composed of the following three components: (i) Common Term Set Extraction (ii) Common Term Set Refinement (iii) PTA Generation.

Common Term Set Extraction. The input to this component is a flow-set that contains the payload 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.

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.

• *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.

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.

3.4.2. Binary-based Signature Generator

In this section, we describe our algorithm for extraction of signatures for binary protocols.

Common Feature Extraction. The input to this component is a flow-set that contains the payload of each flow in the flow-set. For binary signatures, we consider only the first B bytes from the start of the payload. Empirically we found that 24 bytes are sufficient to capture application headers. We fix feature size to be $fsize$ bits. If $fsize$ is very small, then fields get broken into multiple features. On the other hand, large values of $fsize$ result in multiple fields being included in a feature. Both cases result in the extraction of features which lead to poor accuracy and recall. Empirically we found a value of 4 allows us to capture many of the binary fields such as version number. We consider $fsize$ bits in the B byte payload for feature extraction. For each offset, we find the most common value at that offset. If the number of flows containing this value is above a certain threshold, say t_f, we consider that feature as a common feature.

PTA Generation from Features. The ordered list of common features forms a path in the PTA. The flows that match the path completely are said to *contribute* to the PTA. All the flows that do not contribute to the PTA are grouped together and the Common Feature Extraction step is applied to this group. The ordered list of new common features forms a new path in the PTA. These two steps are repeated till a minimum threshold percentage of the flows from the flow-set contribute to the PTA. The

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

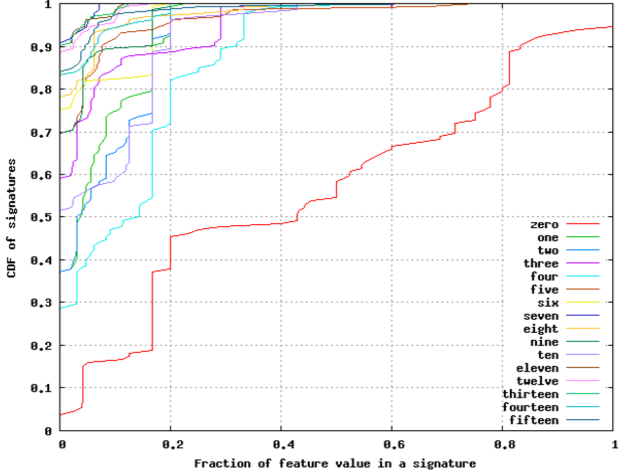

Figure 8: CDF of values that common terms can take in our dataset

reason for repeating these steps becomes clear if we consider the example of version field in a protocol like RTCP which can take a few different values. In the case of RTCP, the version field can take one of two values i.e., 1 or 2. So in the first round we may extract a feature $F_0 = 8$ which corresponds to version being 2. Now assume that 60% of the flows contain this value but 40% contain the value 1. If our signature only contains a path with $F_0 = 8$, we will not be able to match 40% of the flowset in this case. In the second round, since we consider only the flows that did not contribute to the PTA in the previous round, the feature $F_0 = 4$ will show up 100% of the times and hence, be picked up in a second path. Now the PTA can match 100% of the flows for the version field.

PTA Refinement. Finally we refine the PTA as follows: (i) We discard paths that have a probability of occurrence of less than a given threshold (t_p). These paths typically may be due to noise or may belong to the mode of the application that is not very commonly used. Pruning these paths may reduce recall, but precision will not be affected which is consistent with our design goals. (ii) We discard paths that don't have enough deterministic features. We require signatures to have a minimum number of features in them (f_{min}) to avoid weak signatures which can lead to false positives. We require a minimum of 5 features in our experiments. (iii) Zero-valued terms are more likely to occur than other terms for many protocols. Consider Figure 8, where we show, for all signatures the fraction of zero values, as well as the fraction of other values taken by common features. Note that for 50% of our signatures, more than 45% of the feature values in the signature are zero. This clearly shows that zero values dominate the feature value space. Given this fact, considering signatures with large fraction of zero values may lead to an increase in false positives. Therefore, we drop signatures with very few non-zero values (i.e., less than a threshold nz). In our experiments, we typically use $nz = 1$, i.e., at least one feature must have a value different than zero.

The final PTA contains the paths that have not been pruned. For example, Figure 6 shows the PTA generated for RTCP, which we described in Section 2.1. Note that there is only one path, with frequency of 96%. However, there is no second path for the remaining 4% of the flows. This means that there isn't a frequent enough path to cover those flows and our system prefers not to include any other path in the PTA.

3.5. 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:

3.5.1. 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.

3.5.2. 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 9 shows the optimized PTA corresponding to the PTA shown in Figure 3.

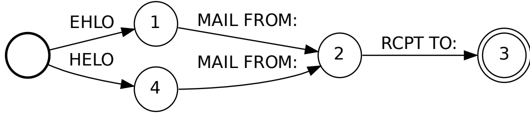

Figure 9: Optimized PTA for SMTP

3.5.3. 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) 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 for text-based protocols. Note that this confidence score is used only for the text-based signatures. 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.

Note that none of the above measures will be very accurate in all scenarios. For example, the signature for BitTorrent protocol, shown in Figure 10, 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.

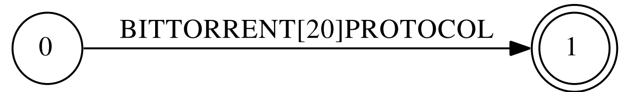

Figure 10: PTA for Bittorent

4. Evaluation

In this section we describe our evaluation results.

4.1. Datasets

In our evaluation we use three traces, which are described in Table 2. Trace-1 and Trace-2 are collected from two large ISPs, with residential ADSL users. The third trace, named Trace-3, has traffic from a mobile service provider, mainly offering 3G/4G services. The traces cover hundreds of thousands of users, and tens of millions of TCP and UDP flows. Overall, the traces cover a diverse set of users, from different countries/continents, using different devices; desktops versus mobile devices. One trace was collected in 2010 and two traces during 2011.

Name	Date	Duration	Total Flows	TCP Flows	UDP Flows
Trace-1	Aug2010	30 mins	3.4M	1.7M	1.7M
Trace-2	Aug2011	1 day	15.2M	11M	4.2M
Trace-3	Mar2011	3 hours	14M	5.7M	8.3M

Table 2: Data Traces Used in Experiments

4.2. Experimental Setup

In all our tests we used identical experimental setup. First, we pass each trace through SANTaClass and automatically extract signatures. We populate the SANTaClass signature database

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

with these signatures and replay the same trace again. We let our system decide which signature(s) match each flow in the trace and store the results. Finally, we classify the same set of traces using Tstat [11], a popular open source traffic classifier. Once we have all the classification results from both classifiers, we compare the results of SANTaClass with the Tstat in an off-line manner.

We face one challenge in comparing results between the two classifiers. The labels from SANTaClass are very different from the labels in Tstat. That is, we need to use the same set of labels as the reference classifier in order to do a fair comparison. To achieve this, we map each SANTaClass signature ID to an application name (i.e., the label from the reference classifier). Next, we describe the approach we use for mapping signatures to application names.

4.2.1. Mapping Signatures to Application Classes

We assign an application name to each signature using the following process. First, we take all the flows that match the SANTaClass signature X_i, referred to as F_{X_i}, and search for those flows in the results table for the reference classifier (Tstat). If all flows in F_{X_i} match the same protocol (say POP3), then we set the label of X_i to be protocol name (POP3 in this case). In the case where the flows in F_{X_i} match more than one protocol, we take the protocol with the largest number of matches. By applying this process to all signatures, we have a mapping $X_i \rightarrow C_i$, where C_i is the protocol/application name in the reference classifier.

4.3. Evaluation Metrics

We use standard classification metrics to evaluate our system. The set of classes in our classification problem is the set of protocols/applications given by Tstat. First, we measure the number of true positives (TP), the number of false positives (FP), and the number of false negatives (FN) for each class. The number of TP of a class C_i is the number of flows that belong to class C_i and are correctly labeled to be of type C_i. Its FP is the number of flows that are not of type C_i and are mistakenly labeled to be of type C_i. Finally, FN is the number of flows of type C_i that were not reported to be of the type C_i. Using the number of TP, FP, and FN we define the *Precision* (P) and *Recall* (R) for each class as follows: $P = TP/(TP + FP)$, $R = TP/(TP + FN)$, respectively.

In addition to precision and recall for each class, we use two metrics that summarize the results for the entire trace: *Coverage* and *Accuracy on Covered Set* (*AoC* for short). Coverage is the ratio of the total number of flows that are given a classification label over the total number of flows in the trace. AoC is the ratio of the correctly labeled flows over all the flows for which we perform classification. A flow is set to be *Unknown* by a classifier if it does not match any of its signatures. Note that a large number of unknown flows will decrease the overall Coverage of the classifier, but it will not affect its AoC.

4.4. Results

The accuracy and coverage of SANTaClass for all three traces are shown in Figure 11. First, we see that the AoC for both TCP and UDP is high, with more than 97.5% of all labels by SANTaClass are found to agree with the labels from Tstat. The coverage is above 82% for all traces with the exception of Trace-3 for TCP which has 64% coverage.

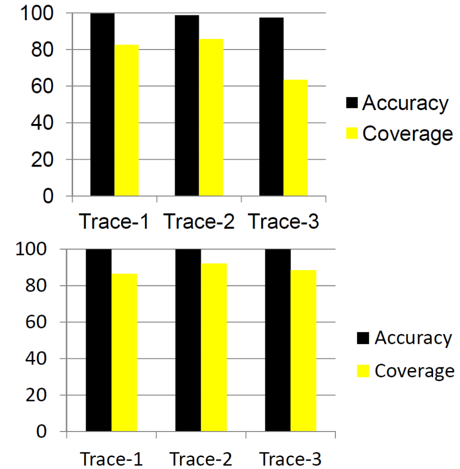

Figure 11: The accuracy and coverage of SANTaClass for TCP (top) and UDP (bottom) traffic.

We make several observations regarding these results. First of all, by further analyzing the data, we notice that for TCP, the protocols that we are not able to identify include various P2P protocols (e.g. BitTorrent and eDonkey), applications tunneled over HTTP and encrypted traffic, the last of which even Tstat is not able to identify. In particular, for Trace-3, encrypted traffic accounts for 83% of all the flows for which we are not able to produce a label. At the same time, we see from Figure 11, the accuracy of Trace-3 is as high as in the other traces, which shows that the all the encrypted traffic only affects the coverage for that trace and does not affect the classification accuracy. Second, for UDP we observe similar patterns as for TCP where the majority of the flows unlabeled by our system are either P2P applications or encrypted traffic. Third, note that our overall coverage is high, considering that we are learning all these protocol signatures automatically, while state-of-the-art traffic classification systems, such as Tstat, have to write the protocol signatures by hand. As a final observation, we want to highlight the very high accuracy we achieve for UDP. In fact, the average UDP accuracy over all three traces is significantly high, i.e., 99.98%. Overall, the classification performance of our system, both in terms of accuracy and coverage, for UDP traffic is very good.

4.4.1. Precision and Recall for Popular Protocols

In Figure 12, we show the precision (left) and the recall (right) for five different protocols. Three of the protocols use TCP and the others use UDP. The detailed per protocol results presented here are consistent with the high accuracy observation from Figure 11. As we see, the precision for all these popular protocols is also very high, i.e., above 0.97. In the bottom plot of Figure 12, we see that the TCP protocol for the BitTorrent application has low recall. This means that some BitTorrent flows running on TCP are not identified by our sys-

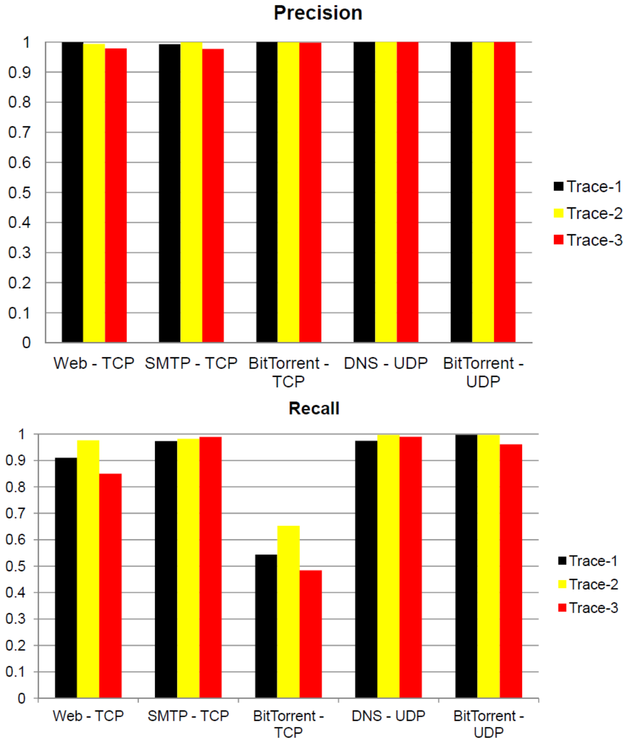

Figure 12: The precision (left) and recall (right) for popular protocols over all three traces.

tem. We attribute this behavior to the multiple communication modes of BitTorrent over TCP that are distributed over multiple ports. Given that our system uses port number to generate flow sets, BitTorrent traffic mixed with other protocols/applications on non-standard ports do not result in the generation of any signature. This explains the lower recall for BitTorrent over TCP. On the other hand, the recall for BitTorrent using UDP is much higher (>0.96), which is also consistent with the results in Figure 11 that highlight the high classification performance of SANTaClass for UDP-based protocols.

4.4.2. Analyzing Unknown Traffic

Figure 13 shows the difference in unknown traffic reported by SANTaClass compared to Tstat. To generate these results we subtract the percentage of unknowns reported by SANTaClass from the unknown traffic reported by Tstat. Therefore, positive values indicate that Tstat reported more unknown flows than our system. Conversely, negative values indicate that our system reported more unknown flows than Tstat. From Figure 13, we see that for UDP traffic, SANTaClass always does better than Tstat (positive values). This is another observation that highlights the high classification performance of our system in identifying UDP-based protocols.

The TCP coverage for Trace-1 and Trace-2 is higher for Tstat, as we see from the negative values in the plot. The main difference comes from P2P protocols, such as BitTorrent and eDonkey that use multiple ephemeral ports that make it harder for SANTaClass to extract signatures from such ports. In addition, the Tstat tool is more aggressive in reporting traffic as

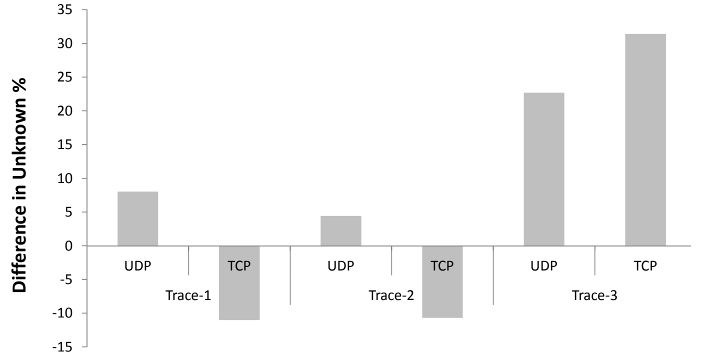

Figure 13: Difference between the unknown traffic reported by Tstat and the unknown traffic reported by SANTaClass. Positive values show that Tstat reported more unknown flows than SANTaClass, and negative values show that our system reported more unknowns than Tstat.

HTTP, whereas the signatures generated by SANTaClass are longer, stricter, and tend to be more conservative. Last but not least, we want to highlight the big difference in coverage between Tstat and SANTaClass for Trace-3. This trace belongs to a mobile provider and has a very different application mixture compared to the other traces. As we see, our system performs significantly better in such unusual environments by learning the intrinsic behavior of traffic. On the other hand, Tstat relies on popular protocols which are present in, primarily, wireline traces and thus fails to identify traffic when the protocols are different. By further analyzing the flows identified by our system but not by Tstat, we observed that approximately 40% of them match previously unknown P2P traffic. In addition, some new applications, such as online games and custom video streaming protocols were also identified. These observations highlight the advantages of SANTaClass over traditional traffic classification tools in identifying new protocols.

4.4.3. Analyzing Generated Signatures

Name	Binary (Known)	Text (Known)	Binary (Unknown)	Text (Unknown)
Trace-1(TCP)	0	17	0	14
Trace-1(UDP)	6	7	12	14
Trace-2(TCP)	0	25	0	35
Trace-2(UDP)	42	6	84	24
Trace-3(TCP)	0	13	0	17
Trace-3(UDP)	11	6	21	30

Table 3: Number of signatures generated per trace for protocols ”Known” and ”Unknown” by Tstat

Table 3 presents the number of signatures that SANTaClass generates for protocols known and unknown by Tstat, both for UDP and TCP flows. We can draw several observations from this table. First, we note that in our experiments, all binary signatures are generated for UDP-based protocols. This signatures significantly help increase the coverage of key protocols, such as DNS and eDonkey. Second, we can observe that even for short traces, such as Trace-1, with duration 30 minutes and 3.4M flows, we are able to generate a total of 70 signatures,

which allows us to achieve the very good coverage and AoC shown in Figure 11. Third, we highlight the fact that we are able to generate many signatures for new protocols which are unknown to Tstat. For all three traces, we generate 117 binary signatures and 134 text signatures for unknown protocols. By investigating these flows further, we identify that approximately 40% of them, match previously unknown P2P traffic.

4.4.4. Interesting Observations

Below we present some interesting findings from our experiments 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 system would have classified these flows as POP3.

Tunneled applications. Many applications tunnel traffic inside other applications. Traditional DPI 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. In our experiments, we identified the following tunneled applications based on the 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.

4.4.5. Discussion

As we have seen before, our system loses coverage on traffic from P2P applications such as BitTorrent and Edonkey, particularly because of the diversity of destination ports used by these applications. This could prevent our system from building a flowset and generating a signature for the P2P protocol. In order to improve our coverage of P2P applications, we can take a few approaches: (i) Collect traffic for a longer period of time, so that we have enough flowsets of a protocol going to a certain port and our system could create a signature out of it; (ii) Make our thresholds to construct flowsets less stringent (refer to Table 1) and generate more signatures for flows directed to ephemeral ports. Along these lines, we could also reduce these thresholds based on a decaying factor proportional to the time a set of flows associated to a port has waited for a label. These new and less strict signatures could have a lower confidence score, which would be communicated to systems consuming the output from SANTaClass. In addition, similar signatures could be merged to avoid replications; (iii) Merge similar flowsets with distinct destination port into a single flowset, so that a signature can be generated. The similarity of flowsets can be evaluated based on the number of common terms between them.

5. Related Work

DPI Signature Extraction. Failure of port based traffic classification systems led to a growing interest in DPI solutions [11, 12, 13]. Manual signature generation for DPI systems is tedious and does not scale well. This resulted in lot of reverse engineering techniques being developed to automatically extract signatures. Techniques such as [4, 5], 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 [14] uses the first 200 bytes from the payload to automatically extract application signature. 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. Ma et al [15] 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 [16, 6, 17]. Other techniques in this category, reverse engineer network traffic from specific protocols to extract message formats [2, 3, 18]. However, all the above techniques also require the flows, for which each per-message signature is to be extracted, to be grouped together a priori. This limits the use of these techniques for generating signatures for new applications. Finally, KISS [19] generates statistical signatures for UDP protocols (mostly binary-based). However, their technique is not practical for flow identification in real world as it requires long lived flows which are rare in the case of UDP.

Statistical Signature Extraction. A drawback of DPI systems is their inability to handle encrypted traffic and cases in which the full payload data cannot be accessed (e.g. because of legal reasons). This led to the development of techniques that use flow statistics (i.e., L4 data) [20, 21, 22, 23, 24, 25]. 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.

6. Conclusions

In this work we presented SANTaClass, an automated signature generation and traffic classification system based on the

packet payload content (i.e., application header and data) of binary and text based applications. We proposed algorithms for distilling the generated signatures, and showed that the distilled signatures are practical for real-time classification in the real-world.

References

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

[2] W. Cui, J. Kannan, H. Wang, Discoverer: Automatic Protocol Reverse Engineering of Input Formats, in: Usenix Security Symposium, 2007.

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

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

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

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

[7] R. 3550, http://www.ietf.org/rfc/rfc3550.txt.

[8] A. Aho, M. Corasick, Efficient string matching: An aid to bibliographic search, Communications of the ACM 18 (6) (1975) 333–340.

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

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

[11] A. Finamore, M. Mellia, M. Meo, M. Munafo, Experiences of Internet traffic monitoring with tstat, IEEE Network 25 (3) (2011) 8–14.

[12] CloudShield Technologies, http://www.cloudshield.com.

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

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

[15] J. Ma, K. Levchenko, C. Kreibich, S. Savage, G. Voelker, Unexpected Means of Protocol Inference, in: ACM Internet Measurement Conference, 2006.

[16] H. A. Kim, B. Karp, Autograph: Toward Automated, Distributed Worm Signature Detection, in: USENIX Security Symposium, 2004.

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

[18] Y. Wang, Y. Xiang, W. Zhou, S. Yu, Generating Regular Expression Signatures for Network Traffic Classification in Trusted Network Management, Journal of Network and Computer Applications 35 (3) (2012) 992 – 1000.

[19] A. Finamore, M. Mellia, M. Meo, D. Rossi, KISS: Stochastic Packet Inspection Classifier for UDP Traffic, IEEE/ACM Transactions on Networking 18 (5) (2010) 1505 – 1515.

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

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

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

[23] J. Erman, M. Arlitt, A. Mahanti, Traffic Classification using Clustering Algorithms, in: ACM SIGCOMM Workshop on Mining Network Data, 2006.

[24] A. Moore, K. Papagiannaki, Toward the Accurate Identification of Network Applications, in: Passive and Active Measurements, 2005.

[25] J. Chung, B. Park, Y. Won, J. Strassner, J. Hong, Traffic Classification Based on Flow Similarity, in: IEEE Workshop on IP Operations and Management, 2009.