# Network Traffic Classification and AppID Creation

By: Avidan Avraham

## Abstract

Identification of applications over IP networks have become a crucial part of network administration. Unidentified applications can result in QoS abuse or even malicious communication. In the past, firewalls relied almost entirely on transportation layer information, such as port numbers, to identify applications by their application protocol. Such an approach is no longer sufficient as application protocols will employ multiple port numbers, and many different protocols. Accurately classifying applications has required reconstruction of application flows. Indeed, next-generation firewalls have become application-aware, identifying applications by relying on protocol structure or other application-layer headers to permit or deny unwanted traffic. Reconstructing application flows, though, is processor-intensive process that does not scale. Many vendors have resorted to manual classification, a labor-intensive process that is costly, lengthy, and limited in accuracy. As such, the list of AppID signatures, the application labels network administrators choose while creating their network and security policies, often categorizes many applications as "unclassified." In this paper, we propose a new approach for automatic, non-HTTP application classification and AppID creation, making it much faster to label unknown traffic and deploy signatures to rule-based AppID engines.

## Introduction

Classifying applications using deep packet inspection (DPI) within network flows has become critically important for a variety of scenarios:

- Next-generation firewall rule definition — Whether for controlling the rightful usage of the enterprise network or to prevent sensitive data from being exfiltrated from the network, firewall rules must accurately identify applications.
- Application-based QoS rule creation — As more and more traditional enterprise applications become cloud-based , the enterprise need prioritize Internet bandwidth use. Music and video services, for example, should not consume line bandwidth when an important voice call needs to occur. Application-based QOS rules are essential to this goal.
- Bot detection and malicious activity prevention — Unknown proprietary applications can be often used by hackers to introduce malware. Similarly, hackers often use known port numbers that are usually allowed to any Internet address to

infiltrate an organization (or exfiltrate data). Detecting malicious bots and other activity requires accurate application identification.

With applications moving to the cloud and the Internet, many are now using HTTP(S) as their protocol. Classification of HTTP(S) traffic can be done largely by identifying the domain address of the application server.

However, numerous applications are not based on HTTP(S). Classifying these applications present a challenge given that most common approach for application identification is by using a set of known destinations defined by variables, such as IP addresses, DNS addresses, port numbers, ASNs (autonomous system numbers), DSCP values or geo-location. However, these methods have a few disadvantages:

- Server IP addresses are most likely to change over time and therefore cannot be used in a static signature.
- Many services publish updateable lists of addresses. Yet these lists cannot be an overall solution since not all providers publish such lists. Even if some do, maintaining such lists is not a generic task for all rule-based AppID engines.
- DSCP values can indicate the packet priority but not its label.
- Many applications do not use fixed, unique port numbers.

A combination of these features is still not enough when classifying WAN traffic, in particular, where still more applications use proprietary protocols and not HTTP. WAN traffic has internal addresses used and therefore cannot be used to sign a specific application.

Manual protocol analysis is possible, but analysts lack good, scalable methodologies for accurately identifying application-protocols. They often resort to information contained in Internet Request for Comments (RFC) though many applications are not defined by RFCs. In most cases, manual classification is a lengthy process requiring full setup of an application environment, packet capturing, protocol analysis and eventually, ending up with signature development.

Another method is statistical flow properties analysis as presented in [1]. To automatically identify applications using a machine learning model, the publication's authors used attributes such as packet inter-arrival time, packet length mean and variance, flow size (bytes) and duration. Such an approach, though, requires looking deep into the packet flow to identify the application.

We propose an approach that automatically classifies protocol-based applications based on information in the first packet of the flow using a machine learning model. In this way, we

can reduce the time to classify new applications and classify existing applications more accurately than when done manually.

## High-level overview

In our proposed approach, we record various types of data per each network flow:

- First 32-64 bytes of the first packet data, we name it *'first_data'*
- Destination port number
- Destination domain name (when available)
- Destination IP address
- Destination ASN - Autonomous system number (augmented via an external ASN DB)

We use a metric to compute the distance between different values of *first_data*. The distances help us to create groups of close values and along with a clustering algorithm we create numerous clusters representing various applications distributed over the network. Later, we compute a list of longest common substring across the entire cluster values set, we name it '*LCSS'* (longest common substring). The *'LCSS'* value represents an application signature that can be used for:

- A *'first_data'* search term for features aggregation over the relevant recorded flows (to be described further in the "Labeling" section).
- An AppID signature for DPI-based rule engines.

The relevant clustered content is unlabeled. The extracted attributes from the matching recorded flows are stored and will be used for evaluation and application labeling (see Figure 1).

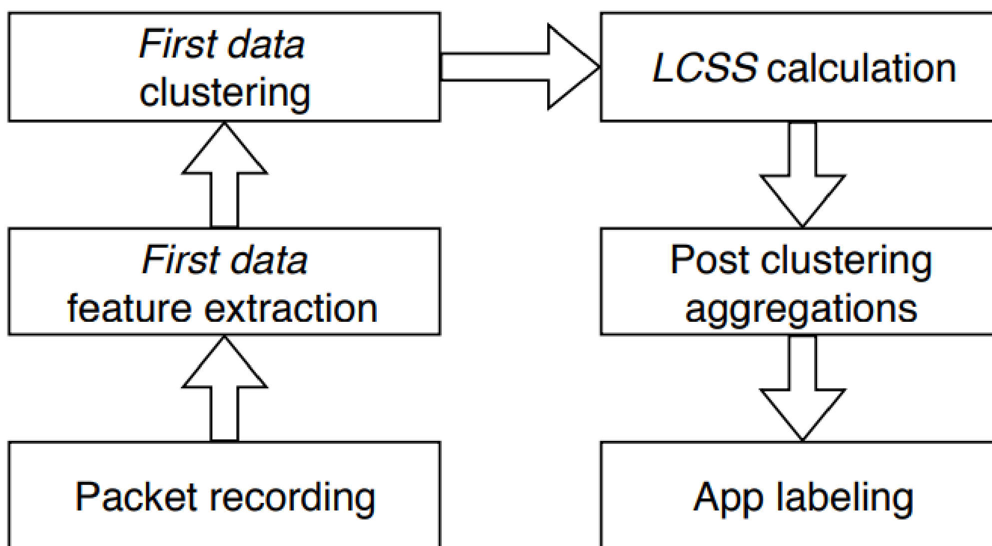

*Figure 1: The approach uses a six-step process to identify application*

Our approach utilizes the actual flow content as the very fundamental feature for traffic classification. We see that relying on this feature yields optimal results as if an analyst classifies network flows manually using traffic analysis tools such Wireshark. Depending on the application set, an analyst can classify on average 10 applications in an hour. Our six-step process managed to create more than 100 application signatures in less than an hour.

## Clustering algorithm

To create clusters of the 'first_data' flow attribute, we created a parallel map-reduce implementation of a Density-based spatial clustering of applications with noise (DBSCAN) algorithm. DBSCAN is a good fit for our problem since we have no prior estimation of the number of output clusters to be found. Also, different executions of the algorithm of different datasets may end up with a different number of clusters.
DBSCAN algorithms require a few parameters - distance metric, eps and min-pts (see Figure 2):

- Distance metric — We use Levenshtein distance, the number of insertions and deletions needed to transform one word into another.
- eps(ε): Defines the maximum distance between two points to be considered as part of the same cluster. Defining eps is a matter of trial-and-error on a small dataset.
- Min-pts - Defines the minimum number of points in a cluster.

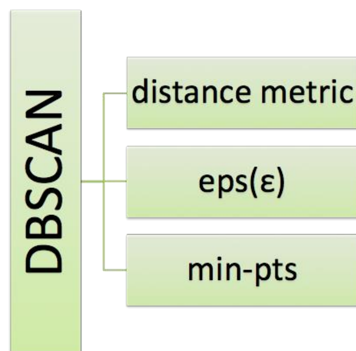


*Figure 2: The DBSCAN algorithm requires at least three parameters*

While executing DBSCAN, we create in memory an NxN matrix with the result of the calculated metric between each and every point in our space. Working with large datasets, such as network traffic flows of large enterprise networks, requires reducing the number of distances calculated to complete execution in time (see Figure 3). For example, after several hours, a DBSCAN execution of a 100,000 sample dataset on 4-CPU/4GB-RAM machine had not completed.

Our approach reduces overall processing time by executing DBSCAN in parallel on smaller divided datasets.  For the full result, we merge the resulting clusters with clusters from parallel executions.

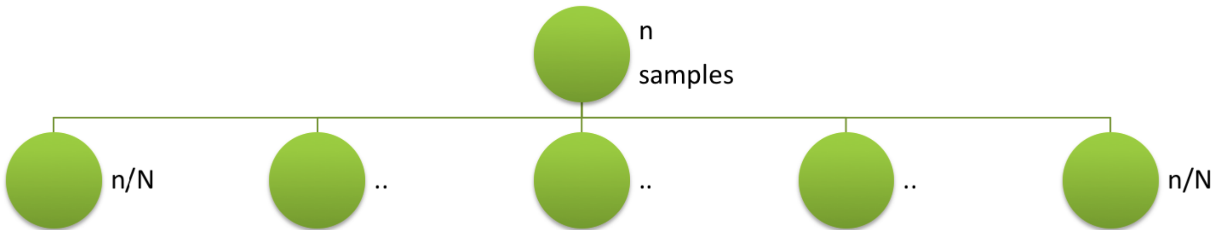


*Figure 3: To reduce execution time, we divide our dataset into "n" number of a total ("N") partitions*

Assuming each DBSCAN execution creates X clusters. Clusters group of n/N might consist of clusters which exist in another clusters group. As a result, a merge function between clusters of different clusters groups is required.

A technique for a merging function solution was proposed by [2]. In this publication, the authors propose taking advantage of DBSCAN's differentiation between a cluster's boundary points and core points.

There are three types of points described in DBSCAN:

- A core point is a point where its closest neighbors are within the cluster.
- A boundary point is a point where one of its closest neighbors are outside of the cluster.
- A noise point is a point which does not belong to any cluster. (See Figure 4)

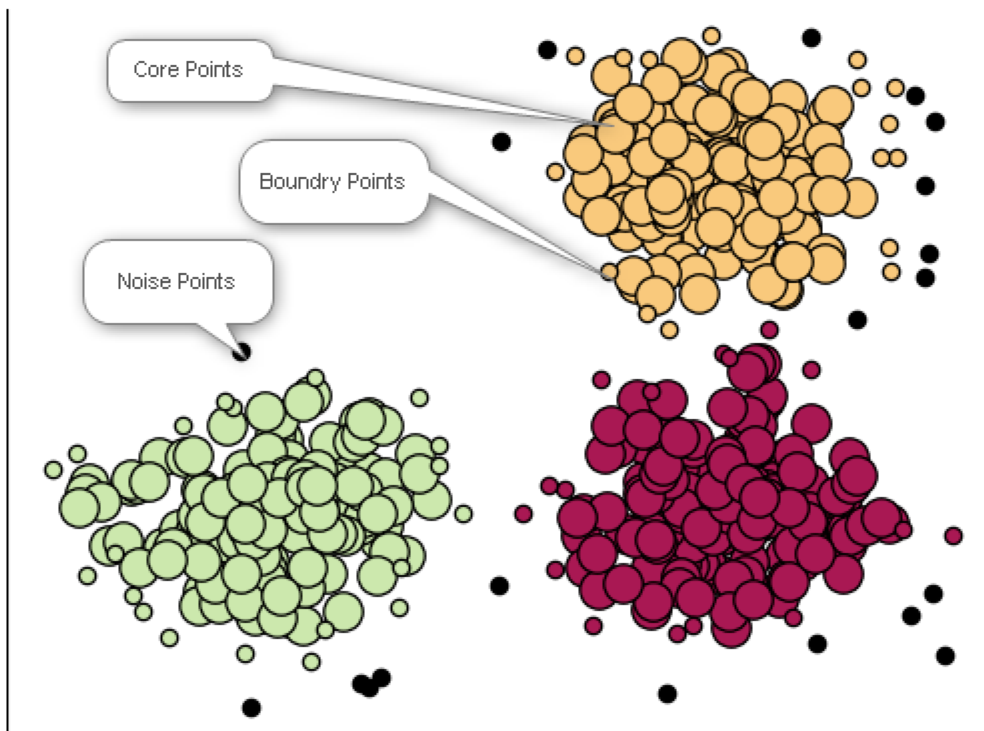


*Figure 4: The following bubble graph illustrates three different applications from a DBSCAN execution represented as clusters of bubbles. Core points are the wide colored bubbles. Boundary points are the small colored bubbles. The black colored points are noise.*

The authors of [2] aggregate boundary points of each cluster as a baseline for another DBSCAN execution. The result of this execution is used to merge the origin clusters of each boundary point. Inspired by this concept, we defined our merging method, with a major change: Our DBSCAN execution didn't yield enough boundary points for each cluster. If clusters lack boundary points there will be no way to merge them.

We wanted to find an alternative approach and add items to the boundary point baseline. We extracted an additional 10% of core points and used them along with boundary points. Afterwards, we executed DBSCAN one more time on the entire set of collected points (see Figure 5).

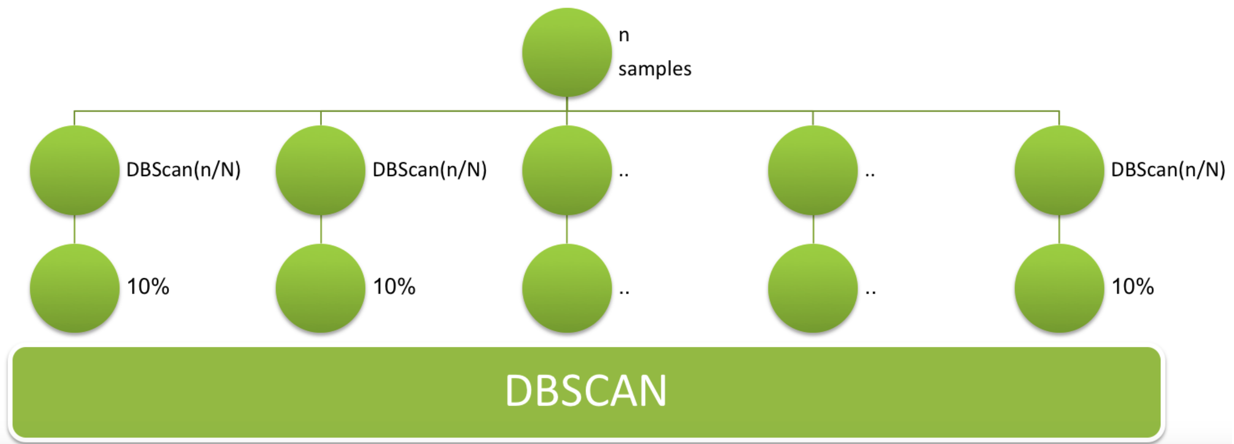


*Figure 5: DBSCAN execution on additional 10% of a cluster's core points allowed us to merge the entire cluster set.*

With the final clustering result, we managed to merge the entire cluster set. Based on the merging principal presented in [2], within the final DBSCAN result cluster, if two points p1 and p2 originated from different clusters c1 and c2 now reside in the same cluster d8, then c1=c2=d8 and we can merge the different clusters content (see Figure 6).

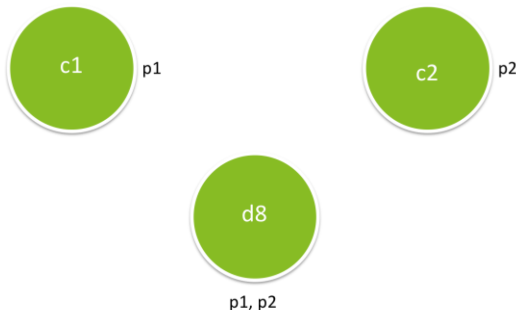


*Figure 6 : Merging clusters from partitioned DBSCAN executions*

## Execution

By executing DBSCAN in parallel, we were able to process the same amount of samples in 10 minutes (4 CPUs, 4 GB RAM), which was not feasible in a non-parallel manner. Recursive execution of this solution allows us to cluster large amounts of data (Figure 7).

```
Algorithm: merge dbscan clusters

function dbscan_reduce(clusters_groups)
     merge_collection <- array()
     for cluster_group in clusters_group do
          for cluster in cluster_group do
               BP <- cluster.fetch_boundary_points()
               CP <- cluster.fetch_10_precent_core()
               merge_collection <- merge_collection ∪ BP ∪ CP
     final_dbscan_results = DBSCAN(merge_collection)
     merge <- array()
     for cluster in final_dbscan_results do
          origins <- array()
          for point in cluster do
               if point.origin not in origins do
                    origins <- origins ∪ point.origin
          merge <- merge ∪origins
     for cluster_list in merge do
          merge_clusters(cluster_list)
```

*Figure 7: Pseudo code of our proposed merging algorithm*

## LCSS Calculation

Longest common substrings (*'LCSS'*) is a list of substrings that exist in every *'first_data'* sample in a cluster. Here is an example of *'LCSS' (highlighted in purple and green)*:
["050000040400031353532c", "0500000312100006a58f7ab647637cdf77975367"]
for the *'first_data'* cluster sample below:

- d1050000040400031353532c60500000312100006a58f7ab647637cdf77975367
- d2050000040400031353532c70500000312100006a58f7ab647637cdf77975367
- ce050000040400031353532c30500000312100006a58f7ab647637cdf77975367
- d050000040400031353532c50500000312100006a58f7ab647637cdf77975367
- cf050000040400031353532c40500000312100006a58f7ab647637cdf77975367
- d3050000040400031353532c80500000312100006a58f7ab647637cdf77975367

*'LCSS'* calculation requires iteration of every sample in a cluster, dividing the samples into smaller sub-strings and validating whether a sample is a substring of all other samples. Using this list for identification of network flows gives us a valid content-based signature for an application (*'AppID'*).

This part significantly reduces the amount of work performed in order to create applications or protocols signatures. Usually, this kind of work requires manual observation of data samples. Since *'LCSS'* is created based on an entire cluster samples, thousands or more samples can be taken into account automatically. As a result, the

automatic protocol analysis supports a variety of use-cases of the application using cloud data rather than manual generation and observation of this traffic.

## Labeling

Labeling means to give a cluster a meaningful classification or tag, which can be commonly used by a human. Such labels allow network administrators, security personnel, and IT managers to create related network or security rules. The labeling process can be done manually, but below we propose an automatic method that will work in some cases. In order to correctly label a cluster, more features associated with the *'LCSS'* search term need to be extracted from our dataset, such as Port numbers, Domain Names, ASN Names, and Common ASCII strings.

We aggregate the above features for every 'LCSS' value. If a single value is found with a feature, for example an 'LCSS' value was seen with only a specific domain name, the feature becomes the label. If no features are found, a randomly generated name will be given to the 'LCSS' value and it will no longer be an unlabeled traffic.

To demonstrate the idea, see the two examples below:

1. The following byte streams of packets were clustered together the common LCSS value of '17240a2000' (Figure 8). Further inspection reveals this cluster data is sent only to a single domain: 'teamviewer.com'. This cluster actually represents the 'teamviewer' application. The LCSS value can be used as a signature to classify this application.

| | |
|---|---|
| 17240a200037793b95f23c80004800800001000000148000004fb380806efdf3 | teamviewer.com |
| 17240a20000a0a98048813800057808000010000000148000004fb380806e7df7 | teamviewer.com |
| 17240a2000380abe9a0813800057808000010000000148000004fb380806e7df7 | teamviewer.com |
| 17240a2000116f44a088138000578080000100000000148000004fb380806efdf7 | teamviewer.com |
| 17240a20006972c18d88138000578080000100000000148000004fb380806ebdf7 | teamviewer.com |

*Figure 8: By identifying a common LCSS value, we're able to identify flows related to the Teamviewer application.*

2. The following bytes streams of packets were clustered into a single group with the common LCSS value:'114163655374726561 6d50726f746f636f6c0000000000000000' (Figure 9). ASCII String representation of the byte streams reveals this traffic represents the AceStream protocol, a peer-to-peer, BitTorrent protocol used for streaming multimedia. The LCSS value is used for its AppID signature.

114163655374726561 6d50726f746f636f6c0000000000000000e848f88c802e .AceStreamProtocol.........H....

## Summary

We presented an unsupervised approach that can efficiently run on large amounts of data to cluster network flows into similar application groups. Later, we found an optimized way to create a parallel map-reduce implementation for DBSCAN. Using this implementation, we demonstrated a technique to classify those apps using *'LCSS'* values. Finally, we show examples of how analysts can leverage this method for application identification. This approach provides an efficient method for application signatures creation in parallel. Applying this method can reduce the costs of AppID creation for identifying protocol-based applications by reducing the amount of human involvement in the process.

**References**

[1] - Sebastian Zander, Thuy Nguyen, Grenville Armitage, "Automated Traffic Classification and Application Identification using Machine Learning" (http://researchrepository.murdoch.edu.au/id/eprint/35006/1/automated%20traffic.pdf) {Return}

[2] - Fang Huang, Qiang Zhu, Ji Zhou, Jian Tao, Xiaocheng Zhou, Du Jin, Xicheng Tan and Lizhe Wang, "Research on the Parallelization of the DBSCAN Clustering Algorithm for Spatial Data Mining Based on the Spark Platform" (https://www.mdpi.com/2072-4292/9/12/1301/htm)
{Return}