Integrating short history for improving clustering based network traffic anomaly detection

Juliette Dromard, Philippe Owezarski

Integrating short history for improving clustering based network traffic anomaly detection

Juliette Dromard, Philippe Owezarski
LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France

Index Terms—**Communication networks, anomaly detection, clustering, traffic history, ground truth, Spark Streaming**

***Abstract*—Traffic anomaly detection is of premier importance for network administrators as anomalies have a dramatic impact on network performances, and QoS perceived by users. It is, however, a very time consuming and costly task that often requires decision from network and security experts. For making anomaly detection autonomous, many research works started investigating the use of unsupervised machine learning techniques, and in most cases traffic clustering. Identifying the clusters corresponding to anomalous traffic classes among the full set of detected clusters still remains a challenge. This is mostly due to the nature of clustering techniques that work on traffic samples of a given duration, each cluster being classified after an uncertain post processing stage. In this paper, we show how anomaly detectors can benefit from keeping a temporal track of the clustering results along time. This improvement has been added to ORUNADA (Online Real-time Unsupervised Network Anomaly detection Algorithm) that aimed at providing efficient anomaly detection on high speed networks. This new ORUNADA version - called H-ORUNADA for History-ORUNADA - is then evaluated on a new ground truth, called SynthONTS, that is currently designed to provide a modern and complete dataset with labeled anomaly. H-ORUNADA has also been implemented on Spark Streaming for being able to work on very high speed networks (targeting several hundreds of Gbits/s), and evaluated on the Google Cloud Platform.**

I. Introduction

With the booming in the number of network attacks, the problem of network anomaly detection has received increasing attention over the last decades. Existing solutions are mainly knowledge-based and this knowledge must be continuously updated to protect the network. However building signatures or new normal profiles to feed these detectors takes time and money. As a result, current detectors often leave the network unprotected for long periods of time.

To overcome these issues, a new generation of detectors has emerged which takes benefit of intelligent techniques which automatically learns from data and allows bypassing the strenuous human input: unsupervised network anomaly detectors [14]. These detectors aim at detecting network anomalies in an unsupervised way, i.e. without any previous knowledge on the anomalies. One of such solutions leverages Principal Components Analysis (PCA) and has been widely studied [8], [13]. However, the majority of unsupervised techniques relies on clustering techniques [12], [2] as K-mean [11], SVM [7] or DBSCAN [15] to quote a few. The related improvement in term of quality of detection is significant, as these clustering based anomaly detectors can now detect 0day attacks. These attacks may appear as outliers or new clusters in the clustered traffic, whereas legitimate classes of traffic appear as temporally regular clusters.

It is then obvious that temporality of clustered information is of high importance. But the existing detectors do not consider temporal information. They only consider the information gathered at a time slot to decide whether it is normal or not. It may be important to consider the evolution of the data and keep track of the history of this information.

In this paper, we then show how anomaly detectors can benefit from keeping a temporal track of the clustering results along time. This improvement has been added to ORUNADA (Online Real-time Unsupervised Network Anomaly detection Algorithm) [6], leading to H-ORUNADA (History-ORUNADA). For reducing significantly the time to detect anomalies in the traffic, and improving the capacity in working with large throughput networks (working in real-time on flowing traffic in current networks is a big data challenge), H-ORUNADA has been implemented on the Spark Streaming big data platform. H-ORUNADA has then been evaluated on a new ground truth, called SynthONTS, that is currently designed to provide a modern and complete dataset with labeled anomaly for efficient evaluation of the detection accuracy of anomaly or intrusion detection tools, as well as on the Google Cloud Platform for evaluating its ability to handle the traffic of high speed networks.

The rest of this paper is as follows: section II details the characteristics of anomalies on long periods of time, and then details the problematic for classifying them. Sections III and IV present the preprocessing stage for building the research space, taking into account long term tracking of the traffic, and then how the clustering is done for efficiently achieving accurate results. Section V presents the evaluation results on the detection accuracy. Then section VI presents some elements about the implementation of H-ORUNADA with Spark Streaming, and presents its performance evaluation on Google Cloud Platform. Finally, section VII concludes the paper.

II. Characteristics of anomalies

An anomaly is usually defined as a flow which is different from the other flows. However, this definition is quite vague. Therefore, in the following, we characterize more precisely what we consider as an anomaly.

First, we make some assumptions about the nature of the anomalies, and how these anomalies should appear using

clustering techniques. First, let's introduce some concepts of clustering techniques applied to network anomalies. Usually, the data to partition is represented by a matrix where each line represents a flow and each column a statistic. This matrix is called the space or feature space. Each flow represents a point of the space and the coordinates of the point are the statistics of the flow. The set of points is the space. A clustering algorithm applied on a space (the data matrix) outputs a partition of the feature space. It identifies clusters (group of points which are close to each other according to a given distance function) and outliers. An outlier is a point which is isolated.

In the following we assume, that any network administrator wants to be aware of rare events going on in the network. This event should be rare not only at a given time, but also considering the traffic history. We define a rare event as either:

- A flow that is now different from the others but that was not different in the past. If such a flow appears rarely but always with the same characteristics (i.e. the flow set of statistic stays the same over time), it however may not be of interest for the network administrator. It may be just a flow induced by a particular server on the network, like the flow induced by the google DNS server for instance. Such a flow may however appear as a new outlier in a space, but is has not to be considered as anomalous, at the opposite of what is generally done, as in [3].
- A flow different from the others whose statistics suddenly change. Two cases have to be considered: a flow which is always different from the other may not be considered as an anomaly, as it can be related as a new kind of application. However, If its statistics change suddenly, it means that something special like an attack or a failure is happening (on a server for instance). Therefore, such a flow needs to be considered as an anomaly. Such a flow may appear as an outlier shifting suddenly in the space.
- A set of flows which appear or disappear suddenly. This set of flows can be, for example, induced by a DDoS attack. For example, when a server is under a DDoS attack the number of flows targeting this network may increase in a significant way. Therefore, the cluster representing the set of flows targeting this network may increase drastically. Such an anomaly may appear as a change (increase or decrease) in the size of a cluster or as the disappearance or appearance of a new cluster.

By using clustering techniques, these rare and interesting events can be defined in a formal way. To define them, three new parameters (in addition to the clustering algorithm parameters) are considered:

- T_{his}: This parameter represents the length of the historic in seconds that is considered. A cluster or an outlier is considered as new in a space if it was not present in this space during the T_{his} last seconds. This value should not be too long in order to adapt to the long term traffic changes.
- N_{clust}: This parameter is a threshold. When the number of points of a cluster changes (it increases or decreases of at least N_{clust} points) the cluster (and all the flows that it contains) can be considered as an anomaly.

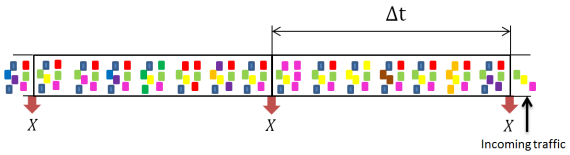

Figure 1. Feature space computation at the end of each time-slot of Δt seconds

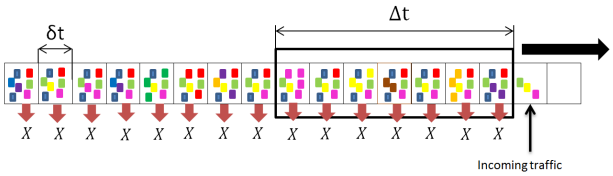

Figure 2. Feature space computation at the end of each micro-time-slot of δt seconds

- d: This parameter is a threshold, if a point (flow) moves of a distance of at least d during the T_{his} last seconds. We consider that the flow statistics changed.

We define formally an anomaly as either:

1) Every flow f detected as an outlier in the partition of a subspace and which has never been detected as an outlier during the T_{his} last seconds.
2) Every flow f detected as an outlier in the partition of a subspace and which slides of at least a distance d during the T_{his} last seconds.
3) Every cluster C which did not exist during the T_{his} last seconds.
4) Every cluster C whose size changes of at least N_{clust} points during the T_{his} last seconds.

III. Preprocessing : construction of the features space

A. The discrete time sliding window

Before applying any unsupervised network anomaly detector, the network traces must be collected on the network link in time slots. Time-slots have to be large enough (of length Δt) in order to gather enough packets to catch flows patterns. Evaluations presented in [10] showed that time-slots of 15 seconds give good results in terms of detection performance: True Positive Rate (TPR) and False Positive Rate (FPR).

Collected traffic is then aggregated into flows using N different aggregation levels. In the following, we decide to use 7 different aggregation levels that will be described later. Therefore, our solution outputs 7 different data matrices $X^1, X^2, X^3, ..., X^7$.

Every flow is described by a set of features (these features are different according to the aggregation level used to generate the flow) stored in a vector. All the vectors generated with the same aggregation level are then concatenated in a normalized matrix X_i, where $i \in [1; 7]$ is the aggregation level. The network anomaly detector processes independently every data matrix. The process of consecutive time-slots is illustrated in Figure 1.

However, using such a large time-slot size introduces an equivalent delay before detecting anomalies, thus forbidding any real-time detection. To avoid that attacks damage the network, anomalies have to be rapidly detected. To speed up the anomaly detection, we propose to update the N feature space and launch the detection in a near continuous way, i.e. every micro-slot of length δt seconds. Then, any sufficiently fast and efficient detector can benefit from the proposed solution to reach continuous and real-time detection. However, if the feature space is computed with only the network traffic contained in a micro-slot, it may not contain enough information for the detectors to identify flows patterns and thus anomalies. To solve this issue, we use a discrete time-sliding window of length Δt. The time window slides every micro-slot of length δt. When it slides, the feature space is updated. The feature space is the summary of the network traffic collected during the current time-window (figure 2).

A discrete time-sliding window is made up of M micro-slots with $M = \Delta t/\delta t$. To speed-up the computation of a feature space X, the sliding window associates to each of its M micro-slots a micro-feature space mX. Each micro-feature space is computed with the packets contained in its micro-slot.

For every aggregation level, the current window stores the M micro-feature spaces in a FIFO queue $Q = (mX_1, mX_2, ..., mX_M)$. mX_M denotes the micro-feature space computed for one aggregation level with the packets contained in the newest micro-slot and mX_1 in the oldest. For a given aggregation level, when the window slides a new feature space denoted X_{new} can be computed as follows:

$$X_{new} = X_{old} + mX_{new} - mX_1 \quad (1)$$

where X_{old} is the previous feature space and mX_{new} the new micro-feature space. Finally, the FIFO queue is updated (mX_{new} is added to the FIFO queue and mX_1 is removed). To benefit from these feature space updates, we devise a detector algorithm capable of detecting continuously anomalies. To reach this goal this detector is based on an incremental grid clustering step which has a low complexity.

B. Description of the aggregation levels and their associated features

Incoming packets are collected in consecutive time bins Δt and aggregated into flows according to different aggregation levels. An aggregation level can be described by a filter and a flow aggregation key. Incoming packets are first filtered and then grouped into flows according to a flow aggregation key. A flow aggregation key specifies a set of fields to inspect in a packet. Packets with similar values for these fields are aggregated into flows. Each flow is then described by a set of attributes or features. Anomalies identified by a detector may be different according to the aggregation level used. Therefore, we apply different aggregation level to the incoming traffic. For every aggregation level i at the end of every time bin, it outputs a set of flows forming a feature space X^i. We use seven different aggregation levels. Every aggregation level is described in Table I. This table displays for every level its filter, its flow aggregation key and the features used to describe a flow. To compute a feature space, they consider every packet of flows in the current time slot (window). Some features are based on the entropy, as previous studies showed that the distribution of some traffic features may reveal anomalies [8]. For example, for the aggregation level at the IP source, we compute the entropy of the source and destination ports and the entropy of the IP destinations. A high entropy of the IP destinations and a low entropy of the source ports imply that the distribution of the source ports is very sparse while the distribution of the IP destinations is very dense and this may reveal a port scan.

We also use the TCP socket pair and the UDP socket pair as aggregation levels. A socket pair is a unique 4-tuple consisting of source and destination IP addresses and port numbers.

Packets are collected on a large network link in consecutive time slots. For every aggregation level except for the aggregation level 7, our solution computes a large set of flows at every time slot. This set is assumed to be large as our solution is applied on the traffic captured on a large network link. However, for the aggregation level 7, only one flow is computed at each time slot. This unique flow summarizes the behavior of the entire link. Therefore, our solution is slightly different when it processes flows computed at the aggregation level 1, 2, 3, 4, 5, 6 and flows generated with the aggregation level 7.

For the aggregation level 1, 2, 3, 4, 5, 6, flows computed during a time slot are directly partitioned using the solution presented thereafter. However, for the aggregation level 7, a certain number of time-slots must pass to collect enough flows (one per time slot) to partition them. Once a set of N (with N large) flows are collected they can be partitioned.

C. Feature space normalization

In the following, data for each aggregation level is represented by a matrix X of size F*D where each row represents a point (or flow in our case) $x = (x_1, x_2, ..., x_D)$ and each column a feature (or dimension). To apply data mining techniques, data features must be comparable and therefore have the same common domain. Data normalization refers to the creation of shifted and scaled versions of every feature. It allows mapping the features values via a transformation function in a common domain. After normalization, features values can be compared. As explained in [1], normalization may be sensitive to outliers and should be removed for the normalization process. To overcome this issue, we propose a robust normalization method. It processes each feature independently and assures that most of the values are in the range [0,1]. The normalization of a feature takes place in two steps. First, it selects data situated between the α and the $1-\alpha$ percentile and remove the lowest and the largest values and therefore, potential outliers. It then computes the feature max and min value.

In a second step, it applies the max/min normalization, using the max and the min value computed during the first step. Therefore, for a point represented by a vector x, its normalized vector is denoted x_{norm}. It can be computed as follows:

$$x_{norm} = \frac{x - perc_{min}(X)}{perc_{max}(X)} \quad (2)$$

Table I
DESCRIPTION OF THE DIFFERENT AGGREGATION LEVELS AND ASSOCIATED FEATURES.

Aggregation level	Filter	Aggregation key	Features	Number of features
1	TCP packets	TCP socket pair	nbPacketsIP1, nbPacketsIP2, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytesIP1, bytesIP2, land, nbChristmasTree, nbMoreFrag	14
2	UDP packets	UDP socket pair	nbPacketsIP1, nbPacketsIP2, bytesIP1, bytesIP1, land, nbMoreFrag	6
3	ICMP packets	Pair of IP addresses	nbPacketsIP1, nbPacketsIP2, bytesIP1, bytesIP2, land, nbReply, nbEcho, nbOther, nbRedirect, nbUnreach, nbTimeExceeded, nbMoreFrag	12
4	no	Pair of IP addresses	nbPacketsIP1, nbPacketsIP2, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytesIP1, bytesIP2, land, nbChristmasTree, nbTimeExceeded, nbUnreach, nbEcho, nbRedirect, nbReply, entPortIP1, entPortIP2, nbICMPOther, nbMoreFrag, nbPacketsTCP, nbPacketsUDP	24
5	no	IP source	nbPackets, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytes, nbland, nbChristmasTree, nbTimeExceeded, nbUnreach, nbEcho, nbRedirect, nbReply, entPortSrc, entPortDst, nbICMPOther, nbMoreFrag, nbPacketsTCP, nbPacketsUDP, entIPSrc, simIPSrc,	24
6	no	IP destination	nbPackets, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytes, nbland, nbChristmasTree, nbTimeExceeded, nbUnreach, nbEcho, nbRedirect, nbReply, entPortSrc, entPortDst, nbICMPOther, nbMoreFrag, nbPacketsTCP, nbPacketsUDP, entIPSrc, simIPSrc	24
7	no	no	nbPacketsIP1, nbPacketsIP2, nbSyn, nbAck, nbCwr, nbUrg, nbPush, nbRst, nbFin, bytesIP1, bytesIP2, land, nbChristmasTree, nbTimeExceeded, nbUnreach, nbEcho, nbRedirect, nbReply,entPortIP1, entPortIP2, nbICMPOther, nbMoreFrag, nbPacketsTCP, nbPacketsUDP	24

with $perc_{min}(X)$ and $perc_{max}(X)$ are respectively a vector made up of the min and max value of every feature computed during the first step. The max and min values for every feature are stored in order to re-use them to normalize the data obtained in future slots.

However, data may evolve in time. Therefore they should be recomputed to adapt to network changes. These values should be recomputed when an important percentage of the normal data do not lie any longer under the max and min value of the feature. We make the assumption that an anomaly is a temporal event and should then not last longer than n slots. Therefore, the max and min values of a feature should be recomputed if the percentage of data not lying between the α and the $1-\alpha$ percentile for at least n slots is superior to a certain threshold in order to ensure that this shift in the feature distribution is not induced by an anomaly. To summarize, the max and min value of a feature is recomputed when the percentage of the data lying in the α and the $1-\alpha$ percentile is under a threshold th (for example 90 %) during more than n slots (for example $n * \Delta T$ equals 60 minutes).

IV. THE CLUSTERING STEP

Every feature space (or data matrix X) is then processed independently and partitioned in order to identify the clusters and outliers in the data. In order to overcome the curse of dimensionality, the feature space is split in different subspaces, each being processed independently. The curse of dimensionality phenomena occurs with high dimensions. In high dimensions, distance becomes meaningless and every point tends to become an outlier. Due to this curse, unsupervised network anomaly detectors tend, in high dimensions, to detect every flow as an outlier, i.e. as an anomaly. Our solution is a robust and efficient detector which addresses this issue by applying subspace clustering and evidence accumulation techniques. It divides the whole space in subspaces and partitions each subspace independently. To speed up the execution time of the clustering step, it takes advantage of a grid and incremental clustering algorithm. Instead of clustering directly points, grid clustering algorithms divide the feature space in cells where points are placed, and partition the cells. As the number of cells is much lower than the number of points, their complexity is lower than usual clustering algorithms which cluster points like DBSCAN and K-means.

Among available grid clustering algorithms, GDCA (Grid Density-based Clustering Algorithm) [4] offers many advantages; it is a density based grid clustering, able to discover any shape of clusters and to identify noise. Our solution takes advantage of both the discrete time-sliding window and the incremental grid clustering algorithm IGDCA. Our solution can be divided in three steps. The preprocessing step during which each feature space X is updated every micro-slot and then divided in N two-dimensional subspaces: $(X_1, X_2, ..., X_N)$. Next, the clustering step updates the partition of each subspace. To update the partition of a subspace X_i, IGDCA needs as input the points to add X_i^{add} and the points to remove X_i^{rem} from the previous partition P_i^{old}. Thus, for each subspace, two matrices are provided in order to update its partition. It can be noticed that the current subspace X_i^{new} can be computed from these two matrices and the previous subspace denoted X_i^{old} as follows:

$$X_i^{new} = X_i^{old} - X_i^{rem} + X_i^{add} \tag{3}$$

For every subspace, IGDCA outputs a new partition P_i^{new}. Among available grid clustering algorithms, GDCA offers many advantages; it can discover any shape of clusters and identify outliers. In GDCA, a group of consecutive dense cells forms a cluster. For our solution we have slightly modified GDCA. We denote $S = (A_1, ..., A_k)$ a k dimensional space where $S = (A_1, ..., A_k)$ are the dimensions of S. Our modified version of GDCA takes as input a feature space X of size

$|F|*k$ made up of k-dimensional points. GDCA can be divided into four steps:

1) The space is divided into non-overlapping rectangular units or cells. The units are obtained by partitioning each dimension into intervals of size l. Each unit has the form $u = \{r_1, ..., r_k\}$ where $r_i = [l_i; h_i)$ is a right open interval in the partitioning of A_i.
2) Points are placed into the cells. Cells containing at least $minDensePts$ are marked as dense units. A point $x = \{x_1, ..., x_k\}$ belongs to a unit $u = \{r_1, ..., r_k\}$ if $l_i \leq x_i < h_i$ for all u_i.
3) Set of connected dense units are grouped together to form a cluster. Two k-dimensional dense units u_1 and u_2 are connected if they have a common face or if there exists another k-dimensional unit u_3 such that u_1 is connected to u_3 and u_2 is connected to u_3. Units $u_1 = \{r_1, ..., r_k\}$ and $u_2 = \{r'_1, ..., r'_k\}$ have a common face if there are $k-1$ dimensions, assume $A_1, ..., A_(k-1)$ such that $r'_i = r_i$ for all $i \in [1; k-1]$ and either $h_k = l'_k$ or $h'_k = l_k$.
4) It returns the clusters whose number of points is superior to $minClusPts$.

Points situated in cells which do not belong to any cluster are considered as outliers. Let n be the total number of points, c the number of cells, c_n the number of non-empty cells, and c_d the number of dense cells, DGCA time complexity is then $O(n + c_d.log(c_n))$. For the sake of comparison, DBSCAN complexity is $O(n^2)$ and $O(n.log(n))$ when used with an R-tree index. Therefore, and as usually $c_d < c_n \ll c \ll n$ holds, DGCA has a lower complexity than DBSCAN.

There is an incremental version of GDCA called IGDCA (Incremental GDCA). IGDCA is able to update a feature space partition and, for a given input, outputs the same partition as GDCA.

IGDCA requires three input parameters (the same as GDCA): l the length used to divide each dimension into intervals, $minDensePts$ the minimum number of points in a dense unit (or cell) and $minClustPts$ the minimum number of points to return a cluster. As in GDCA, the space is divided into non-overlapping rectangular units or cells. The units are obtained by partitioning each dimension into intervals of length l_i. At each feature space update, IDGCA upgrades the previous partition. It takes as inputs the points to add X^{add}, the points to remove X^{rem} and the points to update X^{up} from the previous partition. At each feature space update, IGDCA upgrades the previous partition in five steps:

1) for each point $x_{up} \in X^{up}$, IDGCA identifies its new unit u_{new} and its previous unit u_{old} (the unit to which it belonged at the last update). If u_{new} is different from u_{old}, IGDCA removes the point x from u_{old} and adds it to u_{new}. It then removes every point $x_{rem} \in X^{rem}$ from its unit and place every point $x_{add} \in X^{add}$ into its unit.
2) it then computes two lists: the lists of new and old dense units $listNewDenseUnits$ and $listOldDenseUnits$. The first list contains the units which are now dense and were not dense in the previous partition. The second list contains the units which were dense in the previous partition, and wich are not dense any longer.
3) every unit u in $listOldDenseUnits$ is then processed and a list of units to re-partition $listUnitToRep$ is built. For each unit $u \in listOldDenseUnits$ IGDCA removes u from the cluster C to which it belongs. If the unit u has two neighboring units which belong to the cluster C, then all the units of the cluster which are still dense are put in $listUnitToRep$ and the cluster is removed. Indeed, if the unit has two neighbors belonging to the cluster, its removal from the cluster may lead to a division of the cluster into two little clusters. Therefore all the units of the cluster which are still dense need to be re-partitioned. Once every unit in $listOldDenseUnits$ has been processed, the dense units in $listUnitToRep$ are grouped to form clusters. Set of connected units forms a cluster.
4) every unit u in $listNewDenseUnits$ is processed. Each unit can either (1) form a new cluster (2) be absorbed by an existing cluster (3) or merge multiple clusters in one. If the unit u has no neighboring dense unit, IGDCA creates a new empty cluster to which it adds u. If the unit u has at least one dense neighboring unit and all its dense neighboring unit(s) belong to the same cluster, IGDCA adds u to this cluster. If the unit u has two or more neighboring dense units belonging to different clusters, IGDCA merges these clusters in one and adds u to this new cluster.
5) it returns the clusters whose number of points is superior to $minClusPts$. Points which do not belong to any of these clusters are considered as outliers.

V. Evaluation

A. Evaluation methodology

To validate our solution, we use SynthONTS, a new public and evolving ground truth that is made available on demand for the network community[1]. This ground truth has been built using real traffic traces collected on the access network of a large Spanish Cloud Service Provider (CSP), which synthetic anomalies have been injected in. This work has been performed in the context of the European ONTIC project (Online Network TraffIc Characterization) and takes advantage of the ONTS traffic dataset collected during the last two years. These synthetic traces contain two kinds of network anomalies:

- Anomalies already existing in the real-life dataset (ONTS dataset)
- Anomalies artificially injected

The real anomalies already existing in the dataset have been found by manual inspection and using ORUNADA . The anomalies found are network scans, port scans, large ICMP echo, a large multipoint to point flooding attack and RST

[1]Interested readers can request an access to the SynthONTS ground truth by following the link: http://ict-ontic.eu (Data page). This link points to a request form together with an Acceptable Use Agreement (according to the Spanish privacy law). After filling the request and signing the agreement, an access to the requested dataset will be granted.

attacks. The synthetic attacks injected in the traces are various network scans, ports scans, large ICMP echo attacks, RST attacks, smurf and fraggle attacks, various flooding attacks of various intensities, and brute force attacks. More information about this SynthONTS dataset can be found in [5]. The ONTS traces that have been gathered contain a significantly large amount of traffic: 300,000 packets/s and 1.2 Gbit/s on average.

The following evaluations have been performed on a single machine with 16 GB of RAM and an Intel Core 5-4310U CPU 2.00GHz. The window size Δt is set at 15 seconds, as recommended in [10].

B. Detection performance

H-ORUNADA detection performance is compared with two detectors representing the two most famous families of detectors encountered in the literature, and for which tools are available (and that we were able to install and to run): a PCA-based [9] and a DBSCAN-based [15] detector. Figures 3 displays the ROC curves obtained with the different detectors. To generate these curves, we vary the minimum number of points required to form a cluster (for the DBSCAN-based detector and H-ORUNADA) and the number of principal component directions of the abnormal subspace (for the PCA-based detector). These curves show that H-ORUNADA has a high detection rate with a low number of false positives and outperforms the other detectors. Its performance can be explained by the fact that it does not make any assumption on the data distribution unlike detection methods based on PCA and does not suffer from the curse of dimensionality like the DBSCAN-based detector. Indeed, the DBSCAN-based detector is sensitive to high dimensions because it processes the whole feature space directly and does not divide it into subspaces.

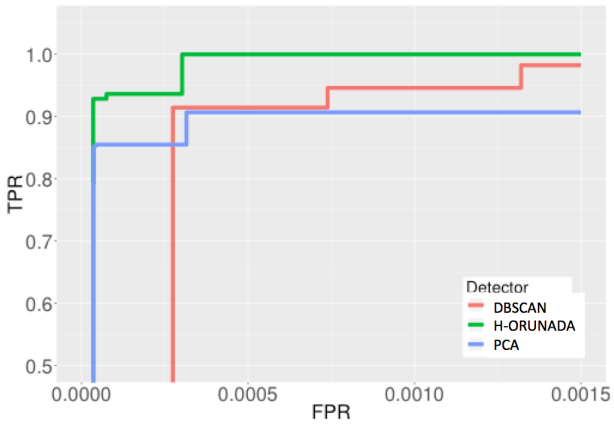

Figure 3. ROC curves for detection performance evaluation

The validation result however shows that H-ORUNADA configured with the seven aggregation levels as described in Table I does not detect all anomalies contained in the SynthONTS dataset (the point (0;1) is not part of the ROC curve). A brute force attack using a variety of port numbers was no detected. We then added port numbers in the list of features to be looked at when detecting anomalies, then adding port based aggregations. Considering port numbers, the new version of H-ORUNADA is now able to detect all anomalies of the SynthONTS dataset. Note however that no aggregation level can detect every network anomaly. This validation shows that:

- It is very important to use multiple aggregation levels to get different views of the data.
- Computing only statistics on the entire network (aggregation level 7), what is often done in algorithms encountered in the literature, to gain time are only able to detect huge anomalies. Indeed, these statistics only give a coarse view of the network.

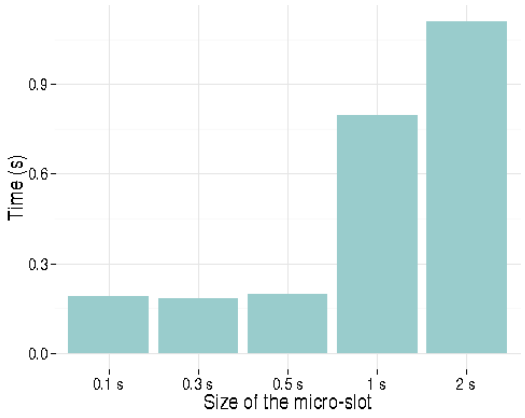

Figure 4. H-ORUNADA execution time according to the micro-slot length

C. Detection time

The following evaluation aims at measuring the detection time for H-ORUNADA depending on its configuration. It is especially foreseen whether H-ORUNADA can be run online and in real-time. Given the algorithm designed for H-ORUNADA, the micro-slot size is of essential importance. The smaller the micro-slot size, the faster H-ORUNADA identifies the anomalies and the network administrator takes counter-measures. Thus, we have evaluated H-ORUNADA execution time with different micro-slot sizes. The results are displayed in Figure 4. It can be noticed that a reduction of the micro-slot size improves H-ORUNADA average runtime. H-ORUNADA can process the incoming traffic faster than it arrives as long as the micro-slot size is superior or equal to 0.3 seconds. This evaluation proves that H-ORUNADA can detect online with a low delay network anomalies (less than half a second elapses between an anomaly occurrence and its detection).

VI. Distribution of H-ORUNADA using Spark Streaming and evaluation on the Google Cloud Platform

In the context of the ONTS traffic, H-ORUNADA proved is efficiency both in terms of detection and speed. It is especially shown that it can run in real time on network links up to several GBit/s. However, given the increase in the amount of traffic

to be transmitted, we wanted to test whether H-ORUNADA could work with several hundreds of Gbit/s of traffic. For that purpose, we implemented H-ORUNADA on the Spark Streaming big data platform, and test it on the Google Cloud Platform.

A. Distribution of H-ORUNADA using Spark Streaming

H-ORUNADA is by design a parallel program and is implemented in a distributed way. Computations on every aggregation level and every subspace in every aggregation level can be performed in parallel. Therefore, to speed up the execution time, H-ORUNADA is implemented to be distributed on a cluster of servers using Spark and more precisely Spark Streaming. We use Spark Streaming as H-ORUNADA computes a continuous stream of network traffic.

Many applications benefit from acting on data as soon as it arrives. Spark Streaming is the Spark's module for processing streaming of incoming data. Much like Spark, it is built on the concept of RDDs (Resilient Distributed Datasets), Spark Streaming provides an abstraction called DStreams, or discretized streams. A DStream is a sequence of data arriving over time. Internally, each DStream is represented as a sequence of RDDs arriving at each time step (hence the name "discretized"). DStreams can be created from various input sources, such as Flume, Kafka, or HDFS (Hadoop Distributed File System - in our case we use HDFS as input source). Once built, they offer two types of operations: transformations, which yield a new DStream, and output operations, which write data to an external system. DStreams provide many of the same operations available on RDDs, plus new operations related to time, such as sliding windows. More information on the implementation of H-ORUNADA on Spark Streaming is provided in [5], especially for the ORUNADA time sliding window (kept in H-ORUNADA).

B. Experimentation on the Google Cloud Platform

The Google Dataproc allows running powerful and cost-effective Apache Spark and Apache Hadoop clusters easily on the Google Platform. Using a simple interface, clusters can be easily and quickly created. They can be resized at any time: from three to hundreds of nodes and run Spark or Hadoop applications. The Google DataProc also provides the Spark and Hadoop ecosystem tools, libraries, and documentation and offers frequently updated and native versions of these tools. It provides the latest version of Spark (Spark 2.0.2) and Hadoop (Hadoop 2.7). For its ease and flexibility of utilization, its high-quality documentation and its up-to-date Spark and Hadoop versions, we decide to run our solution on the Google Dataproc.

Our cluster is made up of only one type of machine with 8 vCPU* and 30GB of memory. According to the Google Cloud Platform documentation, a vCPU is a virtual CPU, it is implemented as a single hardware hyper-thread on whether a 2.6 GHz Intel Xeon E5 (Sandy Bridge), or a 2.5 GHz Intel Xeon E5 v2 (Ivy Bridge), or a 2.3 GHz Intel Xeon E5 v3 (Haswell), or a 2.2 GHz Intel Xeon E5 v4 (Broadwell). For every experiment, we specify the number of machines used

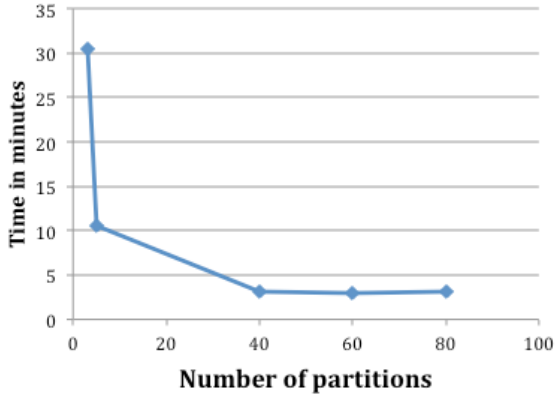

Figure 5. Spark streaming H-ORUNADA execution time according to the number of partitions (for slots of 2 minutes)

in the cluster. We could take advantage of more powerful machines with more vCPU or more memory. However, using machines with 8 vCPU* and 30GB of memory is a good compromise between quality/price.

The way the Google dataproc can be configured is detailed in [5]. It especially explains how to configures the level of parallelism (i.e. the number of partitions), how to serialize computings, and how to configure the size of the cluster (i.e. the number of cores involved in th experiment).

We performed two kinds of experiments on ONTS traces collected on February 9th, 2017, and representing 384 GB of network data. The first experiment aims at evaluating the impact of the level of parallelism of our application run time execution. When Spark Streaming runs tasks, Spark can only run 1 concurrent task for every partition of an RDD. To modify the level of parallelism, it is possible to play on the number of partitions of the data. As our application partitions each subspace independently and in parallel (the clustering is not distributed), the maximum level of parallelism equals to the total number of subspaces. We do not distribute the clustering step as it is an increment grid clustering algorithm with a low complexity. However, it could be possible to distribute it if needed, i.e. if a subspace has many points, the subspace could be split in multiple slices, and each slice could be partitioned independently. By computing the number of features and the number of subspaces per aggregation level, we found out that the maximum number of partitions that should be used equals to 1288. The results of the experiment are depicted on Figure 5. We can notice that the rune time execution of our solution decreases till reaching a limit. Beyond a certain number of partitions the run time execution of our solution, increases slightly. These results can be explained by the fact that beyond a certain number of partitions Spark overhead (serializing, repartitioning which leads to shuffling) is more important than the gain of creating a new partition.

The second experiment aims at evaluating the impact of the size of the cluster on the speed of H-ORUNADA. Once again, the performance of our application should not improve once there is more than one core per subspace, i.e. 1288 cores. For this experiment, we also use different level of

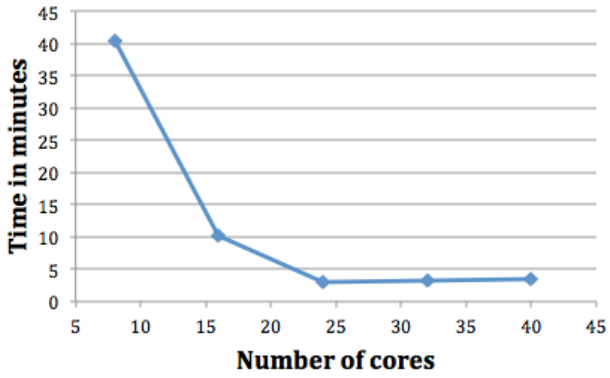

Figure 6. Spark streaming H-ORUNADA execution time according to the number of partitions (for slots of 2 minutes)

parallelisms (i.e. different number of partitions). For each size of cluster, we display the best results obtained with the optimal number of partitions. The results are depicted on Figure 6. Once again we can notice that the rune time execution of H-ORUNADA decreases till reaching a limit. Beyond a certain number of cores the run time execution of H-ORUNADA, increases slightly. These results can be explained by the fact that beyond a certain number of core, Spark overhead (serializing, repartitioning, shuffling) is more important than the gain of adding new cores.

The results obtained with spark streaming appear not to be real-time (whereas with a normal parallel implementation in C it is), it takes more than 2 minutes to process a slot of two minutes of traffic. These results can be partially explained as follows:

1) the small-files problem. HDFS deals with blocks of 64MB and our incoming text files are in average of 20 MB. Therefore, our inputs are not well suited to HDFS.
2) Spark Streaming's performance can be improved by using larger batches, but larger batches moves further away from real-time processing towards stored batch mode, and exacerbates the stream processing and real-time, time-based analytics issues.
3) It consumes a lot of memory and issues around memory consumption are not handled in a user friendly Manner.
4) Spark streaming is mostly used for web application which only does a simple process for every batch of data. We think that it is not well suited for our solution. Indeed, our solution needs to perform an extensive process at every batch of data.

Furthermore, Spark Streaming tuning and configuration must be modified according to the size of the incoming data and is then not well suited to our use cases. Spark should allow a fast distribution of an application over a cluster of servers.

VII. Conclusion

This paper presented two main contributions: the first one deals with an improvement of clustering techniques for network traffic anomaly detection. It consists in considering short term traffic history to be able to better identify clusters corresponding to anomalous classes of traffic, then allowing an autonomous classification of clusters and related classes of traffic. Its detection and classification accuracy g-has been evaluated on the new SynthONTS ground truth. Evaluation results exhibited that it is very important, for accurate detection, to use multiple aggregation levels to get different views of the data. This paper also exhibited that the evaluation results are highly dependent on the ground truth used. Up to our knowledge no existing public ground truth is satisfactory by providing full certainty on the labeled anomalies, and being complete. Given the effort put on the building of SynthONTS, the confidence that we can get on the accuracy of labels is high. It is however far from being complete. The SynthONTS building is still a running project to provide more traces with more various anomalies. The second contribution deals with implementing H-ORUNADA on the Spark streaming big data platform, and running it on the Google dataproc for evaluating its scalability limit. Results are not satisfactory, as a simple parallel implementation in C on a simple PC performs better. Our feeling with this result is that Spark streaming must provide more dynamic distribution over a cluster, and more flexible and efficient file system and memory management to adapt to network traffic analytics requirements.

References

[1] C. C. Aggarwal and P. S. Yu. Outlier detection for high dimensional data. *SIGMOD Rec.*, 30(2):37–46, May 2001.
[2] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita. Network Anomaly Detection: Methods, Systems and Tools. *IEEE Commun. Surveys & Tutorials*, 16, 2014.
[3] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41(September):1–58, 2009.
[4] N. Chen, A. Chen, and L. Zhou. An incremental grid density-based clustering algorithm. *Journal of Software*, 13(1), Aug. 2002.
[5] J. Dromard, V. Baudin, P. Owezarski, A. Mozo-Velasco, B. Ordozgoiti, and S. Gomez-Canaval. D4.3: Experimental evaluation of algorithms for online network characterizations. Technical report, FP7 European ONTIC project: Online Network Traffic Characterization, January 2017.
[6] J. Dromard, G. Roudiére, and P. Owezarski. Online and scalable unsupervised network anomaly detection method. *IEEE Transaction on Network and System Management (TNSM)*, 14(1), January 2016.
[7] D. Ippoliti and X. Zhou. Online adaptive anomaly detection for augmented network flows. In *IEEE 22nd Int. Symp. on Modelling, Analysis Simulation of Comput. and Telecommun. Syst.*, pages 433–442, Sept. 2014.
[8] A. Lakhina and M. Crovella. Mining anomalies using traffic feature distributions. *ACM SIGCOMM Comput. Communication Review*, 35(4):217, 2005.
[9] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Conf. on Applications, Technologies, Architectures, and Protocols for Comput. Commun.*, pages 219–230, New York, NY, USA, 2004. ACM.
[10] J. Mazel. *Unsupervised network anomaly detection*. PhD thesis, INSA Toulouse, France, Dec. 2011.
[11] A. P. Muniyandi, R. Rajeswari, and R. Rajaram. Network anomaly detection by cascading k-means clustering and c4.5 decision tree algorithm. *Procedia Engineering*, 30:174 – 182, 2012.
[12] L. Portnoy, E. Eskin, and S. Stolfo. Intrusion detection with unlabeled data using clustering. In *Proc. of ACM CSS Workshop on Data Mining Applied to Security (DMSA)*, 2001.
[13] B. Schökopf, J. Platt, and T. Hofmann. *In-Network PCA and Anomaly Detection*, pages 617–624. MIT Press, 2007.
[14] Robin Sommer and Vern Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. *IEEE Symp. on Security and Privacy*, 0(May):305–316, 2010.
[15] T. M. Thang and J. Kim. The anomaly detection by using dbscan clustering with multiple parameters. In *Information Science and Applications (ICISA)*, pages 1–5, April 2011.