# Efficient Identification of Cloud Gaming Traffic at the Edge

Philippe Graff, Xavier Marchal, Thibault Cholez, Bertrand Mathieu, Olivier Festor

# Efficient Identification of Cloud Gaming Traffic at the Edge

Philippe Graff*, Xavier Marchal*, Thibault Cholez*, Bertrand Mathieu†, Olivier Festor*
*Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France, {first.last}@loria.fr
†Orange Innovation, Lannion, France, bertrand2.mathieu@orange.com

***Abstract*—Cloud Gaming (CG) has been gaining a lot of interest and major actors have entered this market such as Google, Nvidia, Sony or Microsoft. They operate CG platforms that attract an increasing number of players worldwide. This type of traffic is highly demanding for network infrastructures because it requests simultaneously high bandwidth, low delay and no traffic degradation (interruptions or jitter) to ensure a good end-user's QoE. To improve the delivery of low-latency applications, new Active Queue Management architectures like L4S (Low Latency, Low Loss, Scalable Throughput) are proposed. Currently, traffic is routed to a low-latency queue only based on the presence of the Explicit Congestion Notification bit (ECN) in the IP header, but this is too restrictive and can be easily manipulated. Instead, we aim at analyzing and detecting CG traffic based on its inherent characteristics, to forward the packets in the low-latency queue. This paper presents our models to efficiently detect CG traffic based on flow-level features among other high-bitrate applications transported over UDP. The evaluation proves that our model based on decision trees achieves very good results (98.5% accuracy) and can be realistically deployed as a Virtualized Network Function at the edge, handling more than 10Gb/s of medium-sized flows on a low-end server. Our network captures and source code are open to ensure reproducible results.**

***Keywords*—Low Latency Traffic, Traffic Classification, Cloud Gaming, Machine Learning, Virtualized Network Function**

## I. Introduction

CG was initially thought of in the late 2000s [1]. The principle is to decouple the player's device from the rendering hardware through the network. Games controls are sent through the Internet to a computer or console located in a datacenter to which the game is deployed and returns the multimedia flow. Early versions failed to gain traction because of limited network performance and slow encoding. Since then, network technologies have evolved and now offer higher bandwidth and lower delays. Several enterprises have launched Cloud Gaming platforms in the past few years, such as Google Stadia (STD), Nvidia GeForce Now (GFN), Microsoft Xbox Cloud Gaming (XC), Sony Playstation Now (PSN) or Amazon Luna. However, this type of traffic is still highly demanding for current network infrastructure. These networks are required to maintain several constraints such as high bandwidth, low delay and low jitter. Failing to which the Quality of Experience (QoE) can degrade significantly. To limit latency, some enterprises implement their own advanced mechanisms, such as Outatime [2] or Stadia's negative latency algorithms [3]. On the opposite, some solutions do not even leverage congestion control mechanisms to limit latency increase caused by degraded network conditions [4], which can negatively affect user experience. Most implement end-to-end mechanisms that take often too much time to detect and react to network issues, leaving the service hard to play with stuttering experienced at the user side, at least during transient phases. The main cause of latency is due to the *bufferbloat* phenomenon [5], [6] that occurs at the edge of the network, resulting in problematic queues. Large buffers are indeed placed at the edge to cope with variations in network capacity, especially in cellular networks. The problem is that these buffers are often full, which adds significant and fluctuating delays.

Meanwhile, new network technologies propose to optimize queuing in network equipment. L4S [7] is one of the most promising solutions. Two queues are defined: one for the common traffic and one for the low-latency (LL) traffic. The LL queue forwards packets with higher priority, respecting a low delay (in the order of 1 ms). Both queues are coupled so as the LL queue does not starve the classic one. In the L4S architecture, a classifier defines in which queue to forward the current packet. Currently, this classification only relies on the ECN bit in the IP header of the packet. It is purely declarative by the end-host and not protected. It has been proven to be vulnerable to misconfigurations or malicious behaviors [8], [9]. Any application/endpoint can cheat and set this bit to "1" to be processed by the network as a LL traffic. To avoid this, we advocate the use of an intelligent classifier, which will be able to analyze and detect the LL behavior of the sessions transiting via the L4S node. CG being a LL service, we investigate the network behavior and patterns of this traffic in order to be able to classify it as a LL application. Moreover, since this classification should be done in real-time by a Virtual Network Function (VNF), an efficient method is defined to classify the flows at the edge. The contributions of this paper are the following:

- We select flow-level features and evaluate three Machine Learning (ML) models that aim to detect CG traffic among other high-bitrate UDP applications. The best models provide excellent results (98.5% accuracy);
- We publicly release a dataset composed of network traces (`pcap` files) of the four main CG platforms, and other services running on UDP;
- We propose a high performance VNF implementation of

our classifier as micro services. It can classify network flows at line rate (more than 10Gb/s). The code is also open source.

The remainder of this paper is organized as follows. Section II presents the related work on CG platforms and on traffic classification. Then, section III describes our dataset creation. Section IV details the features' creation from raw data. We then present the ML models used. Section V focuses on hyper-parameters fine-tuning. Once done, we evaluate our models' performance in section VI. Section VII presents the implementation of our CG classifier as micro-services. Finally, we discuss the obtained results in Section VIII and conclude this study in Section IX.

## II. Related Work

### A. Cloud Gaming

In [10], Di Domenico & al. study 3 of the biggest CG platforms: STD, GFN and PSN, while [11] made an emphasis on STD. They highlight the different protocols used before and during a game session. All these three platforms transport data over UDP. STD and GFN rely on the Real-time Transport Protocol (RTP) to stream audio and video. PSN uses a proprietary protocol. It is also emphasized that several flows are multiplexed on a single UDP port for STD and PSN. The GFN platform, on the other hand, uses several port pairs. In addition, communications are secured by the Datagram Transport Layer Security (DTLS). The diversity of transport protocols and the use of encryption make it a non-obvious task to classify CG traffic. It can be considered as a special case of encrypted traffic classification.

Another difficulty is that CG traffic is highly sensitive to network conditions. This creates traffic variations to which the ML model must be robust. In [4], [12], the authors alter the network conditions (available bandwidth, latency, jitter and packet loss) and observe the consequences on CG traffic. In [12], the authors focus on the GFN platform. The authors emphasize that the bit rate drops sharply as soon as latency is introduced. Moreover, the platform takes a long time to resume to a normal state. So, there would be a wrong estimation of the available bandwidth, based on the Round Trip Time (RTT). The QoE is highly impacted by this phenomenon. In [4], the four main CG platforms are studied, namely STD, GFN, PSN and XC. Network constraints are enforced at several levels of intensity and are applied before and during a game session. The authors point out that the four platforms exhibit different behaviors when facing network constraints, reacting with variable intensity. They report that CG platforms may have difficulties stabilizing their bit rate after a sudden network change and take a long time to recover once the network returns to a normal state. Even worse, PSN is blind to the added latency.

### B. Encrypted Traffic Classification

Traffic classification is a critical task for network operators. The objective is to gain insights of the traffic transiting in their networks. This allows them to better configure the network to meet security or performance requirements. In our case, our goal is to classify CG traffic so as to minimize packet latency. There are several ways to perform network traffic classification. The "*Port based*" approach consists in matching the TCP/UDP port numbers used with the default application ports given by the Internet Assigned Numbers Authority (IANA). Because most applications use dynamic port numbers, which is the case of CG applications [10], this approach is now obsolete. In our case, we can at least filter some well-known UDP applications with a standardized port number, like for example DNS (*port 53*), to classify only relevant traffic. The "*Payload based*" approach consists in studying the payload of the packets [13]. This approach provides good results but suffers from two major downsides; its high computational cost and its inability to handle encrypted traffic that represents the great majority of traffic nowadays. According to [14], the proportion of encrypted traffic reaches 85% and is still growing. In our case, CG traffic is indeed encrypted with DTLS [11].

A third approach consists in considering the statistical properties of the different network flows to be classified and learning them automatically with ML methods. The authors of [15] made a survey on methods for encrypted traffic classification. They note that supervised machine learning methods based on statistical features tend to perform well but regret the lack of study considering TLS traffic instead of SSH and the availability of labeled datasets. A recent study [16] deals with encrypted HTTP/2 traffic. The goal is to filter traffic when a prohibited keyword is searched instead of filtering the whole website and without leaking the exact keyword. To do this, they extract characteristics from the traffic such as TLS connection statistics, record sizes, etc. and use a Random Forest classifier. Their approach achieves a good accuracy, but it depends on the service being monitored. Studies such as [17]–[19] proposed Deep Learning (DL) models which provide good results for classifying different types of services (video streaming, social networks, web, emails, etc.) and applications (Youtube, Netflix, Facebook, Instagram, etc.), both with HTTP/2 and QUIC (Quick UDP Internet Connections) traffic. Other approaches rely on representation learning. In [19]–[21], the authors use image recognition techniques to perform classification. They convert raw data into a 2D image. Each image is then submitted to a Convolutional Neural Network (CNN). The output layer consists of as many neurons as there are classes to predict. In [20], the researchers distinguish two scenarios. In the first one, a binary classifier is applied to determine whether it is malware traffic and followed by two 10-classes classifiers designed to identify sub-classes of malware and normal traffic. The second scenario consists of a single 20-class classifier, able to classify all types of traffic at once. In [21], the authors propose a CNN architecture composed of 7 layers. The 2D image representing a flow is made from packet sizes and packet arrival times. In their study, they just consider unidirectional flows, defined by a 5-tuple. When doing traffic categorization on regular encrypted traffic sessions (*non-VPN traffic*), their model is outperformed by a simple Decision Tree

classifier.

No study so far has tried to classify CG traffic. Our goal is to define a relevant set of features and to evaluate different ML algorithms to select the most suitable to achieve efficient classification of CG flows and ensure an operational applicability.

## III. Dataset Creation

In this section we describe our dataset and how we built it. It was made available for the sake of reproducibility[1]. It is composed of different network captures (*pcap files*) collected with *Wireshark* (*v2.6.20*) at the client side.

We have developed a first dataset, representative of Cloud Gaming traffic composed of the 4 main CG platforms available in Europe to date[2]: STD, GFN, XC and PSN. We only considered full-fledged platforms where the service provider operates the servers in contrast to other CG services that must be instantiated by users like Steam Remote Play, Parsec, Moonlight, Rainway, among others. We made this choice because instantiating CG software by ourselves would result in numerous biases. Indeed, the chosen hardware, its location and the software configuration can have an impact on the performance. In our case, we use commercial services so that the collected traffic is representative (i.e., comes from shared servers, traverses Internet, etc.).

To have a complete dataset, we ensure to collect captures corresponding to different games, frame rates and video resolutions for each platform. Please note that CG platforms support different maximum resolution. For example, only STD allows a game to stream in 4K. According to the possibilities of each platform, we captured resolutions of 720p, 1080p, and 4K and frame rates of 30fps and 60fps. Among the types of games used during our network captures, there are *Shooter* games (First Person (*FPS*) and Third Person (*TPS*)), *Platform* games and *Racing* games. When possible, we also vary the input option; keyboard+mouse or controller. The type of input can indeed have an impact on the characteristics of the "*client-to-server*" traffic.

A second dataset representing "*non-CG*" (NCG) traffic focuses on high-bitrate applications transported over UDP. If protocol type and port number are not sufficient to detect CG traffic, they can at least be used to easily filter irrelevant traffic. We also neglect DNS traffic (UDP port 53) among the considered UDP applications. We consider 4 types of applications in the "*non-CG*" traffic class;

- Video conferencing (VC): with *Discord*, *Google Meet*, *Jitsi Meet* and *Cisco Meeting* captures;
- Video streaming (VS): *YouTube* and *Facebook Watch* captures;
- Live video streaming (LV): with *YouTube Live* and *Facebook Live* captures;
- Facebook navigation (FN): Facebook news feed browsing, consultation of several profiles.

[1]https://cloud-gaming-traces.lhs.loria.fr/data.html

[2]November 2022

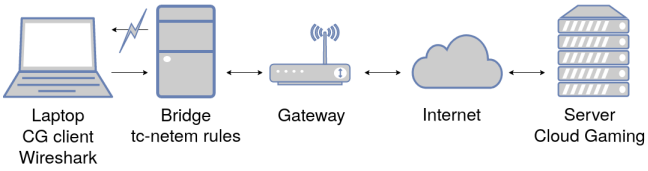


Fig. 1: Testbed to collect network traffic

These applications exhibit high bitrate UDP flows, just like CG. However, they have less stringent latency requirements. Videoconferencing applications rely heavily on RTP for transport like CG. A Facebook navigation relies on the QUIC protocol, and therefore on UDP for the transport layer. QUIC is becoming a new standard [22] supposed to replace HTTP/2 overall. Because of the importance of this type of traffic, we believe it is interesting to take it into account.

Since CG platforms can heavily adapt their traffic to network conditions [4] and to ensure that our classifier is general enough to recognize CG traffic in all cases, we made another autonomous set of network captures by playing under degraded conditions. Figure 1 shows a representation of our testbed. We placed a network bridge between our gateway and the CG client, which alters the network conditions using `tc-netem` rules.

We altered latency, jitter, available bandwidth and packet loss on the downlink. For each platform, we start by applying our constraints, and then we launch a game session. We chose the following values that were used in [4] and proven sufficient to trigger the congestion control mechanisms of CG platforms.

- Latency: +20ms;
- Jitter: +5ms;
- Losses: +5%;
- Bandwidth: 10Mbps;

The resulting dataset is called "CG *disrupted*" (CGP).

Overall, we made 57 distinct captures accounting for 17GB of traces, 19 million packets and lasting 240 minutes.

## IV. Methodology

In this section, we first present the features used to classify the data. Then, we focus on the classification methods. We start by presenting our baseline; "*Thresholds classifier*". This method classifies data according to the number of features falling between pre-defined thresholds. We then briefly describe the other two methods used; a Decision Tree (DT) classifier and a Random Forest (RF) classifier.

### A. Considered Features

For each network conversation between two IP addresses, we compute several features per time window. We choose to not consider port numbers to benefit from a higher level of abstraction. Indeed, some platforms use multiple ports to stream audio and video, while other rely on application level or RTP multiplexing (SSRC (*Synchronization Source*) [4], [10].

Because CG traffic is by nature asymmetric (player's commands in one direction and multimedia flow in the other one),

we differentiate the statistics from the server-to-client direction (downlink) from those from the client-to-server direction (uplink). We compute the 6 following statistics for each direction of traffic separately:

1) mean packet size;
2) standard deviation of packet sizes;
3) average Inter-Arrival Times (IAT);
4) standard deviation of IATs;
5) total number of packets;
6) total size of application data;

We expect to have large packet sizes on the downlink to transport multimedia streams and small packets sizes on the uplink (user's inputs). IATs should exhibit frequent and regular patterns. In the downstream direction, they are driven by the constant number of frames per second (30fps or 60fps). In the upstream direction, they depend on the user's commands probing frequency. The standard deviation of IATs should be therefore lower than for other flows with a bursty traffic pattern like standard video streaming. The amount of data exchanged by direction should allow to differentiate from Video Conferencing that has lower bandwidth and refresh frequency requirements and a different type of information sent on the uplink (video vs commands).

Moreover, these features are computed over a predefined period. To do so, we divide the capture duration into three windows of 33ms, 66ms and 100ms. This is to ensure that at least one video frame is transmitted in a window at the lowest framerate (30fps). We will evaluate the impact of the different window durations on classification performance.

The data are represented as a vector X of features extracted per window *i*, and a vector Y composed of the associated binary labels (*CG:True* or *non-CG:False*). Each element $X[i]$ is composed of 12 features (*6 statistics in the server-to-client direction + 6 statistics in the client-to-server direction*). We will refer to these elements as *instances* throughout our study. The purpose of the classifier $\mathcal{C}$ is to decide whether or not an element is associated with CG traffic. A good classification would result in $pred_i = Y[i]$.

$$\forall i, \mathcal{C} : X[i] \rightarrow pred_i \in \{True, False\} \tag{1}$$

### B. Classification methods

We have developed a first simple classifier "*Thresholds classifier*". It is a baseline that relies on thresholds defined over the features. To configure them, we analyze the training set and observe the distribution of the 12 traffic features. We then compute the $1^{st}$ and $3^{rd}$ quartiles for each of them. This gives a lower bound $Q_{1,j}$ and a upper bound $Q_{3,j}$ for each feature $j$. When classifying an item, we compare each feature to its related thresholds; $Q_{1,j}$ and $Q_{3,j}$. The final decision is made depending on the total number of features being between the thresholds. If this number is above a certain value (it is a hyperparameter to be evaluated), the item is classified as CG.

The second model is a DT Classifier [23]. The tree is built during the training phase. Each of its nodes corresponds to a test on one of the 12 features. In our case, it consists in comparing a feature's value to a certain threshold. The thresholds are defined to obtain two homogeneous subsets. Each of these subsets is representative of a label. By default, the algorithm aims at minimizing the *Gini* index. That criterion is the probability of incorrectly classifying a random data point if it was labeled based on the class distribution. Then, the principle of classification consists in moving along the tree until reaching a leaf, each leaf being associated with a label. We are then able to classify the element considered.

Finally, a RF Classifier is considered. It trains several DTs in parallel and then aggregates the decisions of all its DTs to perform the final classification.

## V. Hyper-Parameters tuning

Our goal is to set some hyper-parameters in a way that balances the classification performance and the computation cost to achieve real time classification. First, we evaluate the impact of the duration of the time window from which statistical features are calculated (see Section IV-A). Then we focus on the hyper-parameters of the 3 classification methods we consider. We seek to optimize *Accuracy* because we want to avoid a normal flow to be misclassified as a low-latency one and disturb the L4S queue.

### A. Impact of the window size

We keep the default hyper-parameters listed in Table I for the DT and RF classifiers and perform a 5-fold cross validation on the whole training set while considering the different window sizes. The training set fairly represents both classes and even CG platforms or applications within a class. The accuracy is reported in Figure 2. For the DT classifier, we get the lowest results for a window size of 33ms. However, we only lose approximately 1% of accuracy compared to the 66ms and 100ms window sizes. In the case of the RF classifier, the results are slightly better. There is no real difference between the results achieved with the different window sizes (even if the lowest result is found for a window size of 33 ms, the difference is insignificant: less than 0.5%). We will therefore choose a window size of 33ms, which allows us to determine as quickly as possible the type of traffic we are dealing with and evaluate more *instances* per second.

### B. Hyper-parameters of the ML models

The first hyper-parameter we consider is the number of features that must fall between the thresholds defined by their 1st and 3rd quartiles to predict CG. We noticed that we could not require that all 12 features satisfy these constraints. Indeed, in this case, the classifier never recognizes CG traffic with the approximated thresholds we defined. We can see in Figure 3 the evolution of *Precision*, *Recall*, *F1-Score* and *Accuracy* according to the number of constraints to satisfy. We note that if the number of constraints is too small, the classifier tends to always label the traffic as CG. The *Precision* is then equivalent to the proportion of CG traffic in the testing set ($\approx 50\%$). Conversely, when the number of constraints to be satisfied approaches 12, the *Precision* increases at the expense

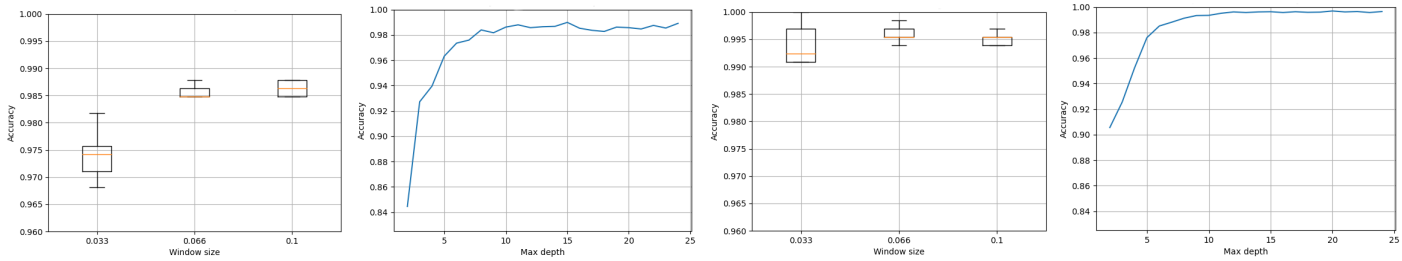


Fig. 2: Impact of Window size and Maximum depth on the Accuracy of DT (left) and RF (right) models

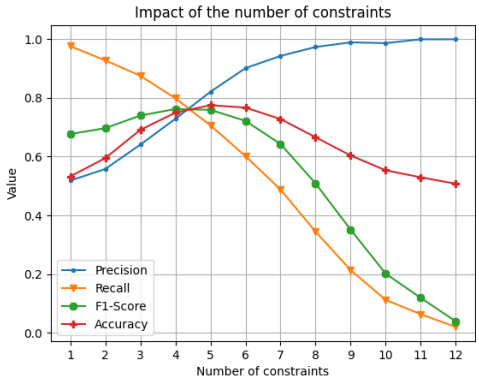


Fig. 3: Impact of the number of features to be satisfied for the Threshold classifier, w=33ms

| Hyper-Parameter | DT value | RF value |
|---|---|---|
| *max_depth* | *None* | *None* |
| *max_leaf_nodes* | *None* | *None* |
| *criterion* | *gini* | *gini* |
| *max_features* | *None* | *sqrt* |
| *min_impurity_decrease* | 0.0 | 0.0 |
| *min_samples_leaf* | 1 | 1 |
| *min_samples_split* | 2 | 2 |
| *n_estimators* | - | 100 |
| *bootstrap* | - | *True* |

TABLE I: DT & RF default hyper-parameters

| Hyper-Parameter | DT | RF | Testing Range |
|---|---|---|---|
| *max_depth* | 15 | 20 | $[2; 24]$ |
| *max_features* | 8 | 3 | $[1; 12]$ |
| *n_estimators* | - | 119 | $[80; 119]$ |
| *bootstrap* | - | $False$ | $\{True; False\}$ |

TABLE II: DT & RF : Hyper-Parameters w=33ms

of *Recall*. This means that there are few false positives, but many false negatives. In other words, a large number of CG *instances* are not recognized, which is a drawback. Therefore, it is necessary to find a compromise in the choice of the parameter. As mentioned in the beginning of the section, we seek to maximize *Accuracy*. The value maximizing *Accuracy* is the best compromise between *Precision* and *Recall*. We can see in Fig.3 that with $5$ constraints, the *Accuracy* and *F1-Score* are maximized. With this value, we have a *Precision* of $\approx 82\%$ and a *Recall* of $\approx 70\%$.

As the RF classifier is made of several DTs, both classifiers share several hyper-parameters. We find *max_depth*, that establishes the maximum depth of a tree. It constitutes an upper bound on the number of tests before classifying an item. *max_leaf_nodes* limits the total number of leaves composing a tree. *criterion* is the function that establishes the quality of a split. We let the default value, which is *gini*. This metric measures the dispersion of the classes inside a node. *max_features* defines the total number of features to be taken into account for each split. These features are chosen randomly. The rest of the hyper-parameters affect the splitting phase. *min_impurity_decrease* is the lowest impurity decrease that a split must satisfy. *min_samples_leaf* and *min_samples_split* constitute the minimum number of samples in a leaf (resp. node). Then, these last two hyper-parameters are specific to the RF classifier;

- *n_estimators*: the total number of DTs composing the RF classifier. A large number improves the robustness of the model at the cost of a greater training and predicting time.
- *bootstrap*: if *true*, each individual tree is trained with *samples* randomly taken in the dataset (*random sampling with replacement*).

We consider the four following parameters: *max_depth*, *max_features*, *n_estimators* and *bootstrap*. We decided to study the first three of them because they are related to the complexity of the models, and so their capacity to achieve real time classification. The others will be set to their default values (see Table I). We chose *max_depth* and *n_estimators* because they have a direct impact on the classifiers complexity. Furthermore, the greater the depth of the tree, the greater the risk of over-fitting the training data. To ensure that the trees do not go to deep, we limit their sizes with *max_depth*. By limiting the number of features on which the models are trained, *max_features* also limits the risk of over-fitting.
We have set a list of values for each of these hyper-parameters. It is then a matter of testing the models for each combination. For that, we used Grid search to try all the possibilities. This step is computationally expensive. We can see in Table II the values that we have retained for the 4 chosen hyper-parameters.

Because our goal is to perform live traffic classification, it is interesting to further investigate the maximum depth to find a compromise between tree depth (that affects the computation cost) and performance gain on *Accuracy*. Figure 2 (right) shows the accuracy that is achieved according to the

| Model | Accuracy | Precision | Recall | F1-Score |
|---|---|---|---|---|
| Thresholds | 0.797 | 0.860 | 0.710 | 0.777 |
| DT | 0.985 | 0.984 | 0.985 | 0.984 |
| RF | 0.996 | 0.996 | 0.996 | 0.996 |

TABLE III: CG traffic classification, *normal* conditions

| Model | Accuracy | Precision | Recall | F1-Score |
|---|---|---|---|---|
| Thresholds | 0.540 | 0.521 | 0.995 | 0.684 |
| DT | 0.924 | 0.930 | 0.917 | 0.923 |
| RF | 0.950 | 0.952 | 0.948 | 0.950 |

TABLE IV: CG traffic classification, *disrupted* testing set

| | | DT | | RF | |
|---|---|---|---|---|---|
| True Label | NCG | 1753 (**93.15%**) | 129 (**6.85%**) | 1793 (**95.27%**) | 89 (**4.73%**) |
| | CG | 157 (**8.34%**) | 1725 (**91.66%**) | 98 (**5.21%**) | 1784 (**94.79%**) |
| | | NCG | CG | NCG | CG |
| | | Predicted label | | | |

TABLE V: Confusion Matrix of DT and RF classifiers for a *disrupted* testing set

maximum depth value of DT and RF classifiers, respectively. As one would expect, both curves exhibit an asymptote. After a certain depth, the gains on *Accuracy* are negligible. For the DT classifier, we notice that from a maximum depth of 10, the *Accuracy* stabilizes. For the RF classifier, the same maximum depth could be chosen. For a maximum depth fixed at 10, we have an *Accuracy* above $98\%$. It is therefore possible to maintain a high *Accuracy* while reducing the maximum depth suggested in Table II.

## VI. Evaluation of ML models

To evaluate the performance of our classifiers, hyper-parameters are set to the values maximizing *Accuracy* for a window size of 33ms, as defined in the previous section. Only the max-depth is fixed at 10. New samples are used for this final evaluation. DT and RF models are built thanks to the *scikit-learn* [24] machine learning library.

### A. Normal testing set

The data we use to train the models was acquired under "normal" network conditions. We take care to balance the representation of the CG and NCG *instances*. Thus, we start with a base of 3356 *instances* for each traffic class. We then apply K-Fold cross Validation (KFV) on this dataset, with K=5. We can see on Table III different performance metrics related to CG recognition. A correct classification of a CG element constitutes a *True Positive* (TP). The values displayed were obtained by averaging the scores obtained by each KFV iteration.

The scores obtained by the *Thresholds classifier* are average at best, justifying the use of more advanced ML methods whose added value is clear. We see for example that the *Thresholds classifier* has $20\%$ less *Accuracy* than the DT or RF classifiers. On the opposite, DT and RF achieve excellent classification results, with a reported accuracy of respectively 98.5% and 99.6%. Note that the three other metrics *Precision*, *Recall* and *F1-Score* have same compelling value, ensuring a well-balanced classification without noticeable weakness.

### B. Disrupted testing set

In this subsection, we want to evaluate if our models are general enough to classify CG traffic captured in degraded network conditions and so to perform well under all circumstances. We keep the previous models trained on with data captured under "normal" network conditions but we will use traces captured under disrupted network conditions for testing. More precisely, we start by distributing NCG *instances* uniformly through the training and testing sets. We then add "normal" CG instances to the training set and "perturbed" CG instances to the testing set. We also make sure that NCG and CGP are equally represented in the latter. More specifically, the testing set consists of 1882 *instances* of disturbed CG traffic versus 1882 *instances* of "non-CG" traffic, all applications type considered.

Our results are shown in Table IV and the corresponding confusion matrices in Table V. The first column represents the labels to recognize and the last row shows the predictions made by the two models. The lower left corner of each matrix corresponds to the total number of *False Negatives* (FN). On the other hand, the upper right corner represents the *False Positives* (FP). For each label, we show its ratio of good predictions and wrong predictions. For example, considering CG traffic, the DT classifier has 91.66% good predictions (resp. 8.34% wrong predictions). From these two matrices, we can calculate the performance metrics given in Table IV. If we compare Tables III and IV, we can notice a slight decrease in *Accuracy* (*not more than* $6\%$) for the DT and RF classifiers. However, these results are still good, suggesting that our models are general enough to recognize CG traffic under various network conditions.

But our approach can provide even better results when we add some "perturbed" CG *instances* to the training data set. The evaluation applied to the same testing set now gives F1 scores of 97.5% for the DT classifier and 98.8% for the RF classifier. So, helping the model to learn the range of adaptation of CG traffic reduces the performance decrease from 6% to 1% when classifying such perturbed CG traffic.

In consideration of all these results, we finally retain the DT classifier. Not only are its results equivalent to those of the RF classifier, but it is also much less complex to execute (we have indeed seen in Table II that a RF classifier should be made up of 119 DTs) and the gain in classification performance is not worth it. Overall, the excellent classification performance can be explained by the fact that CG traffic has specific patterns that are well captured through by the 12 features we selected. It is thus easy for our classifier to recognize CG traffic. This is a finding of the paper. The risk of overfitting is minimal for several reasons: 1) DT is a simple model that is less subject to overfitting, 2) we limit the maximum tree depth to 10 using *pre-prunning*, 3) we use a small number of features, 4) we use K-fold cross-validation, and 5) the high accuracy is confirmed by the disrupted dataset whose traces were not

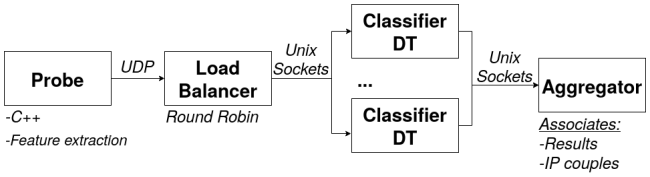


Fig. 4: Micro-service architecture of the live classifier

considered neither for hyper-parameters setting nor training. The next section will now focus on our implementation of this DT model.

## VII. Implementation of VNFs to perform live classification

In this section, we present our micro-service architecture to implement the DT classifier. The source code is available and released in open-source[3]. We assessed that the classification performance achieved by our live classifier implementation is in compliance with the results of the DT model in Table III and will focus our evaluation of the implementation on its capacity to handle a high traffic load.

### A. Micro-service architecture of the classifier

The first micro-service facing the traffic is a probe that listens to a given network interface. For each network conversation between two IP addresses, a report is created every 33ms. A report includes the 12 features described in Section IV-A and the IP address pair concerned. Each report is sent in a single UDP packet. The probe is written in `C++`, and the code is open-source[4]. The load-balancer listens to the probe's reports and is responsible for distributing the load among the different compute nodes. We distribute the load by applying a round robin strategy, but another method can be easily implemented. In our testbed, the probe and the load-balancer are executed on the same machine and communicate via the loopback interface. However, they are designed to be executed on different hosts. In fact, our micro-service architecture makes it possible to either run the compute nodes at the edge or in a cloud-provider datacenter, depending on the resources/space available. The compute nodes execute the DT model and are responsible for determining whether the received report is a CG report or not. Once the classification is done, the result is sent to the last component of the chain. The *aggregator* finally associates each IP address pair with the set of its predictions. At the end of the chain, we associate to each IP address pair the percentage of reports classified as CG. The whole architecture is illustrated in Figure 4.

### B. Performance Evaluation

We divide our evaluation into two parts. We start by evaluating the probe that observes network traffic to generate features. Then, we evaluate the components that classify the reports generated by the probe. The server used for this experiment is an *Arch Linux* system with two CPU *Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz*(released in 2012). Each CPU has *8* physical cores. The network link used has a capacity of 2x10 Gbps aggregated. We use one dedicated core for the probe and one per classifier instance. The load balancer and aggregator share a core with the system.

[3]https://github.com/mosaico-anr/CG_Classifier

[4]https://github.com/Nayald/probe-CG-detection

| Reports sent per second | Nodes | Reports processed per second |
|---|---|---|
| 10.000 | 1 | 6.951 |
| | 2 | 10.000 |
| 20.000 | 2 | 14.250 |
| | 3 | 19.997 |
| 30.000 | 3 | 20.552 |
| | 4 | 27.259 |
| | 5 | 29.196 |

TABLE VI: Performance evaluation of the DT classifier micro-service

*1) Probe Evaluation:* To evaluate the network performance of our probe, we deployed *iperf* instances on two servers generating bidirectional traffic. Only one of these servers runs the probe. Most of the given results are not limited by the probe itself, but by the CPU load induced by the *iperf* instances to generate traffic.

Regarding the throughput, we report around 30 Gbps of traffic captured by the PF_PACKET socket (only the first 82 bytes of each packet are captured by our probe), using $\sim$60% of a CPU core for the capture thread (excluding features computation). Since the probe does not capture the packet payload by focusing only on header parsing, the performance is the same for large and small packets.

The number of packets processed per second is around $\sim$2.80 Mpps with similar CPU usage. Regarding how many windows per second the probe can handle, we fixed the global throughput to 10 Gbps, that is today a classical bandwidth seen on access networks. It can handle 1000 flows at 10 Mbps without suffering losses. This costs $\sim$70% of the parse thread and $\sim$37% of the compute thread to generate a bit more than 29100 reports/s. We also tested 2000 flows of 5 Mbps. Here, traffic generation induces so much load that it is difficult to distinguish the bottleneck. Without reporting any loss, the probe sends up to 49800 reports/s. However, the parse thread is at 100% and the compute thread at $\sim$70% of CPU usage. All cores assigned to *iperf* instances exceed 94% CPU usage.

*2) Classifier Evaluation:* For this experiment, we replayed the probe's reports at predefined rates. Each experiment lasts 30 seconds. The columns in Table VI gives respectively the number of reports sent per second, the number of classification nodes deployed, and the number of reports processed per second by the live classifier. Ideally, this value should be equal to the value of the first column. To achieve a classification rate of 10.000 reports per second, a second compute node must be added. The same observation can be made when moving to a throughput of 20.000 reports per second, going from 2 to 3 classification nodes. We can see that for 30.000 reports sent per second, only 68.5% of the reports are processed when 3 classification nodes are deployed. When we go from

3 to 4 classification nodes, we notice that the processing speed increases by 24.6%. However, 8% of the reports are not classified, and a 5th additional node is required. Globally, the improvement of the classification speed brought by the addition of a compute node scales almost linearly with the number of nodes, which was expected. Please note that the classification time per window is the same for all window sizes (33ms, 66ms, 100ms), since it just considers the 12 features. In addition, the overhead of the window size on the probe is negligible. Of course, the smaller the size, the more samples are to be classified per second, which requires more computing power.

To conclude, we can say that our live classifier can efficiently process traffic on-the-fly. Based on our two-levels performance evaluation, it appears that the entire solution can support 1.000 parallel flows of 10 Mbps on a low-end x86 server if we instantiate 5 classification services in parallel.

## VIII. Discussion

Like all machine learning approaches, our models may be biased due to the dataset, despite carefully building the models. It would be interesting to further extend our datasets, especially in regards to the diversity of applications in the Non-Cloud Gaming dataset.

Considering the fast identification of CG traffic (33ms windows), we can imagine ways to stabilize and strengthen the classification decision by observing the traffic during a longer period (a few seconds) and taking the majority label among reports. The traffic between two IP addresses can also be periodically re-evaluated at a frequency to be defined (for instance every minute). This will also save resources from the live classifier to process in priority new starting flows.

The live classifier presented in Section VII follows a VNF (Virtualized Network Function) approach. The solution is built to be run in containers within a NFV architecture (Network Function Virtualization). The network performance of the probe could be improved by leveraging the DPDK technology (Data Plane Development Kit, a set of libraries and drivers that accelerates packet processing in Linux applications) if needed. The compute nodes should be automatically scaled according to the current load level by the NFV orchestrator in use, for instance Kubernetes.

Another approach would consist in doing all the processing in a programmable switch (*e.g an Intel Tofino*) thanks to the P4 (Programming Protocol-independent Packet Processors) language [25]. The advantage of this language is that a program can be written once and compiled according to the targeted network hardware. Recent advances in P4 programming have shown that it is possible to implement simple ML models like a DT in P4. In [26], the authors present a method to map trained machine learning models to match-action pipelines. It is therefore possible to use our trained DT classifier to do live classification directly on the data plane. Nevertheless, we must face the inherent limitations of the P4 language, but also those of the targeted network equipment. Among P4 inherent constraints, we can highlight that it does not contain any type for representing floating point numbers. Therefore, our mean and variance calculations will have to be approximated, which may impact the classification performance of the model. A re-training of the ML model considering the various approximations imposed by P4 seems to be inevitable if we want to classify from a P4 switch. Using P4 and programmable switches to do classification seems to be more optimal than using VNFs. Indeed, the switch itself would calculate the features, do the classification, and drive the AQM.

Please note that the classification result could be propagated by Diffserv. Therefore, it would be possible to trigger specific network treatments in the core network, for instance by selecting routes based on the latency criteria. However, the expected gain would be limited to a slight and constant latency reduction and may not be worth the effort. It is more important to help solving the bufferbloat problem at the edge since it can add unpredictable latency increase and variation, or even packet drops.

## IX. Conclusion

The main motivation behind this work was to classify CG traffic to make it benefit from the low-latency queue, as defined by the L4S architecture. This can reduce latency perturbations that degrade the QoS at the edge of the network. We have presented and evaluated three models (Thresholds, DT, RF classifiers) to detect CG traffic from other high bitrate UDP-based applications. They rely on 12 features extracted from network flows. Our results showed that CG traffic is specific enough to allow excellent classification results with the DT model. We selected this model because of its high identification accuracy (98.5%), its ability to recognize CG traffic under degraded network conditions, and its lower complexity compared to RF. We also proposed and evaluated a full implementation of the classifier as a micro-service architecture that can be deployed as VNFs (regarding the NFV paradigm) to efficiently classify CG traffic at the edge. Our implementation is scalable, can easily process 10Gb/s of medium-sized flow on a low-end x86 server. The dataset composed of full packet captures of CG traffic and other UDP applications used for this study is available for the community, as well as the source code of the micro-service implementation.

Our future work will integrate our classifier with a L4S architecture and evaluate the gain for CG platforms to respect latency requirements under limiting network conditions. Because we use simple features and models, we will also consider implementing our classifier directly in the data plane, leveraging the P4 programmable network technology. We will evaluate the benefit of this approach compared to the micro-service architecture.

## Acknowledgment

This work is partially funded by the French ANR MOSAICO project, No ANR-19-CE25-0012.

## References


[1] P. E. Ross, "Cloud computing's killer app: Gaming," *IEEE Spectrum*, vol. 46, no. 3, pp. 14–14, 2009. DOI: 10.1109/MSPEC.2009.4795441.

[2] K. Lee, D. Chu, E. Cuervo, *et al.*, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015*, G. Borriello, G. Pau, M. Gruteser, and J. I. Hong, Eds., ACM, 2015, pp. 151–165. DOI: 10.1145/2742647.2742656. [Online]. Available: https://doi.org/10.1145/2742647.2742656.

[3] M. Carrascosa and B. Bellalta, "Cloud-gaming: Analysis of google stadia traffic," *CoRR*, vol. abs/2009.09786, 2020. arXiv: 2009.09786. [Online]. Available: https://arxiv.org/abs/2009.09786.

[4] P. Graff, X. Marchal, T. Cholez, S. Tuffin, B. Mathieu, and O. Festor, "An Analysis of Cloud Gaming Platforms Behavior under Different Network Constraints," in *HiPNet 2021 - 3rd International Workshop on High-Precision, Predictable, and Low-Latency Networking*, ser. Workshops of the 17th International Conference on Network and Service Management (CNSM21), IFIP-IEEE, Izmir (Virtual), Turkey: IEEE, Oct. 2021, p. 7. [Online]. Available: https://hal.inria.fr/hal-03421031.

[5] B. Briscoe, A. Brunstrom, A. Petlund, *et al.*, "Reducing internet latency: A survey of techniques and their merits," *IEEE Communications Surveys Tutorials*, vol. 18, no. 3, pp. 2149–2196, 2016. DOI: 10.1109/COMST.2014.2375213.

[6] D. Genin and J. Splett, *Where in the internet is congestion?* 2013. arXiv: 1307.3696 [cs.NI].

[7] B. Briscoe, K. D. Schepper, M. Bagnulo, and G. White, "Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture," Internet Engineering Task Force, Internet-Draft draft-ietf-tsvwg-l4s-arch-10, Jul. 2021, Work in Progress, 42 pp. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-l4s-arch-10.

[8] S. Bommisetti, B. Annappa, and M. P. Tahiliani, "Extended ecn mechanism to mitigate ecn-based attacks," in *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, 2014, pp. 1105–1110. DOI: 10.1109/ICCICCT.2014.6993126.

[9] A. Laraba, J. François, I. Chrisment, S. R. Chowdhury, and R. Boutaba, "Defeating Protocol Abuse with P4: Application to Explicit Congestion Notification," in *2020 IFIP Networking Conference (Networking)*, Paris, France, Jun. 2020. [Online]. Available: https://hal.inria.fr/hal-02993199.

[10] A. Di Domenico, G. Perna, M. Trevisan, L. Vassio, and D. Giordano, "A network analysis on cloud gaming: Stadia, geforce now and psnow," *Network*, vol. 1, no. 3, pp. 247–260, 2021, ISSN: 2673-8732. DOI: 10.3390/network1030015. [Online]. Available: https://www.mdpi.com/2673-8732/1/3/15.

[11] M. Carrascosa and B. Bellalta, *Cloud-gaming:analysis of google stadia traffic*, 2020. arXiv: 2009.09786 [cs.NI].

[12] M. Suznjevic, I. Slivar, and L. Skorin-Kapov, "Analysis and qoe evaluation of cloud gaming service adaptation under different network conditions: The case of nvidia geforce now," in *2016 Eighth International Conference on Quality of Multimedia Experience (QoMEX)*, 2016, pp. 1–6. DOI: 10.1109/QoMEX.2016.7498968.

[13] S.-H. Lee, J.-S. Park, S.-H. Yoon, and M.-S. Kim, "High performance payload signature-based internet traffic classification system," in *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2015, pp. 491–494. DOI: 10.1109/APNOMS.2015.7275374.

[14] N. Shah. "The challenges of inspecting encrypted network traffic." (2020), [Online]. Available: https://www.fortinet.com/blog/industry-trends/keeping-up-with-performance-demands-of-encrypted-web-traffic (visited on 06/24/2022).

[15] P. Velan, M. Čermák, P. Čeleda, and M. Drašar, "A survey of methods for encrypted traffic classification and analysis," *Netw.*, vol. 25, no. 5, pp. 355–374, Sep. 2015, ISSN: 0028-3045.

[16] P.-O. Brissaud, J. François, I. Chrisment, T. Cholez, and O. Bettan, "Transparent and service-agnostic monitoring of encrypted web traffic," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 842–856, 2019. DOI: 10.1109/TNSM.2019.2933155.

[17] I. Akbari, M. A. Salahuddin, L. Ven, *et al.*, "A look behind the curtain: Traffic classification in an increasingly encrypted web," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 5, no. 1, 04:1–04:26, 2021. DOI: 10.1145/3447382. [Online]. Available: https://doi.org/10.1145/3447382.

[18] L. Yang, A. Finamore, F. Jun, and D. Rossi, "Deep learning and traffic classification: Lessons learned from a commercial-grade dataset with hundreds of encrypted and zero-day applications," *CoRR*, vol. abs/2104.03182, 2021. arXiv: 2104.03182. [Online]. Available: https://arxiv.org/abs/2104.03182.

[19] G. Wei, "Deep learning model under complex network and its application in traffic detection and analysis," in *2020 IEEE 2nd International Conference on Civil Aviation Safety and Information Technology (ICCASIT*, 2020, pp. 448–453. DOI: 10.1109/ICCASIT50869.2020.9368560.

[20] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *2017 International Conference on Information Networking (ICOIN)*, 2017, pp. 712–717. DOI: 10.1109/ICOIN.2017.7899588.

[21] T. Shapira and Y. Shavitt, "FlowPic: A generic representation for encrypted traffic classification and applications identification," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1218–1232, 2021. DOI: 10.1109/TNSM.2021.3071441.

[22] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, RFC 9000, May 2021. DOI: 10.17487/RFC9000. [Online]. Available: https://www.rfc-editor.org/info/rfc9000.

[23] S. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 660–674, 1991. DOI: 10.1109/21.97458.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] P. Bosshart, D. Daly, G. Gibb, *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. [Online]. Available: https://doi.org/10.1145/2656877.2656890.

[26] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, ser. HotNets '19, Princeton, NJ, USA: Association for Computing Machinery, 2019, pp. 25–33, ISBN: 9781450370202. DOI: 10.1145/3365609.3365864. [Online]. Available: https://doi.org/10.1145/3365609.3365864.