Request: Real-Time QoE Detection for Encrypted YouTube Traffic

Craig Gutterman†, Katherine Guo‡, Sarthak Arora†, Xiaoyang Wang‡,
Les Wu‡, Ethan Katz-Bassett†, Gil Zussman†
†Electrical Engineering, Columbia University, ‡Nokia Bell Labs

ABSTRACT

As video traffic dominates the Internet, it is important for operators to detect video Quality of Experience (QoE) in order to ensure adequate support for video traffic. With wide deployment of end-to-end encryption, traditional deep packet inspection based traffic monitoring approaches are becoming ineffective. This poses a challenge for network operators to monitor user QoE and improve upon their experience. To resolve this issue, we develop and present a system for **RE**al-time **QU**ality of experience metric detection for Encrypted Traffic, **Request**. *Request* uses a detection algorithm we develop to identify video and audio chunks from the IP headers of encrypted traffic. Features extracted from the chunk statistics are used as input to a Machine Learning (ML) algorithm to predict QoE metrics, specifically, *buffer warning* (low buffer, high buffer), *video state* (buffer increase, buffer decay, steady, stall), and *video resolution*. We collect a large YouTube dataset consisting of diverse video assets delivered over various WiFi network conditions to evaluate the performance. We compare *Request* with a baseline system based on previous work and show that *Request* outperforms the baseline system in accuracy of predicting buffer low warning, video state, and video resolution by 1.12×, 1.53×, and 3.14×, respectively.

CCS CONCEPTS

• **Information systems** → **Multimedia streaming**; • **Networks** → *Network performance analysis*; • **Computing methodologies** → *Classification and regression trees*;

KEYWORDS

Machine Learning, HTTP Adaptive Streaming

ACM Reference Format:
Craig Gutterman, Katherine Guo, Sarthak Arora, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, Gil Zussman. 2019. Request: Real-Time QoE Detection for Encrypted YouTube Traffic. In *10th ACM Multimedia Systems Conference (MMSys '19), June 18–21, 2019, Amherst, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3304109.3306226

1 INTRODUCTION

Video has monopolized Internet traffic in recent years. Specifically, the portion of video over mobile data traffic is expected to increase from 60% in 2016 to 78% by 2021 [2]. Content providers, Content Delivery Networks (CDNs), and network *operators* are all stakeholders in the Internet video sector. They want to monitor user video QoE and improve upon it in order to ensure user engagement. Content providers and CDNs can measure client QoE metrics, such as video resolution by using server-side logs [8, 20]. Client-side measurement applications can accurately report QoE metrics such as player events and video quality levels [33, 44].

Traditionally, Deep Packet Inspection (DPI) enabled operators to examine HTTP packet flows and extract video session information to infer QoE metrics [7, 11]. However, to address security and privacy concerns, content providers are increasingly adapting end-to-end encryption. A majority of YouTube traffic has been encrypted since 2016 [4] with a combination of HTTPS [9, 17, 36] and QUIC [14, 23]. Similarly, since 2015 Netflix has been deploying HTTPS for video traffic [10]. In general, the share of encrypted traffic is projected to grow to over 75% by 2019 [5].

Although the trend of end-to-end encryption does not affect client-side or server-side QoE monitoring, it renders traditional DPI-based video QoE monitoring ineffective for operators. Encrypted traffic still allows for viewing packet headers in plain text. This has led to recent efforts to use Machine Learning (ML) and statistical analysis to derive QoE metrics for operators. These works are limited as they either provide offline analysis for the entire video session [15, 34] or online analysis using both network and transport layer information with separate models for HTTPS and QUIC [32].

Previous research developed methods to derive network layer features from IP headers by capturing packet behavior in both directions: *uplink* (from the client to the server) and *downlink* (from the server to the client) [24, 32, 34]. However, determining QoE purely based on IP header information is inaccurate. To illustrate, Fig. 1 shows a 20-sec portion from two example sessions from our YouTube dataset, described in §4, where each data point is over 100 ms. Both examples exhibit similar traffic patterns in both directions. However, Fig. 1(a) shows a 720p resolution with the buffer decreasing by 15 secs, whereas Fig 1(b) shows a 144p resolution with the buffer increasing by 20 secs.

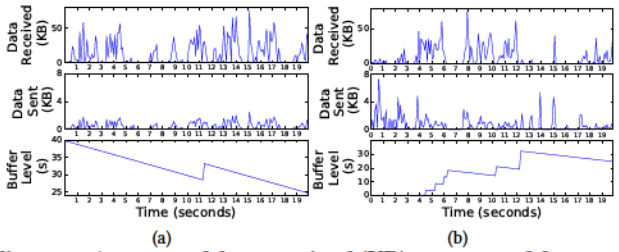

Figure 1: Amount of data received (KB), amount of data sent (KB), and buffer level (sec) for two sessions over a 20sec window (100 ms granularity): (a) 720p, (b) 144p.

MMSys '19, June 18–21, 2019, Amherst, MA, USA

ACM ISBN 978-1-4503-6297-9/19/06…$15.00
https://doi.org/10.1145/3304109.3306226

Given this challenge, our objective is to design features from IP header information that utilize patterns in the video streaming algorithm. In general, video clips stored on the server are divided into a number of *segments* or *chunks* at multiple resolutions. The client requests chunks from the server using HTTP GET requests. Existing work using chunks either infers QoE for the entire session [28] rather than in real-time, or lacks insight on chunk detection mechanisms from network or transport layer data [15, 26, 38].

To improve on existing approaches that use chunks, we develop **Requet**, a system for **RE**al-time **QU**ality of experience metric detection for Encrypted **T**raffic designed for traffic monitoring in middleboxes by operators. *Requet* is designed to be memory efficient for middleboxes, where memory requirement is a key consideration. Fig. 2 depicts the system diagram for *Requet* and necessary components to train the QoE models as well as evaluate its performance. *Requet* consists of the ChunkDetection algorithm, chunk feature extraction, and ML QoE prediction models. The data acquisition process involves collecting YouTube traffic traces (*Trace Collection*) and generating ground truth QoE metrics as labels directly from the player (*Video Labeling*). Packet traces are fed into *Requet*'s ChunkDetection algorithm to determine audio and video chunks. The chunks are then used during the *Feature Extraction* process to obtain chunk-based features. The chunk based features from the training data along with the corresponding QoE metrics are used to generate QoE prediction models. For evaluation, traffic traces from the testing dataset are fed into the trained QoE models to generate predicted QoE metrics. Accuracy is measured comparing the predicted QoE metrics and the ground truth labels.

Recent studies have shown that (i) stall events have the largest negative impact on end user engagement and (ii) higher average video *playback bitrate* improves user engagement [8, 16]. Motivated by these findings, *Requet* aims to predict events that lead to QoE impairment *ahead of time* and the current video resolution. This allows operators to proactively provision resources [12, 35]. *Requet* predicts low buffer level which allows operators to provision network resources to avoid stall events [24]. *Requet* predicts four video states: buffer increase, buffer decay, steady, and stall. Furthermore, *Requet* predicts current video resolution during a video session in real-time. Specifically, *Requet* predicts video resolution on a more granular scale (144p, 240p,360p, 480p, 720p, 1080p), while previous work predicts only two or three levels of video resolution for the entire video session [15, 28, 32].

We make the following contributions:

- Collect packet traces of 60 diverse YouTube video clips resulting in a mixture of HTTP/TLS/TCP and HTTP/QUIC/UDP traffic over WiFi networks from three operators, one in a different country. This is in contrast to prior works which rely on simulation or emulation [24, 32, 40] (see §4).
- Design *Requet* components
  - Develop ChunkDetection heuristic algorithm to identify video and audio chunks from IP headers (see §3).
  - Analyze the correlation between audio and video chunk metrics (e.g., chunk size, duration, and download time) to various QoE metrics, and determine fundamental *chunk-based* features useful for QoE prediction. Specifically, design features based on our observation that audio chunk arrival rate correlates with the video state (see §5).

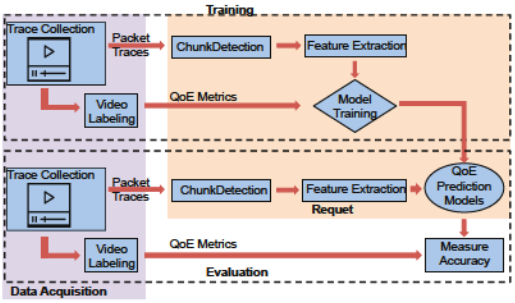

Figure 2: System Diagram: Data acquisition and *Requet* components: ChunkDetection, feature extraction, and QoE prediction models.

  - Develop ML models to predict QoE in real-time: buffer warning, video state, and video resolution (see §6).
- Evaluate *Requet* performance
  - Demonstrate drastically improved prediction accuracy using chunk-based features versus baseline IP layer features commonly used in prior work [24, 32, 34, 41]. For predicting low buffer warning, video state, fine grained video resolution, *Requet* achieves 92%, 84% and 66% accuracy, representing an improvement of 1.12×, 1.53×, and 3.14× respectively, over existing baseline. *Requet* delivers a 91% accuracy in predicting low (144p/240p/360p) or high resolution (480p/720p/1080p) (see §6).
  - Demonstrate that *Requet* trained in a lab environment works on unseen clips with varying lengths from different operators in multiple countries. This evaluation is more diverse than prior work [15, 24, 32, 40] (see §6).

2 BACKGROUND & PROBLEM STATEMENT

2.1 Adaptive BitRate Streaming Operation

A majority of video traffic over the Internet today is delivered using Adaptive BitRate (ABR) streaming with its dominating format being Dynamic Adaptive Streaming over HTTP (DASH) or MPEG-DASH [39, 45]. In ABR, a video *asset* or *clip* is encoded in multiple resolutions. A clip with a given resolution is then divided into a number of *segments* or *chunks* of variable length, a few seconds in playback time [30]. Typically video clips are encoded with Variable Bitrate (VBR) encoding and are restricted by a maximum bitrate for each resolution. An audio file or the audio track of a clip is usually encoded with Constant Bitrate (CBR). For example some of the YouTube audio bitrates are 64, 128, 192 Kbps [43].

At the start of the session, the client retrieves a manifest file which describes the location of chunks within the file containing the clip encoded with a given resolution. There are many ABR variations across and even within video providers [30]. ABR is delivered over HTTP(S) which requires either TCP or any other reliable transport [18]. The ABR algorithm can use concurrent TCP or QUIC/UDP flows to deliver multiple chunks simultaneously. A chunk can be either video or audio alone or a mixture of both.

2.2 Video States and Playback Regions

The client employs a *playout buffer* or *client buffer*, whose maximum value is *buffer capacity*, to temporarily store chunks to absorb network variation. To ensure smooth playback and adequate *buffer*

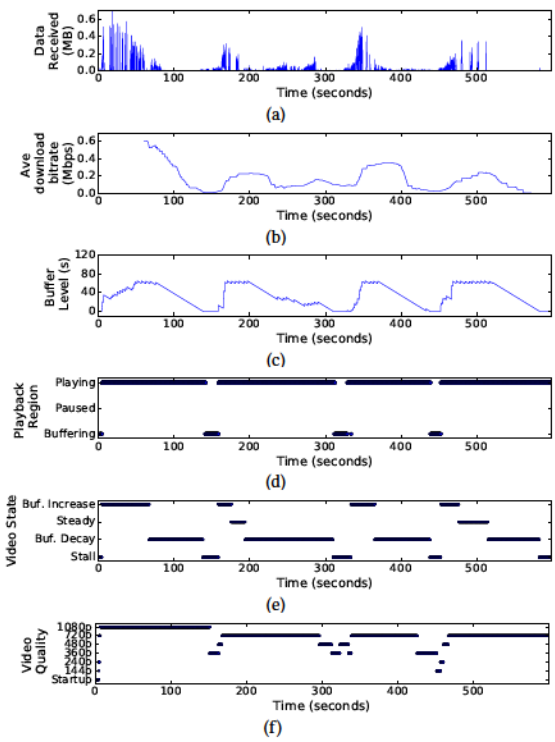

Figure 3: Behavior of a 10-min session in 100-ms windows: (a) amount of data received (MB), (b) average download bitrate (Mbps) over the past 60 sec, (c) buffer level, (d) playback region, (e) video state, (f) video resolution.

level the client requests a video clip chunk by chunk using HTTP GET requests, and dynamically determines the resolution of the next chunk based on network condition and buffer status.[1]

When buffer level is below a low threshold, the client requests chunks as fast as the network can deliver them to increase buffer level. We call this portion of ABR operation the *buffer filling* stage. In this stage, buffer level can increase or decrease. Once buffer level reaches a high threshold, the client aims to maintain buffer level in the range between the threshold and buffer capacity. One example of a client strategy is to request chunks as fast as they are consumed by the playback process, which is indicated by the video *playback bitrate* for the chunk [41]. We call this portion the *steady state* stage. The playback *stalls* when the buffer is empty before the end of the playback is reached. After all chunks have been downloaded to the client buffer, there is no additional traffic and the buffer level decreases. From the perspective of buffer level, an ABR session can experience four exclusive *video states*: *buffer increase*, *buffer decay*, *steady state*, and *stall*.

Orthogonally, from the perspective of YouTube video playback, a session can contain three exclusive *regions*: *buffering*, *playing*, and *paused*. Buffering region is defined as the period when the client is receiving data in its buffer, but video playback has not started or is stopped. Playing region is defined as the period when video playback is advancing regardless of buffer status. Paused region is defined as the period when the end user issues the command to pause video playback before the session ends. In playing region, video state can be in either buffer increase, decay, or steady state.

Fig. 3 shows the behavior of a 10-min session from our dataset in §4 in each 100-ms window with (a) the amount of data received (MB), (b) download throughput (Mbps) for the past 60 sec, (c) buffer level (sec), (d) occurrence of three playback regions, (e) occurrence of four video states, and (f) video resolution. At the start of the session and after each of the three stall events, notice that video resolution slowly increases before settling at a maximum level.

2.3 QoE Metrics and Prediction Challenges

This subsection describes the QoE metrics that we reference and the challenges in predicting these metrics. We focus on metrics that the operator can use to infer user QoE impairments in real-time. Specifically, we use three QoE metrics: *buffer warning*, *video state* and *video quality*. We do not focus on start up delay prediction, as it has been extensively studied in [24, 28, 32].

The first QoE metric we aim to predict is the current *video state*. The four options for video state are: buffer increase, buffer decay, stall, or steady state. This metric allows for determining when the video level is in the ideal situation of steady state. It also recognizes when the buffering is decreasing or stalling and the operator should allocate more resources toward this user.

The *buffer warning* metric is a binary classifier for determining if the buffer level is below a certain threshold $BufWarning_{\text{thresh}}$ (e.g., under 20 sec). This enables operators to provision resources in real-time to avoid potential stall events before they occur. For example, at a base station or WiFi AP, ABR traffic with buffer warning can be prioritized.

Another metric used is the current *video resolution*. Video encoders consider both resolution and target video bitrate. Therefore, it is possible to associate a lower bitrate with a higher resolution. One can argue bitrate is a more accurate indicator of video quality. However, higher resolutions for a given clip often result in higher bitrate values. YouTube client API reports in real-time resolution rather than *playback bitrate*. Therefore, we use resolution as an indicator of video quality.

ABR allows the client to dynamically change resolution during a session. Frequent changes in resolution during a session tend to discourage user engagement. Real-time resolution prediction enables detection of resolution changes in a session. However, this prediction is challenging as *download bitrate* to video resolution does not follow a 1-to-1 mapping. In addition, a video chunk received by the client can either replace a previous chunk or be played at any point in the future. Under the assumption that playback typically begins shortly (in the order of seconds) after the user requests a clip, one can associate the average download bitrate with video quality, since higher quality requires higher bitrate for the same clip. However, this is not true in a small time scale necessary for real-time prediction. Network traffic reveals the combined effect of buffer trend (increase or decay) and video *playback bitrate* which correlates to resolution. During steady state, video's *download bitrate* is consistent with playback bitrate. However, when a client is in non-steady state, one cannot easily differentiate between the case in which a higher resolution portion is retrieved during buffer

[1]The field of ABR client algorithm design is an active research area [22, 31].

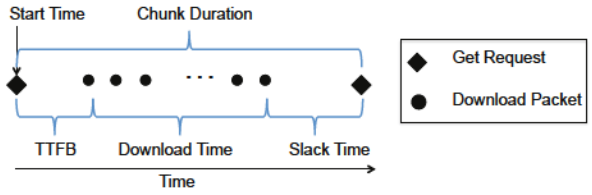

Figure 4: Definition of chunk metrics (video or audio)

decay state (Fig. 1(a)), and the case in which a lower resolution portion is retrieved during buffer increase state (Fig. 1(b)). Both of these examples exhibit similar traffic patterns, however the behavior of QoE metrics is dramatically different.

3 CHUNK DETECTION

The fundamental delivery unit of ABR is a chunk [25]. Therefore, identifying chunks instead of relying on individual packet data can capture important player events. Specifically, the occurrence of a chunk indicates that the client has received a complete segment of video or audio, resulting in increased buffer level in playback time. An essential component of *Requet* in Fig. 2 is its ChunkDetection algorithm to identify chunks from encrypted traffic traces. Features are extracted from the chunks and used as the input to the ML QoE prediction models. Existing work using chunks either lacks insight in chunk detection mechanisms [15, 26, 38] or studies per-session QoE [28] instead of predicting QoE in real-time.

In this section, we first describe metrics capturing chunk behavior. We then develop ChunkDetection, a heuristic algorithm using chunk metrics to identify individual audio and video chunks from IP level traces. *Requet* uses ChunkDetection to detect chunks from multiple servers simultaneously regardless of the use of encryption or transport protocol. It relies purely on source/destination IP address, port, protocol, and payload size from the IP header.

3.1 Chunk Metrics

We define the following metrics for a chunk based on the timestamp of events recorded on the end device (as shown in Fig. 4).

- **Start_Time** - The timestamp of sending the HTTP GET request for the chunk.
- **TTFB** - Time To First Byte, defined as the time duration between sending an HTTP GET request and the first packet received after the request.
- **Download_Time** - The time duration between the first received packet and the last received packet prior to the next HTTP GET request.
- **Slack_Time** - The time duration between the last received packet and the next HTTP GET request.
- **Chunk_Duration** - The time duration between two consecutive HTTP GET requests. The end of the last chunk in a flow is marked by the end of the flow. Note that a different concept called "segment duration" is defined in standards as playback duration of the segment [6]. For a given chunk, Chunk_Duration equals "segment duration" only during steady state.
- **Chunk_Size** - The amount of received data (sum of IP packet payload size) during Download_Time from the IP address that is the destination of the HTTP GET request marking the start of the chunk.

Algorithm 1 Audio Video Chunk Detection Algorithm

```
procedure ChunkDetection
    Initialize Audio and Video for each IP flow I
    for each uplink packet p with IP flow I do
        if uplink(p) and (packetlength(p) > GET_thresh then)
            c ← [GetTimestamp, GetSize, DownStart,
            DownEnd, GetProtocol, I]
            AVflag ← DetectAV(c)
            if AVflag == 0 then
                Append c to Audio
            else if AVflag == 1 then
                Append c to Video
            else
                Drop c
            GetTimestamp ← time(p)
            GetSize ← packetlength(p)
            DownFlag ← 0
        if downlink(p) and (packetlength(p) > Down_thresh then)
            if DownFlag == 0 then
                DownFlag = 1
                DownStart ← time(p)
            DownEnd ← time(p)
            DownSize+ = packetlength(p)
```

Table 1: Chunk Notation

Symbol	Semantics
GET_{thresh}	pkt length threshold for request (300 B)
$Down_{\text{thresh}}$	pkt length threshold for downlink data (300 B)
$GetTimestamp$	timestamp of GET request
$GetSize$	pkt length of GET request
$DownStart$	timestamp of first downlink packet of a chunk
$DownEnd$	timestamp of last downlink packet of a chunk
$GetProtocol$	IP header protocol field
DetectAV	sorts chunk into audio chunk, video chunk or no chunk based on $GetSize$, $DownSize$, $GetProtocol$
$\vec{Audio}$	audio chunks for an IP flow
$\vec{Video}$	video chunks for an IP flow

Note, for any chunk, the following equation holds true: Chunk_Duration = sum(TTFB, Download_Time, Slack_Time).

3.2 Chunk Detection Algorithm

We explore characteristics of YouTube audio and video chunks. Using the web debugging proxy Fiddler [3], we discover that audio and video are transmitted in separate chunks, and they do not overlap in time for either HTTPS or QUIC. For both protocols we notice at most one video or audio chunk is being downloaded at any given time. Each HTTP GET request is carried in one IP packet with IP payload size above 300 B. Smaller uplink packets are HTTP POST requests regarding YouTube log events, or TCP ACKs. Fig. 5 and Fig. 6 plot the HTTP GET request size and subsequent audio/video chunk size in a high (1080p) and a low (144p) resolution session, respectively. It is evident that HTTP GET request size for audio chunks is slightly smaller than that for video chunks (Fig. 5(b),

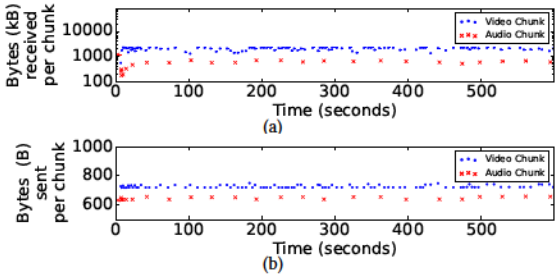

Figure 5: Individual video/audio chunks in a 10-min session with highest resolution (V:1080p, A:160kbps). (a) Chunk Size, (b) Get Request Size.

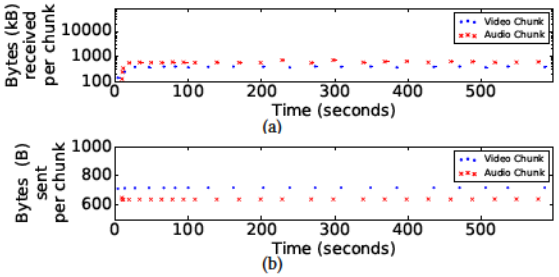

Figure 6: Individual video/audio chunks in a 10-min session with lowest resolution (V:144p, A:70kbps). (a) Chunk Size, (b) Get Request Size.

Fig. 6(b)). This difference is due to the additional fields used in HTTP GET requests for video content that do not exist for audio content. Furthermore, at higher resolution levels, video chunk size is consistently larger than audio chunk size (Fig. 5(a)). However, at lower resolution levels, video chunk size can be similar to or even smaller than audio chunk size (Fig. 6(a)). We can conservatively set the low threshold for chunk size to be 80 KB for our dataset.

Based on the above observations, we propose a heuristic chunk detection algorithm in Alg. 1 using notations in Table 1. ChunkDetection begins by initializing each IP flow with empty arrays for both audio and video chunks. This allows for the chunk detection algorithm to collect chunks from more than one server at a time.

ChunkDetection initially recognizes any uplink packet with a payload size above 300 B as an HTTP GET request (line 4). This threshold may vary depending on the content provider. For YouTube, we note that GET requests over TCP are roughly 1300 bytes, while GET requests over UDP are roughly 700 bytes. For each new GET request the *GetTimestamp*, and *GetSize*, are recorded (lines 14-16). After detecting a GET request in an IP flow, chunk size is calculated by summing up payload size of all downlink packets in the flow until the next GET is detected (lines 17-22). The last downlink packet in the group between two consecutive GET requests marks the end of a chunk download. The chunk download time then becomes the difference in timestamp between the last and first downlink packet in the group. [2]

Once the next GET is detected, ChunkDetection records *GetTimestamp*, *GetSize*, download start time *DownStart*, download end time *DownEnd*, the protocol used *GetProtocol* and the IP flow *I* of the previous chunk (line 5). This allows for the calculation of chunk duration and slack time using the timestamp of the next GET. GET request size and chunk size are used in DetectAV (line 7) to separate data chunks into audio chunks, video chunks, or background traffic (lines 8-11). DetectAV uses the heuristic that HTTP GET request size for audio chunks is slightly smaller than request size for video chunks consistently. Furthermore, if download size is too small (< 80 KB), DetectAv recognizes that the data is neither an audio or video chunk, and the data is dropped (lines 12-13). This allows ChunkDetection to ignore background YouTube traffic.

[2]ChunkDetection does not flag TCP retransmission packets, therefore can overestimate chunk size when retransmission happens. ChunkDetection also assumes chunks do not overlap in time in a given IP flow. If it happens, the individual chunk size can be inaccurate, but the average chunk size over the period with overlapping chunks is still accurate.

4 DATA ACQUISITION

Requet, shown in Fig. 2, is designed to process traffic traces and QoE metrics as ground truth labels for the traces. Data acquisition provides data for training and evaluation for *Requet* QoE prediction models. This includes traffic trace collection, and deriving QoE metrics as ground truth labels associated with traffic traces.

4.1 Trace Collection

We design and implement a testbed (shown in Fig. 7) to capture a diverse range of YouTube behavior over WiFi. We watch YouTube video clips using the Google Chrome browser on a Macbook Air laptop. We connect the laptop to the Internet via an Access Point (AP) using IEEE 802.11n. A shell script simultaneously runs Wireshark's Command Line Interface, Tshark [1], and a Javascript Node server hosting the YouTube API.

The YouTube IFrame API environment collects information displayed in the "Stats for Nerds" window. From this API we monitor: video playback region ('Playing', 'Paused', 'Buffering'), playback time since the beginning of the clip, amount of video that is loaded, and current video resolution. From these values we determine the time duration of the portion of the video clip remaining in the buffer. We collect information once every 100 ms as well as during any change event indicating changes in video playback region or video resolution. This allows us to record any event as it occurs and to keep detailed information about playback progress.

We have two options to collect network level packet traces in our setup, on the end device or on the WiFi AP. Collecting traces at the AP would limit the test environment only to a lab setup. Therefore, we opt to collect traces via Wireshark residing on the end device. This ensures that the YouTube client data is synchronized with Wireshark traces and the data can be collected on public and private WiFi networks. Our traces record packet timestamp, size, as well as the 5-tuple for IP-header (source IP, destination IP, source port, destination port, protocol). Our dataset contains delivery over HTTPS (9% GET requests) and QUIC (91% GET requests). We do not use any transport level information. In addition, we record all data associated with a Google IP address. The IP header capture allows us to calculate total number of packets and bytes sent and received by the client in each IP flow during a given time window.

To generate traces under varying network conditions, we run two categories of experiments: *static* and *movement*. For static cases, we place the end device in a fixed location for the entire session. However, the distance from the AP varies up to 70 feet or multiple

Figure 7: Experimental setup for our trace collection.

rooms away. For movement cases, we walk around the corridor (up to 100 feet) in our office building with the end device, while its only network connection is through the same AP.

We select 60 YouTube clips representing a wide variety of content types and clip lengths. Each clip is available in all 6 common resolutions from YouTube, namely 144p, 240p, 360p, 480p, 720p and 1080p. We categorize them into four groups, where groups *A* and *B* are medium length clips (8 to 12 min), *C* are short clips (3 to 5 min), and *D* are long clips (25-120 min). Table 2 lists the number of unique clips in the group, along with the length of each clip and the session length, that is, the duration for which we record the clip from its start.

For group *A*, we collect 425 sessions in both static (over 300) and movement cases (over 100) in a lab environment in our office building. All remaining experiments are conducted in static cases. For clips in group *B*, we collect traces in an apartment setting in the US (set B_1 with 60 sessions) and in India (set B_2 with 45 sessions) reflecting different WiFi environments. We collect traces in set *C* and *D* from the lab environment, where each set contains more than 25 sessions. Overall, there are over 10 sessions for each clip in group *A* and *B* and 6 sessions for each clip in group *C* and *D*.

Clips in both groups *A* and *B* range from 8 to 12 min in length. In each session we play a clip and collect a 10-min trace from the moment the client sends the initial request. We choose this range of length in order for the client to experience buffer increase, decay and steady state. Shorter clips with a length close to buffer capacity (e.g., 2 min) can sometimes never enter steady state, even when given abundant network bandwidth. In general, when there is sufficient bandwidth to support the clip's requirement, a clip can be delivered in its entirety before the end of the playback happens. On the contrary, when available network bandwidth is not enough to support the clip's requirement, a clip may experience delayed startup and even stall events.

We collect traces over 6 months from Jan. through June 2018, with video resolution selection set to *"auto"*. This means the YouTube client is automatically selecting video resolution based on changes in network condition. For each session, we set an initial resolution to ensure that all resolution levels have enough data points.

Each group includes a diverse set of clips in terms of activity level. It ranges from low activity types such as lectures to high activity types such as action sequences. This fact can be seen in the wide range of video bitrates for any given resolution. Fig. 8 shows the average playback bitrate for each video resolution for each clip in our dataset. All clips are shown in scatter plots, while clips in group *A* are also shown with box plots.[3] One can see that the average video playback bitrate spans overlapping regions. Therefore, this cannot provide a perfect indication of the video resolution even if the entire session is delivered with a fixed resolution.

[3]For all box plots in the paper, the middle line is the median value. The bottom and top line of the box represents Q1 (25-percentile) and Q3 (75-percentile) of the dataset respectively. The lower extended line represents $Q1 - 1.5IQR$, where IQR is the inner quartile range (Q3-Q1). The higher extended line represents $Q3 + 1.5IQR$.

Table 2: Clip distribution in our dataset.

Group	Clip Length	Session Length	No. of Unique Clips
A	8 – 12 min	10 min	40
B	8 – 12 min	10 min	10
C	3 – 5 min	5 min	5
D	25 – 120 min	30 min	5

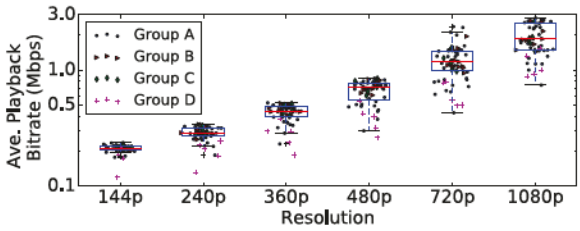

Figure 8: Average playback bitrate vs. video resolution for clips in our dataset. Clips in all four groups are shown in scatter plots, while clips in group *A* are also shown with box plots.

In our dataset, we notice that YouTube buffer capacity varies based on video resolution. For example, it is roughly 60, and 120 sec for 1080p and 144p, respectively.

We collect data for each YouTube video session in the Chrome browser as the sole application on the end device. We record all packets between the client and any Google servers. The client contacts roughly 15 to 25 different Google servers per session. We examine the download throughput (see Fig. 3(a) and 3(b) for example) further by looking at the most commonly accessed server IP addresses for each session sorted by the total bytes received. During a session a majority of traffic volume comes from a single to a few servers.

4.2 Video State Labeling

A goal for predicting video QoE in real-time inside the network is to enable real-time resource provisioning to prevent stalls and decreases in video resolution. To enable this prediction, accurate labeling of video state is critical. The four exclusive video states (buffer increase, decay, stall and steady state) accurately capture the variations in buffer level. They can be used in combination with actual buffer level to predict dangerous portions of ABR operation that may lead to QoE degradation. For example, when the buffer level is close to 0, a stall event is likely to happen in the near future. Increasing network capacity for the session may prevent a stall.

As shown in §2, playback regions reported by the client ignore buffer level changes, and cannot be used to generate video states. Prior work uses manual examination which is time consuming and can be inaccurate [41]. We opt to automate the process by developing the definition of video states based on buffer level variation over time followed by our video state labeling algorithm. We define the four video states as follows:

(1) **Buffer Increase:** Buffer level is increasing. It has a slope greater than ϵ per sec over time window T_{slope}.
(2) **Steady State:** Buffer level is relatively flat. The slope of buffer level is between $-\epsilon$ and $+\epsilon$ $\frac{sec}{sec}$ over time window

Algorithm 2 Video State Labeling Algorithm

```
procedure VideoStateLabeling
    Initialize δ, ϵ, T_smooth, T_slope
    for every t do
        Calculate B̂_t ← median[B_{t−T_smooth}, ..., B_{t+T_smooth}]
        Calculate m_t ← (B̂_{t+T_slope} − B̂_{t−T_slope}) / 2T_slope
        if B_t ≤ δ then
            State_t ← Stall
        else if −ϵ ≤ m_t ≤ ϵ and B_t > Buff_SS then
            State_t = Steady State
        else if m_t < 0 then
            State_t ← Buffer Decay
        else
            State_t ← Buffer Increase
    SmoothState(State)
```

Table 3: Notation Summary

Symbol	Semantics	Defaults
δ	Stall threshold	0.08 sec
ϵ	Buffer slope boundary for Steady State	0.15 $\frac{sec}{sec}$
T_{smooth}	Time window for smoothing buffer	15 sec
T_{slope}	Time window to determine buffer slope	5 sec
$Buff_{\text{SS}}$	Minimum buffer level to be in steady state	10 sec
Thr_{SS}	Minimum time window to stay in steady state	15 sec
$MinTime_{\text{SS}}$	Time window to look for quick changes out of steady state	10 sec
$MinTime_{\text{stall}}$	Time window to look for quick changes out of stall state	10 sec

T_{slope}. To be in steady state the slope needs to be in this range for greater than Thr_{SS} sec.

(3) **Buffer Decay:** Buffer level is decreasing with a slope less than $-\epsilon$ $\frac{sec}{sec}$ over time window T_{slope}.

(4) **Stall:** Buffer level is less than or equal to δ.

We execute our video state labeling algorithm in Alg. 2 for each time instance t when buffer information is recorded (every 100 ms) to determine video state for a session according to our definition.

As a chunk arrives at the client, buffer level increases by chunk length in sec. During playback, buffer level decreases by 1 sec for every sec of playback. Looking at short windows or the wrong point of a window would incorrectly determine that buffer is decreasing. We use a smoothing function to derive a more accurate buffer slope. Specifically, we use a moving median filter over a window around t defined by $[t - T_{\text{smoooth}}, t + T_{\text{smooth}}]$. We examine the rate of change of the buffer slope over a window around t defined by $[t - T_{\text{slope}}, t + T_{\text{slope}}]$.

In order to avoid rapid change of stall state, we set δ to 0.08 sec. This value ensures that small variations in and out of stall state are consistently labeled as being in stall state. If the buffer level is above $Buff_{\text{SS}}$ and has a slope between $-\epsilon$ and ϵ $\frac{sec}{sec}$, then we label it as steady state. If these specifications are not met and the slope is negative, we set the state to buffer decay. If the slope is positive, we set the state to buffer increase.

Table 4: % of chunks in each state (Set *A*).

Resolution	Video State			
	Stall	Decay	Steady	Increase
Audio	1.2	2.8	40.9	55.1
Video	3.7	5.9	47.6	42.8

To ensure that video state does not change rapidly due to small fluctuations of buffer level, we use an additional heuristic of *SmoothState*: steady state has to last longer than Thr_{SS}. This allows chunks with playback time longer than this value to arrive at the client. If there are changes out of and then back into stall state that last less than $MinTime_{\text{stall}}$ we consider the entire period as stall. Similarly, if there are changes out of and then back into steady state that last less than $MinTime_{\text{stall}}$ we consider the entire period steady state. For clarity, we list all symbols in Table 3, as well as the values that we find to work the best empirically for our dataset.

5 REQUET ML FEATURE DESIGN

We develop the ML *QoE prediction models* for *Requet* by using packet traces and associated ground truth labels (§4). As shown in Fig. 2, the traces are converted into chunks by using ChunkDetection (§3), and then the associated features are extracted.

We develop ML models using Random Forest (RF) to predict user QoE metrics[21]. We build the RF classifier in Python using the sklearn package. We choose RF for a number of reasons. (i) ML classification algorithms based on decision trees have shown better results in similar problems [15, 32, 34, 41] with RF showing the best performance among the class [41]. (ii) On our dataset, Feedforward Neural Network and RF result in roughly equal accuracy. (iii) RF can be implemented with simple rules for classification in real-time, well suited for real-time resource provisioning in middleboxes.

Each session in our dataset consists of (i) IP header trace and (ii) QoE metric ground truth labels generated by our video labeling process in data aqcuisition (§4). ChunkDetection (§3.2) of *Requet* transforms the IP header trace into a sequence of chunks along with the associated chunk metrics (§3.1). The goal of *Requet* QoE models is to predict QoE metrics using chunk metrics. To train such ML models, it is critical to capture chunk behavior associated with QoE metrics using chunk-based features. We analyze chunk behavior in our dataset (§5.1), explore how to capture such behavior in chunk-based features (§5.2), and explain how to generate baseline features used in prior work that are oblivious to chunk information (§5.3).

5.1 Chunk Analysis

We apply the ChunkDetection algorithm (Alg. 1) of *Requet* to all sessions from the 40 clips in set *A* in our dataset. We examine the correlation between various chunk metrics (audio or video, chunk size, chunk duration, *effective rate* which we define as chunk size over chunk duration, TTFB, download time, and slack time) to QoE metrics (buffer level, video state, and resolution). In most cases of our dataset, for a given session, audio and video chunks are transmitted from one server. However, in some cases audio and video traffic comes from different servers. In other cases, the

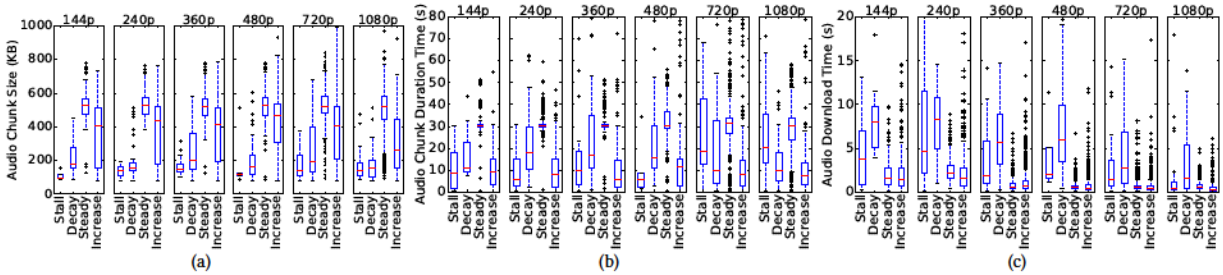

Figure 9: Chunk metrics for all audio chunks in set *A*. (a) chunk size, (b) chunk duration, (c) download time.

server switches during a session. These findings are consistent with existing YouTube traffic studies [33].

We list the distribution of audio and video chunks along with video state at the end of chunk download in Table 4. Most of the chunks occur during steady or buffer increase states. An extremely small fraction (4% audio and 9% video) are associated with stall or buffer decay states. They represent two scenarios: (i) bandwidth is limited and there are not enough chunks to increase buffer level substantially or (ii) buffer is about to transition into increase state.

Fig. 9 and 10 show the box plots for chunk duration, size, and download time for audio and video chunks respecitvely. Each plus sign represents an outlier. TTFB reflects the round trip time from the client to the server, and has a median value of 0.05 sec. This accounts for a tiny portion of chunk duration (median value ≥ 5 sec). We can safely simplify the relationship between various chunk metrics to (slack time = chunk duration - download time). Notice that slack time and effective rate are derivable from chunk duration, size, and download time. The latter three are the key metrics used in our feature selection for ML models.

Audio is encoded with CBR, however our examination of HTTP requests using Fiddler [3] reveal that in the four video states (steady, buffer increase, decay and stall), audio chunk size decreases in the same order. This implies that audio chunk playback time also decreases in the same order. This behavior is consistent across all resolution levels (Fig. 9(a)) and indicates that audio chunk size exhibits a strong correlation with video state. Across all resolution levels, Fig. 9(b) shows median audio chunk duration in steady and buffer increase state is roughly 30 and 10 sec respectively, but does not exhibit a clear pattern in stall and buffer decay states. Fig. 9(c) shows audio chunk download time in steady and buffer increase states are similar in value, both smaller than that of stall state, which is smaller than that of buffer decay state. The longer download time is an indication that the network bandwidth is limited. This is a useful insight that current bandwidth alone can not reveal. For example, a specific throughput can be associated to a low resolution with the buffer increasing or a higher resolution with the buffer decreasing. All three audio chunk metrics are clearly correlated with video state.

Fig. 10 shows video chunk statistics. There is a large overlap across different resolutions and video states in chunk size (Fig. 10(a)) and chunk duration (Fig. 10(b)). It reveals that without knowing video state, it would be difficult to determine video resolution, chunk size, and chunk duration. For example, these statistics are very similar for a 240p chunk in buffer increase state and a 720p chunk in buffer decay. Using audio chunk statistics to identify video state is critical in separating these two cases.

For video chunks, our examination of HTTP requests using Fiddler also shows that for a clip with a given resolution, steady state chunk size is larger than that in the remaining three states. Fig. 10(a) further shows that median video chunk size increases as resolution increases from 144p to 480p and stays roughly the same around 2 MB from 480p to 1080p. Fig. 10(b) shows median chunk duration in steady state is similar for 144p, 240p, and 360p, in the range of $35 - 45$ sec, and decreases from 25 sec for 480p to 5 sec for 1080p. To obtain a higher effective rate for higher resolutions the chunk size levels off, but to compensate chunk duration decreases. Fig. 10(c) shows median chunk download time exhibit smaller values in stall or buffer decay state, higher and similar values in steady or buffer increase state. This is expected as with limited bandwidth, a session may deplete its buffer or even stall. During buffer increase, retrieving smaller chunks faster than steady state results in similar download time as steady state. During steady and buffer increase state, chunk size and duration combined provide some indication of resolution levels. However, during stall and buffer decay state, no indication can be easily seen from the three metrics.

To summarize, our key observations are as follows: (i) Without knowing video state it would be difficult to differentiate between the two cases: (a) Higher resolution clip in buffer decay and (b) Lower resolution clip in buffer increase. (ii) Audio chunk statistics exhibit strong association with video state. (iii) Video chunk size increases and eventually levels off as resolution increases. At the same time, video chunk duration is higher for lower resolution levels and decreases as resolution level increases.

5.2 Chunk-based Features in *Requet*

Requet identifies chunks using Alg. 1 executed over all flows during a YouTube session. For each audio or video chunk, it records the following seven chunk metrics: protocol used to send the GET request, start time, TTFB, download time, slack time, chunk duration, and chunk size. Furthermore, it does not record the server IP address from which the chunk is delivered to the end device as it has no relationship with our QoE metrics.

Results from §5.1 show that the most important metrics for both audio and video are chunk size, duration, and download time. Chunk arrival is not a uniform process in time and therefore, the number of chunks in a time window vary. This would require a variable number of features. Instead, *Requet* uses statistics of chunk metrics in different time windows. Specifically, for the 20 windows representing the immediate past 10, 20, ..., 200 sec, it records total number of chunks, average chunk size and download time for each time window, resulting in 60 features each for audio and video, and

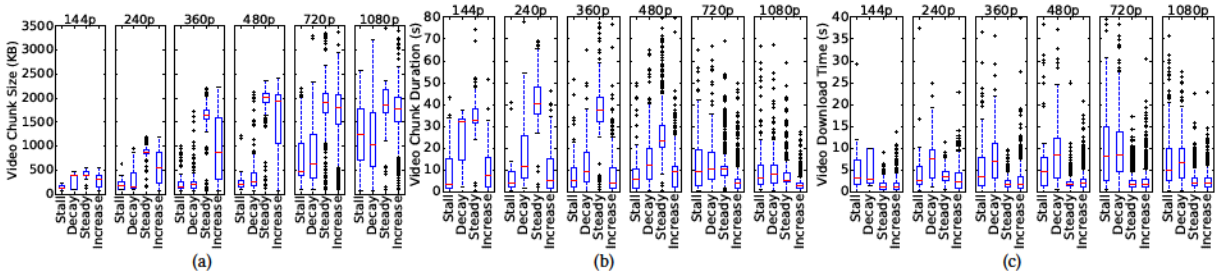

Figure 10: Chunk metrics for all video chunks in set *A*. (a) chunk size, (b) chunk duration, (c) download time.

a total of 120 features.[4] Regarding video resolution, *Requet* only makes predictions upon receiving a video chunk. Therefore, beyond the 120 features, it further includes the seven features associated with the video chunk. The sliding window based features in *Requet* make it ideal for middleboxes with a memory requirement of 1016 bytes for the 127 features (assuming each feature requires a maximum of 8 bytes).

5.3 Baseline Features

For the baseline system, we remove *Requet*'s ChunkDetection algorithm in Fig. 2 and the associated features and instead design baseline features commonly used in prior work [24, 32, 34, 41]. We select features that are used in more than one of these prior works, and feasible to use in network middleboxes where memory is constrained. We collect basic IP level features in terms of flow duration, direction, volume (total bytes), burstiness, as well as transport protocol. For each 100-ms window, we calculate the total number of uplink and downlink packets and bytes, and include a one-hot vector representation of the transport protocols used for each IP address.[5] The five features for transport protocol are QUIC, TCP with TLS, TCP without TLS, no packets in that interval, or other. After examining the total downlink bytes of the top 20 flows in a session in our dataset, we decide to include traffic from the top 3 servers in our feature set. The remaining flows have significantly smaller traffic volume and therefore represent background traffic in a session and do not deliver video or audio traffic. By doing so, we effectively eliminate the traffic that is unrelated to our QoE metrics. In addition, we include the total number of uplink/downlink bytes and packets from the top 20 servers for the session.

We calculate the average throughput and the total number of packets in the uplink and downlink direction during a set of time intervals to capture recent traffic behavior. Specifically, we use six intervals immediately proceeding the current prediction window, and they are of length 0.1, 1, 10, 60, 120, and 200 sec.

Furthermore, during these six windows, we record the percentage of 100-ms slots with any traffic in uplink and downlink separately. These two features are added to determine how bursty the traffic is during the given time window. In addition to the four features for the total network traffic for all servers contacted during the session, the features for each of the top three servers are:

- total bytes in past 100 ms in uplink/downlink
- total number of packets in past 100 ms in uplink/downlink
- transport protocol (5 features)
- for each of the windows of length 1, 10, 60, 120, and 200 sec:
  - average throughput in uplink/downlink
  - total number of packets in uplink/downlink
  - % of 100-ms slots without any traffic in uplink/downlink

To summarize, for each time window, there are up to $4 + 3 \times (4 + 5 + 5 \times 6) = 121$ features for the baseline system.

6 EVALUATION

We evaluate the performance of *Requet* by comparing its accuracy for each QoE metric versus the baseline system. Both systems predict the current QoE metrics every 5 sec, except for *Requet* which predicts resolution every chunk. Since transport payload of network traffic we collect is encrypted, we are unable to evaluate *Requet* against previous works that use deep packet inspection. Data collected as described in §4 is used for training, validation, and testing. Out of the four sets of traces in our dataset (§4.1), we use group *A*, the largest one to train both systems to predict each QoE metric in real-time. We then test *Requet* on smaller groups *B*, *C*, and *D*. Subsequently, to determine how training in the lab environment works on clips with similar length but in different environments we use groups B_1 and B_2. We also use group *A* as the training set for evaluating shorter clips (group *C*) and longer clips (group *D*) in the same lab environment as group *A*.

For group *A*, we conduct 4-fold cross validation on the 40 clips. Specifically, we divide the 40 clips into four exclusive sets each with ten unique clips. In each fold, we first train a model for each QoE metric using RF with features from 30 clips (three of the four sets). We then test the model on the ten clips from the remaining set. We report each model's average performance over the four folds.

The buffer warning model produces two prediction possibilities. It indicates whether the buffer level is below the threshold $BuffWarning_{\text{thresh}}$ or not. The video state model produces four states and the resolution model produces six resolution levels.

We report *accuracy* of each model as the ratio of the number of correct predictions over total number of predictions. For each label a model predicts, we further report: (i) *precision* defined as the ratio of true positives to total positives, that is, the percentage of correct predictions out of all positive predictions of a label, and (ii) *recall* defined as the ratio of correct predictions to total true occurrences of a label, that is, the percentage of a label correctly predicted.

[4] We use the past 200sec history as YouTube buffer rarely increases beyond 3 min.

[5] In natural language processing, a one-hot vector is a $1xN$ matrix (vector) used to distinguish each word in a vocabulary from every other word in the vocabulary. The vector consists of 0s in all cells with the exception of a single 1 in a cell used uniquely to identify the word. In our case, each IP address is treated as a word.

Table 5: Buffer warning performance with data in group *A*.

Type	Baseline		*Requet*	
	Precision	Recall	Precision	Recall
BfW	51.0	11.1	79.0	68.7
NBfW	86.0	98.1	94.1	96.5
Accuracy	84.9		92.0	

Table 6: Video state performance with data in group *A*.

Type	Baseline		*Requet*	
	Precision	Recall	Precision	Recall
Stall	31.1	7.6	70.4	51.9
Buf. Decay	32.0	16.3	78.0	78.7
Buf. Increase	64.1	57.6	80.2	84.2
Steady	57.6	80.2	90.7	92.2
Accuracy	55.4		84.2	

6.1 Buffer Warning Prediction

The first metric we examine is buffer warning. We set the threshold for buffer level warning, $BuffWarning_{\text{thresh}}$, to be 20 secs. This provides ample time to provision enough bandwidth before an actual stall occurs.

For this metric, each time window in our dataset is labeled with either "no buffer warning" (NBfW) or "buffer warning" (BfW). In group *A*, significantly more chunks are labeled with NBfW (84%) than BfW (16%). The results in Table 5 show that both baseline and *Requet* perform well for this task, with accuracy reaching 85% and 92%, respectively. We see that precision and recall for NBfW are higher than those for BfW in both baseline and *Requet*. Given the current label is BfW, *Requet* provides significantly higher probability of predicting BfW correctly with recall of 68% over 11% for the baseline. This is because *Requet* uses chunk features to detect the case when no chunks have recently arrived. However, it is difficult for the baseline system to identify such cases due to the lack of chunk detection. For example, baseline can not differentiate packets as being part of a chunk or background traffic.

6.2 Video State Prediction

The results of video state prediction are shown in Table 6. *Requet* achieves overall accuracy of 84%, compared to 55% for baseline, representing a 53% improvement. *Requet* also outperforms baseline in precision and recall for each state.

Stall, buffer decay, buffer increase and steady state appear in 3.7%, 5.9%, 42.8% and 47.6% of chunks in group *A* respectively (Table 4). The precision and recall for both systems increase in the same order of stall, buffer decay, buffer increase and steady.

However, baseline achieves below 40% in precision and recall for both the stall and buffer decay states. This implies that during these two states, network traffic does not have a significant pattern for baseline to discover. Furthermore, during steady state there can be gaps of 30 sec or longer. A long gap also occurs when buffer is in decay state. Baseline features cannot separate buffer decay from steady state.

Examination of the *Requet* model reveals that audio chunk count for each 20 sec window is an important feature to predict video state. For example, if there are a few audio chunks in the past 20 sec it is likely that buffer is increasing, and if there are no audio chunks in the past 120 sec it is likely to be in stall state. This explains the relatively high performance of *Requet* .

Table 7: Video resolution performance with data in group *A*.

Type	Baseline		*Requet*	
	Precision	Recall	Precision	Recall
144p	13.0	7.6	80.6	79.9
240p	14.6	10.1	68.7	64.3
360p	14.1	9.9	49.2	64.4
480p	24.7	33.3	64.9	63.8
720p	24.5	30.3	60.6	54.5
1080p	22.2	20.1	75.0	76.9
Accuracy	21.8		66.9	

6.3 Video Resolution Prediction

It is extremely challenging for baseline to predict video resolution even with history of up to 200 sec. Overall accuracy is only 22%, slightly better than randomly picking one out of six choices.

As seen in Fig. 8, there is a large overlap of average playback bitrates of video clips of different resolutions due to varying activity levels in the video content. Without any knowledge about the content of the video or the video state, it is extremely difficult if not impossible to associate a chunk given its playback bitrate with the resolution it is encoded with. Furthermore, without knowing video state there is a large overlap in video chunk size and chunk duration across resolutions as seen in Fig. 10.

Requet utilizes chunk-based features. It uses the frequency of audio chunks to indicate the video state. The state information allows *Requet* to determine the resolution of the most recent video chunk with greater accuracy. By using both audio and video chunks, *Requet* achieves a 66% accuracy for predicting resolution (six levels). This result demonstrates that *Requet* is able to use audio chunk features to enhance video resolution prediction. By narrowing down the options in resolution to three: small(144p/240p), medium (360p/480p), and large (720p/1080p), *Requet* achieves an accuracy of 87%. If the number of options is reduced to two: small(144p/240p/360p) and large (480p/720p/1080p) the accuracy improves to 91%.

6.4 Extended Test

Up to this point we have reported results from our systems trained with part of group *A* and tested on different clips in group *A*. Next, we use group *A* as the training data for *Requet* and evaluate with groups B_1, B_2, *C*, and *D*. We test *Requet* on 10 clips from groups B_1 and B_2 for residential WiFi settings in US and India respectively, to see how they perform on unseen clips of similar length and unseen WiFi environments. In addition, we use the same lab WiFi environment in group *A*, to test *Requet* on 5 clips of shorter length of 5 min in group *C* and longer length of 25 min in group *D*. Fig. 11 reports the average precision and recall of these four tests along with the 4-fold cross validation results from group *A*.

Depending on the environment and QoE metric, performance of these extended sets of tests either improves or deteriorates compared with results from group *A* reported earlier in this section. For

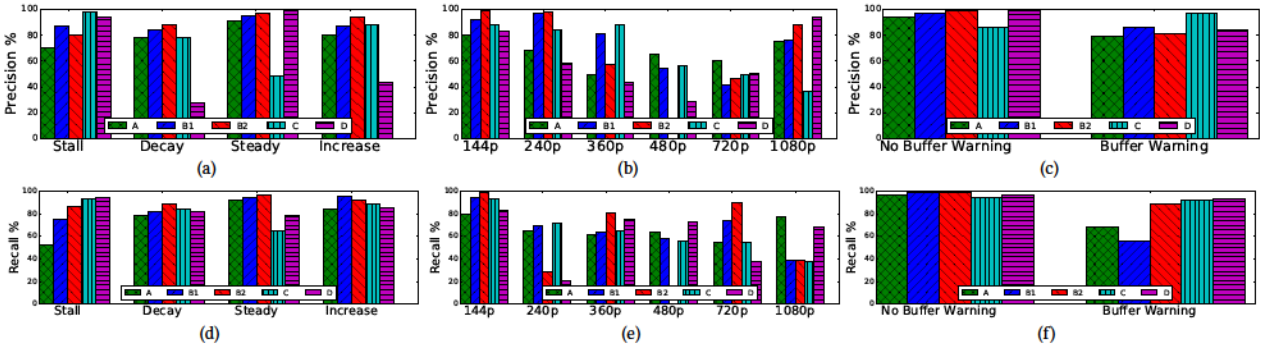

Figure 11: Accuracy of *Requet* models trained with group *A*. (a) Precision of video state, (b) Precision of video resolution, (c) Precision of stall warning, (d) Recall of video state, (e) Recall of video resolution, (f) Recall of stall warning.

example, groups B_1, B_2, and C have improved precision and recall in predicting stall and buffer decay states. Group D shows lower precision in predicting buffer decay, but higher recall for both stall and buffer decay. Improved precision and recall results appear for predicting buffer threshold warning.

Accuracy for video resolution varies from experiment to experiment. Surprisingly, group B_2 has the highest overall accuracy of 70% when training with group A. This is in part due to that there were zero 480p events collected in group B_2. This resolution level has lower precision than 144p, 240p, and 1080p (see Table 7), and is extremely difficult for the other test sets to predict as well.

Most precision and recall results for other sets are better than group A with a few exceptions. This could be due to the fact that group A includes movement experiments, while the other groups only contain static ones. A video session naturally exhibits different behavior in different types of environments. In addition, we plan to improve our prediction models by studying how the imbalance in data samples impacts the precision and recall of each model.

7 RELATED WORK

Traditional traffic monitoring systems rely on DPI to understand HTTP request and reply messages. The systems use meta-data to understand ABR and infer video QoE. The MIMIC system estimates average bitrate, re-buffering ratio and bitrate switches for a session by examining HTTP logs [29]. Comparatively, BUFFEST builds ML classifiers to estimate buffer level based either on the content of HTTP requests in the clear or on unencrypted HTTPS requests by a trusted proxy [24]. HighSee identifies HTTP GET requests and builds a linear Support Vector Machine (SVM) [13] model to identify audio, video, and control chunks to separate audio, video and control flows [19].

For encrypted traffic, proposals fall in two categories. The first category builds session models offline by detecting HTTP requests as in eMIMIC [28], while the second category builds ML models to predict QoE either offline or online.

Offline Models: The offline approach uses entire video session traffic to generate features to classify the session into classes. YouQ classifies a session into two to three QoS classes [34]. The system in [15] builds models to roughly put a session into three categories in terms of stall events ("non-stall", "0-1 stalls", or "2-or-more stalls"), or three classes based on average quality level. Using simulation, [42] builds ML models to predict average bitrate, quality variation, and three levels of stall ratio (no, mid, severe) for entire sessions using post processing. Comparatively, [27] classifies a session in two categories (with or without stall events) based on cell-related information collected at the start of a video session.

Online Models: The online approach uses traffic from the past time window in the session to generate features to predict QoE metrics specific to that time window. The system in [32] develops features based on both network and transport level information in a 10sec time window to build separate classifiers for HTTPS and QUIC traffic to infer startup delay (below or above a given threshold), stall event occurrence, and video quality level ("low" and "high"). This system uses features including packet level statistics such as standard deviation. This has a relatively large memory requirement and makes it infeasible in middleboxes.

Flow Identification: Identifying video flows from encrypted traffic is orthogonal to the QoE detection problem for given ABR flows. It is an example of the broad encrypted traffic classification problem. The Silhouette system [26] detects video chunks (also named Application Data Units) from encrypted traffic in real-time for ISP middleboxes using video chunk size, payload length, download rate threshold values. The real-time system in [37] identifies Netflix videos using TCP/IP header information including TCP sequence numbers. This approach relies on a "finger print" database built from a large number of video clips hosted by Netflix. The finger print is unique for each video title, therefore it is ineffective in classifying new video titles not previously seen. The system in [41] classifies an encrypted Youtube flow every 1sec interval into HTTP Adaptive Streaming (HAS) or non-HAS flows in real-time. For a HAS flow, it further identifies the buffer states of the video session into filling, steady, depleting and unclear. The high accuracy to predict buffer state is partly due to the fact that the entire dataset contains only 3 clips with multiple sessions for each clip. This system also uses a feature based on the standard deviation of packet size, which is not feasible for implementation in middleboxes due to the memory requirement.

8 CONCLUSION AND FUTURE WORK

We present *Requet*, a system for REal-time QUality of experience detection for Encrypted Traffic. We focus on three QoE metrics (1) buffer warning, (2) video state, and (3) video quality, as they

are crucial in allowing network level resource provisioning in real-time. We design a video state labeling algorithm to automatically generate ground truth labels for ABR traffic data.

Requet consists of the ChunkDetection algorithm, chunk feature extraction, and ML QoE prediction models. Our evaluation using YouTube traffic collected over WiFi networks demonstrates *Requet* using chunk-based features exhibit significantly improved prediction power over the baseline system using IP-layer features.

We demonstrate that the *Requet* QoE models trained on one set of clips exhibit similar performance in different network environments with a variety of previously unseen clips with various lengths.

A current limitation of *Requet* is that it is based on YouTube. Therefore, one direction of our future work includes building a generic model for a wide range of networks and client algorithms for ABR. We plan to evaluate traffic over 3G and LTE cellular networks, and expand into additional services such as Netflix. Another direction of our future work includes investigating resource scheduling in real-time to utilize the QoE predictions from *Requet*. We aim to study the joint effect of operator optimization and content provider video optimization mechanisms.

ACKNOWLEDGMENTS

We thank the Shepard, Wei Tsang Ooi, and the anonymous reviewers for helpful comments and suggestions. This work was supported in part by NSF grants CNS-1650685, CNS-1413978, and DGE 16-44869.

REFERENCES

[1] About wireshark. https://www.wireshark.org/about.html.
[2] Cisco visual networking index: Global mobile data traffic forecast update, 2016–2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.
[3] Telerik fiddler, the free web debugging proxy. https://www.telerik.com/fiddler.
[4] How Google is making YouTube safer for its users, Fortune. http://fortune.com/2016/08/02/google-youtube-encryption-https/, Aug. 2016.
[5] Encrypted traffic analytics, Cisco white paper. https://www.cisco.com/c/dam/en/us/solutions/collateral/enterprise-networks/enterprise-network-security/nb-09-encrytd-traf-anlytcs-wp-cte-en.pdf, 2018.
[6] 3GPP. Transparent end-to-end Packet-switched Streaming Service (PSS). TS 26.234, 3rd Generation Partnership Project (3GPP), 2010.
[7] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan. Prometheus: toward quality-of-experience estimation for mobile apps from passive network measurements. In *Proc. HotMobile*, Feb. 2014.
[8] A. Ahmed, Z. Shafiq, H. Bedi, and A. R. Khakpour. Suffering from buffering? detecting QoE impairments in live video streams. In *Proc. IEEE ICNP*, Oct. 2017.
[9] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle, and R. Holz. Mission accomplished?: HTTPS security after diginotar. In *Proc. ACM IMC*, Nov. 2017.
[10] L. Armasu. Netflix adopts efficient HTTPS encryption for its video streams. https://www.tomshardware.com/news/netflix-efficient-https-video-streams, 32420.html, Aug. 2016.
[11] P. Casas, M. Seufert, and R. Schatz. YOUQMON: a system for on-line monitoring of YouTube QoE in operational 3G networks. *SIGMETRICS Performance Evaluation Review*, 41(2):44–46, 2013.
[12] G. Cofano, L. De Cicco, T. Zinner, A. Nguyen-Ngoc, P. Tran-Gia, and S. Mascolo. Design and experimental evaluation of network-assisted strategies for http adaptive streaming. In *Proc. of MMSys*. ACM, 2016.
[13] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
[14] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind. Innovating transport with QUIC: design approaches and research challenges. *IEEE Internet Computing*, 21(2):72–76, 2017.
[15] G. Dimopoulos, I. Leontiadis, P. Barlet-Ros, and K. Papagiannaki. Measuring video QoE from encrypted traffic. In *Proc. ACM IMC*, Nov. 2016.
[16] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. ACM SIGCOMM*, Aug. 2011.
[17] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson. The security impact of HTTPS interception. In *Proc. NDSS*, Feb. 2017.
[18] R. T. Fielding and J. F. Reschke. Hypertext transfer protocol (HTTP/1.1): message syntax and routing. *RFC*, 7230:1–89, 2014.
[19] S. Galetto, P. Bottaro, C. Carrara, F. Secco, A. Guidolin, E. Targa, C. Narduzzi, and G. Giorgi. Detection of video/audio streaming packet flows for non-intrusive QoS/QoE monitoring. In *IEEE Int. Workshop on Measurement and Networking*, Sept. 2017.
[20] T. A. Guarnieri, I. Drago, A. B. Vieira, Í. Cunha, and J. M. Almeida. Characterizing QoE in large-scale live streaming. In *Proc. IEEE GLOBECOM*, Dec. 2017.
[21] T. K. Ho. Random decision forests. In *Proc. IEEE Conf. Document analysis and recognition*, 1995.
[22] T. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: evidence from a large video streaming service. In *Proc. ACM SIGCOMM*, Aug. 2014.
[23] A. M. Kakhki, S. Jero, D. R. Choffnes, C. Nita-Rotaru, and A. Mislove. Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proc. ACM IMC*, Nov. 2017.
[24] V. Krishnamoorthi, N. Carlsson, E. Halepovic, and E. Petajan. BUFFEST: predicting buffer conditions and real-time requirements of HTTP(S) adaptive streaming clients. In *Proc. ACM MMSys*, June 2017.
[25] W. Law. Ultra-Low-Latency STreaming Using Chunked-Encoded and Chunked-Transferred CMAF. Technical report, Akamai, 2018.
[26] F. Li, J. W. Chung, and M. Claypool. Silhouette: Identifying youtube video flows from encrypted traffic. In *Proc. NOSSDAV*, June 2018.
[27] Y. Lin, E. M. R. Oliveira, S. B. Jemaa, and S. Elayoubi. Machine learning for predicting QoE of video streaming in mobile networks. In *IEEE ICC*, May 2017.
[28] T. Mangla, E. Halepovic, M. Ammar, and E. Zegura. eMIMIC: estimating http-based video QoE metrics from encrypted network traffic. In *Proc. IEEE TMA*, 2018.
[29] T. Mangla, E. Halepovic, M. H. Ammar, and E. W. Zegura. MIMIC: using passive network measurements to estimate http-based adaptive video QoE metrics. In *Proc. IEEE TMA*, 2017.
[30] A. Mansy, M. H. Ammar, J. Chandrashekar, and A. Sheth. Characterizing client behavior of commercial mobile video streaming services. In *Proc. ACM MoVid*, Mar. 2014.
[31] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *Proc. ACM SIGCOMM*, Aug. 2017.
[32] M. H. Mazhar and Z. Shafiq. Real-time video quality of experience monitoring for HTTPS and QUIC. In *Proc. IEEE INFOCOM*, Apr. 2018.
[33] A. Mondal, S. Sengupta, B. R. Reddy, M. J. V. Koundinya, C. Govindarajan, P. De, N. Ganguly, and S. Chakraborty. Candid with YouTube: adaptive streaming behavior and implications on data consumption. In *Proc. NOSSDAV*, June 2017.
[34] I. Orsolic, D. Pevec, M. Suznjevic, and L. Skorin-Kapov. YouTube QoE estimation based on the analysis of encrypted network traffic using machine learning. In *Proc. IEEE Globecom Workshops*, Dec. 2016.
[35] S. Petrangeli, T. Wu, T. Wauters, R. Huysegems, T. Bostoen, and F. De Turck. A machine learning-based framework for preventing video freezes in http adaptive streaming. *Journal of Network and Computer Applications*, 2017.
[36] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill. Studying TLS usage in android apps. In *Proc. ACM CoNEXT*, Dec. 2017.
[37] A. Reed and M. Kranch. Identifying HTTPS-protected netflix videos in real-time. In *Proc. CODASPY*, Mar. 2017.
[38] P. Schmitt, F. Bronzino, R. Teixeira, T. Chattopadhyay, and N. Feamster. Enhancing transparency: Internet video quality inference from network traffic. In *Proc. TPRC46*, 2018.
[39] T. Stockhammer. Dynamic adaptive streaming over HTTP -: standards and design principles. In *Proc. ACM MMSys*, Feb. 2011.
[40] D. Tsilimantos, T. Karagkioules, and S. Valentin. Classifying flows and buffer state for YouTube's HTTP adaptive streaming service in mobile networks. *CoRR*, abs/1803.00303, June 2018.
[41] D. Tsilimantos, T. Karagkioules, and S. Valentin. Classifying flows and buffer state for youtube's HTTP adaptive streaming service in mobile networks. In *Proc. ACM MMSys*, June 2018.
[42] V. Vasilev, J. Leguay, S. Paris, L. Maggi, and M. Debbah. Predicting QoE factors with machine learning. In *Proc. IEEE ICC*, May 2018.
[43] N. Vogt. Youtube audio quality bitrate used for 360p, 480p, 720p, 1080p, 1440p, 2160p. https://www.h3xed.com/web-and-internet/youtube-audio-quality-bitrate-240p-360p-480p-720p-1080p, 2015.
[44] F. Wamser, M. Seufert, P. Casas, R. Irmer, P. Tran-Gia, and R. Schatz. YoMoApp: A tool for analyzing QoE of YouTube HTTP adaptive streaming in mobile networks. In *Proc. European Conf. on Networks and Communications (EuCNC)*, June 2015.
[45] N. Weil. The state of MPEG-DASH 2016. http://www.streamingmedia.com/Articles/Articles/Editorial/Featured-Articles/The-State-of-MPEG-DASH-2016-110099.aspx.

9 APPENDIX

This appendix provides a description of the dataset acquired in §4, used for *Requet* chunk detection in §3, and for evaluation in §6. The dataset can be found in a Github Repository (https://github.com/Wimnet/RequetDataSet). The dataset is divided into 5 *group folders* for data from groups *A*, *B*1, *B*2, *C*, *D*, along with a summary file named 'ExperimentInfo.txt' for the entire dataset. Each line in the file describes an experiment using the following four attributes: (a) experiment number, (b) video ID, (c) initial video resolution, and (d) length of experiment in seconds.

A group folder is further divided into two subfolders, one for PCAP files, and the other for txt files. Each experiment is described by a PCAP file and a txt file. The PCAP file with name in the form of (i) *'baseline_{date}_exp_{num}.pcap'* is for an experiment where the end device is static for the entire duration whereas a file with name in the form of (ii) *'movement_{date}_exp_{num}.pcap'* is for an experiment where the end device movement occurs during the experiment. The txt file names end with *'merged.txt'*. The txt file contains data colletect from YouTube API and summary of PCAP trace for the experiment.

In each *'merged.txt'* file, data is recorded for each 100ms interval. Each interval is represented as: [*Relative Time*, # Packets Sent, # Packets Received, # Bytes Sent, # Bytes Received, [Network Info 1], [Network Info 2], [Network Info 3], [Network Info 4], [Network Info 5], ... , [Network Info 25], [*Playback Info*]].

Relative Time marks the end of the interval. *Relative Time* is defined as the time since the Javascript Node server hosting the YouTube API is started. Relative Time for the 0^{th} interval is defined as 0 sec. It is updated in intervals of 100ms. TShark is called prior to the Javascript Node server. Therefore, the 0^{th} interval contains Wireshark data up to the start of the Javascript Node server.

Network Info i is represented as: [IP_Src, IP_Dst, Protocol, # Packets Sent, # Packets Received, # Bytes Sent, # Bytes Received] for each interval. IP_Src is the IP address of the end device. The top 25 destination IP addresses in terms of total bytes sent and received for the entire session are recorded. For each *i* of the top 25 IP_Dst addresses, the Protocol associated with the higher data volume for the interval (in terms of total number of packets exchanged) is selected, and data volume in terms of packets and bytes for each interval is reported for the IP_Src, IP_Dst, Protocol tuple in [Network Info *i*].

Playback Info is represented as: [*Playback Event*, *Epoch Time*, *Start Time*, *Playback Progress*, *Video Length*, *Playback Quality*, *Buffer Health*, *Buffer Progress*, *Buffer Valid*]. From the perspective of video playback, a YouTube session can contain three exclusive regions: *buffering*, *playing*, and *paused*. YouTube IFrame API considers a transition from one playback region into another as an event. It also considers as an event any call to the API to collect data. The API enables the recording of an event and of detailed information about playback progress at the time the event occurs. *Epoch Time* marks the time of the most recent collection of YouTube API data in that interval. *Playback Info* records events occurred during the 100-ms interval.

Each field of *Playback Info* is defined as follows:

- **Playback Event** - This field is a binary array with four indexes for the following states: 'buffering', 'paused', 'playing', and 'collect data'. The 'collect data' event occurs every 100ms once the video starts playing. For example, an interval with a Playback Event [1,0,0,1] indicates that playback region has transitioned into 'buffering' during the 100ms interval and a 'collect data' event occurred.
- **Epoch Time** - This field is the UNIX epoch time in milliseconds of the most recent YouTube API event in the 100ms interval.
- **Start Time** - This field is the UNIX epoch time in milliseconds of the beginning of the experiment.
- **Playback Progress** - This field reports the number of seconds the playback is at epoch time from the start of the video playback.
- **Video Length** - This field reports the length of the entire video asset (in seconds).
- **Playback Quality** - This field is a binary array of size 9 with indices for the following states: unlabelled, tiny (144p), small (240p), medium (360p), large (480p), hd720, hd1080, hd1440, and hd2160. The unlabeled state occurs when the video is starting up, buffering, or paused. For example, a Playback Quality [0, 1, 1, 0, 0, 0, 0, 0, 0] indicates that during the current interval, video playback experienced two quality levels - tiny and small.
- **Buffer Health** - This field is defined the amount of buffer in seconds ahead of current video playback. It is calculated as:

$$\text{Buffer Health} = \text{Buffer Progress} \times \text{Video Length} - \text{Playback Progress}$$

- **Buffer Progress** - This field reports the fraction of video asset that has been downloaded into the buffer.
- **Buffer Valid** - This field has two possible values: True or '−1'. True represents when data is being collected from the YouTube IFrame API. '−1' indicates when data is not being collected from the YouTube IFrame API during the current interval.