Improving ABR Video Streaming Design with Systematic QoE Measurement and Cross Layer Analysis

by

Shichang Xu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Doctoral Committee:

Professor Z. Morley Mao, Chair
Assistant Professor Mosharaf Chowdhury
Assistant Professor Hessam Mahdavifar
Lead Scientist Subhabrata Sen, AT&T Labs Research

Shichang Xu

xsc@umich.edu

ORCID iD: 0000-0002-8997-4041

To my family and beloved.

ACKNOWLEDGEMENTS

I am very fortunate to have the opportunity to pursue a Ph.D. It is a very unique experience: having the freedom to learn and work on interesting unsolved problems. I not only learn valuable knowledge on a specific domain, but also learn the general methodology to tackle a problem step-by-step scientifically. Looking back the past years, I am extremely thankful for all these people who help and support me. I would never be able to make it without their support.

I would like to express my deepest thanks for my advisor Z. Morley Mao. I am so fortunate to have the opportunity to work with her. She gave me valuable guidance on how to do independent research, how to manage projects and how to collaborate with other researchers. She always inspires me to think deeper and gives me valuable advice on the directions for improvement. She encourages me to persist and not easily give up. I will always remember the time when she worked late into night on my first full-length paper submission. I am very grateful for her help and support along the path.

I would also like to thank my mentor and one of my best friends, Shubho Sen from AT&T Labs Research. We have worked together for four years on every work in this dissertation. He offered me opportunity to do internships with him every summer. He always gives me courage and confidence to believe in my work. I enjoyed the numerous calls with him to discuss different problems and do brainstorming. He always cares for me and gives me considerate suggestions. I am so honored to have the opportunity to keep working with him.

I would like to thank Professor Mosharaf Chowdhury and Professor Hessam Mahdavi-

far to serve on my dissertation committee. Their time and input are greatly appreciated.

I am very grateful to my collaborators. I benefit a lot from the collaboration with Professor David Choffnes from Northeastern University and Eric Petajan from AT&T.

I was fortunate to spend three summers doing internship at AT&T Labs. I would like to thank Ed Lambert, Bill Weir, Doug Sillars, Delton Noronha, Barry Nelson, Emir Halepovic, Rupesh Lunia, Oliver Spatscheck and Jennifer Yates for their help. I would also like to thank Rajesh Mahindra, Vinoth Chandar, Sivabalan Narayanan, James Yu and Linda Fu for their help during my internship at Uber Inc. I learn a lot from these internship experience.

I want to thank Professor Ethan Katz-Bassett from University of Southern California (till 2016). I was fortunate to spend a summer in his lab doing research during my undergraduate. That experience helped me understand what research is and led my path to pursuing a Ph.D. I cannot thank enough for his kind guidance and help through my application of graduation schools. I would also like to thank Professor Yong Cui and Professor Jun Li from Tsinghua University for their help.

I am thankful to join the Robustnet research group. I learn a lot from the senior labmates, including Ashkan Nikravesh, Qi Alfred Chen, Yihua Guo, Mehrdad Moradi, Sanae Rosen, Hongyi Yao, Jack Yunhan Jia. They set a good example for me and always generously offer help whenever I need. I also thank my dear friends and colleagues Yuru Shao, Ke Hong, Yikai Lin, Xiao Zhu, Chao Kong, Shengtuo Hu, Yulong Cao, Xumiao Zhang, Jiachen Sun, Won Park, Jiwon Joung, Can Carlak, Dongyao Chen, Amir Rahmati, Sai Gouravajhala, Muhammed Uluyol, Zhe Wu, Vaspol Ruamviboonsuk, Joel Woude and many others.

I would thank the CSE staff, including Dawn Freysinger, Ashley Andreae, Stephen Reger, Karen Liska, and Jamie Goldsmith etc. for their help and support during the past five years.

Special thanks go to my many friends at Ann Arbor, including but not limited to Feng-

min Hu, Shuqi Cheng, Yuanying Wang, Yuanzhan Wang, Dan Zhao, Chunan Huang, Yin Xie and Yibo Pi etc. We share happy moments. We also support and comfort each other when things are not smooth. I am fortune to meet you in my life.

Last but not least, I would like to thank my family: my grandparents Maoda Xu, Maozhen Nie, Shenlin Rao, Haizhen Zou, Heping Wu, Shuilian Rao, my parents Aiguo Xu and Wenying Rao, my sister and brother-in-law Yuchen Xu and Yuantong Huang for their unconditional support and love. Duolan, you are the best thing that ever happened to me. This dissertation is delicated to all of you.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Adaptive Bitrate streaming (ABR) has been widely adopted by mobile video services to deliver satisfying Quality of Experience (QoE) over real-world network with time-varying cellular bandwidth conditions. To build an ABR service, a wide range of critical components spanning different entities need to be determined. It is challenging to achieve designs with good QoE properties, as the streaming performance depends on complex interactions among the various factors. To make it more complex, many design decisions also involve tradeoffs among different QoE metrics.

To address this challenge, in this dissertation, we build four systems to provide systematic support for video QoE measurements and analysis. First, we build a general black-box measurement platform based on standard ABR protocols and common UI designs. It analyzes HTTP information in the network traffic and correlates UI events of mobile video apps to reveal ABR design and identify QoE issues. Second, to address the challenge brought by increasingly adopted encryption protocols such HTTPS and QUIC, we develop a technique called CSI to infer ABR video adaptation behavior based on packet size and timing information still available in the encrypted traffic. Third, we explore a conceptually very different approach to QoE measurement — utilizing the on-device recording capability to record the video displayed on the mobile device screen and measuring delivered QoE from this recording. We design a novel system VideoEye to conduct such screen-recording-based QoE analysis. Lastly, to understand the interaction of existing video streaming system design with the new transport protocol QUIC, we build a platform WIQ to perform what-if analysis and measure the video QoE impact of QUIC without the need of modifying the

server or client implementation. Leveraging these systems, we perform measurements on popular streaming services, understand the QoE implications of various ABR design, identify a wide range of QoE issues and develop best practices.

CHAPTER I

Introduction

1.1 Overview

Mobile video streaming has become increasingly popular in recent years. It now dominates cellular traffic, accounting for 60% of all mobile data traffic and is predicted to grow to 78% by 2021 [1]. However, achieving satisfying streaming user experience over real-world networks with time-varying bandwidth conditions remains challenging. A recent Internet-scale study indicates that 26% of smartphone users face video streaming QoE problems daily [2].

To deliver good streaming QoE, Adaptive Bitrate streaming (ABR) has been widely adopted to adapt the streaming video quality based on network conditions. However, it is challenging to achieve ABR designs with good QoE properties, as the streaming performance depends on the complex interactions between different factors spanning different entities across multiple layers. Some design decisions also involve a complex tradeoff between different QoE metrics as well as other factors such as data usage etc. For example, the track selection algorithm should try to stream the best-quality track supported by the network condition. However, if the track selection is too aggressive, the player is likely to encounter frequent stalls under highly variable network conditions. There is no simple answer on how to realize the best design.

To help identify QoE issues and develop best practices, it is important to perform con-

tinuous measurements to understand the performance and QoE implications of various design decisions. In particular, we need to understand the interactions between the application layer adaptation behavior and network delivery at the transport layer to diagnose the root cause of identified QoE issues and make better design choices. However, performing continuous measurement to gain such understanding on diverse ABR streaming systems can be especially challenging for the following reasons.

- **Limited visibility into system design and QoE.** The proprietary video streaming services do not readily expose information on the design choices and achieved QoE. In addition, the design of various services differs significantly in various aspects, making it challenging to develop a general methodology to infer their design and measure delivered QoE. Approaches like code disassembly suffer from limitations such as code obfuscation. Other approaches that either leverage app-specific features such as URL patterns [3, 4] or rely on deep modifications to the apps [5, 6] cannot be generally applied.

- **Limited information due to traffic encryption.** End-to-end encrypted transport protocols are being increasingly used for security and privacy considerations. For example, video providers such as Netflix [7] use HTTPS to encrypt the video traffic. QUIC, a new encrypted transport protocol, also elicited a strong interest in the field and leads to increasing adoption by industry including Youtube [8]. Many existing QoE analysis techniques require to obtain the HTTP request information including URL and headers from the network traffic, and thus can no longer be applied.

- **Limited support for QUIC.** QUIC has many new features including 0-RTT and better loss recovery. It has been shown to have better performance compared to TCP [9, 10] and attracts interest in adoption for ABR video streaming. However, as QUIC differs from TCP in various aspects including connection management and congestion control etc., it is not clear how the performance of existing ABR stream-

ing services will be if they adopt QUIC. There is little work to support measurements for understanding how ABR services should be designed to better interact with the new features of QUIC.

These challenges make it difficult for entities involved in the ABR streaming system to diagnose QoE issues and develop better designs. For example, as network operators desire to gain a deep understanding into how streaming applications interact with various network conditions, it is challenging for them to understand the QoE impact of various network policies and better manage the network to optimize streaming QoE. It is also challenging for app developers to understand the complex tradeoff imposed by various design choices. For example, some applications use multiple TCP connections to download multiple video chunks concurrently to improve the overall throughput, but it can delay the arrival of the most needed chunk and increase the potential of stalls when buffer occupancy is low [11].

It would be valuable for various entities to perform measurements and gain useful insights to improve the ABR design. In this dissertation, we focus on developing systematic measurement support to enable video QoE measurements and analysis for different entities given all the above challenges such as the proprietary nature of commercial systems. We demonstrate that **systematic support for video QoE measurements and cross-layer is essential to understand the QoE implications of various ABR design, identify QoE issues and develop best practices**. As summarized in Table 1.1, this dissertation explores this problem along four use cases. It develops novel measurement schemes for these use cases and uses them to develop deep insights into complex ABR behavior.

1. We develop a general methodology that leverages common properties of ABR streaming apps to derive valuable insights into the proprietary ABR services without access to the source code [11]. We perform analysis on HTTP requests in the network traffic based on standard ABR protocols and understand what chunks are downloaded. We also extract critical QoE information from common UI components. Based on this information, we infer the apps' internal buffer state which is

critical to gain insights into their behavior. We craft targeted black-box experiments to stress-test the apps by emulating various network conditions and manipulating the communication between the client and server. By analyzing the reaction of the apps, we are able to glean critical properties of their design.

2. With the increasing adoption of traffic encryption, HTTP information is no longer available, making existing traffic analysis techniques no longer applicable. We develop a novel system called Chunk Sequence Inferencer (CSI) to perform analysis on encrypted traffic to infer the downloaded chunk identity. CSI leverages the key insight that common encryption protocols do not obfuscate traffic volume information, likely due to concerns of overhead through padding. It also harnesses the chunk size variability fundamentally caused by the increasingly adopted Variable Bitrate Encoding (VBR). We design CSI to work for ABR streaming systems with various designs covering popular streaming services.

3. As the adoption of traffic encryption makes it challenging to analyze video QoE from network traffic, we explore a conceptually very different approach. We develop a novel system called VideoEye that utilizes commonly available standard on-device recording capabilities of smartphones to record the video displayed on the device screen, and measures the delivered QoE directly from this recording. We conduct a measurement study to characterize the overhead and distortion of screen recording. We find that screen recording does not perturb the ABR video playback process, but does introduce significant distortions involving compression artifacts and color space distortions in recorded videos. We develop techniques to measure streaming QoE based on video properties invariant of recording distortions.

4. The above measurement systems focus on understanding the performance of existing ABR systems, which mostly use HTTP/HTTPS as the underlying network protocol. To help developers understand the streaming QoE of these systems if they adopt

Problem scope	Project
Designing general blackbox measurement platform	Dissecting VOD Services for Cellular: Performance, Root Causes and Best Practices
Developing techniques to analyze QoE with traffic encryption	CSI: Inferring ABR Video Streaming Behavior for HTTPS and QUIC
	Measuring ABR Video Streaming QoE from Screen Recording
Developing platforms to understand QoE with QUIC	WIQ: What-if analysis platform for QUIC

Table 1.1: Summary of dissertation work

QUIC, we build a what-if analysis platform without the need to modify the client and server of existing ABR system implementation. The platform utilizes two proxies to convert HTTPS requests sent from mobile apps to QUIC, then convert them back to HTTPS before servers receive the requests. Evaluations show that the platform incurs minimal overhead. We demonstrate that the platform effectively helps compare streaming QoE with HTTPS and QUIC.

1.2 Thesis organization

The dissertation is organized as follows. We describe background on ABR video streaming and summarize related work in Chapter II. Chapter III presents our measurement study using developed general blackbox measurement platform. Chapter IV presents the system CSI to analyze ABR streaming behavior from encrypted network traffic. Chapter V presents the system CSI to analyze ABR QoE using on-device screen recording. We show WIQ to analyze the QoE impact of QUIC adoption in Chapter VI. We conclude the dissertation in Chapter VIII.

CHAPTER II

Background and Related Work

We provide some background on ABR video streaming and traffic encryption.

2.1 ABR streaming and QoE

Video streaming over the best-effort Internet is challenging, due to variability in available network bandwidth. To address such problems and provide satisfactory QoE, ABR has been proposed to adapt the video bitrate based on network conditions.

In ABR, videos are encoded into multiple *tracks*. Each track describes the same media content, but with a different quality level. The tracks are broken down into multiple shorter *chunks* and the client can switch between tracks on a per-chunk basis. Media meta-information including the available tracks, chunk durations and URIs is described in a metafile called *manifest* or *playlist*.

The manifest specifies a bitrate for each track (referred to as *declared bitrate*) as an estimation of the network bandwidth required to stream the track. Note that this value can be different from the actual bandwidth needed for downloading individual chunks especially in the case of *Variable Bitrate (VBR) encoding*. How to set this declared bitrate is left to the specific service, and a common practice is to use a value in the neighborhood of the peak bitrate of the track. In addition to the declared bitrate, some ABR implementations also provide more fine-grained information about chunk sizes, such as average *actual chunk*

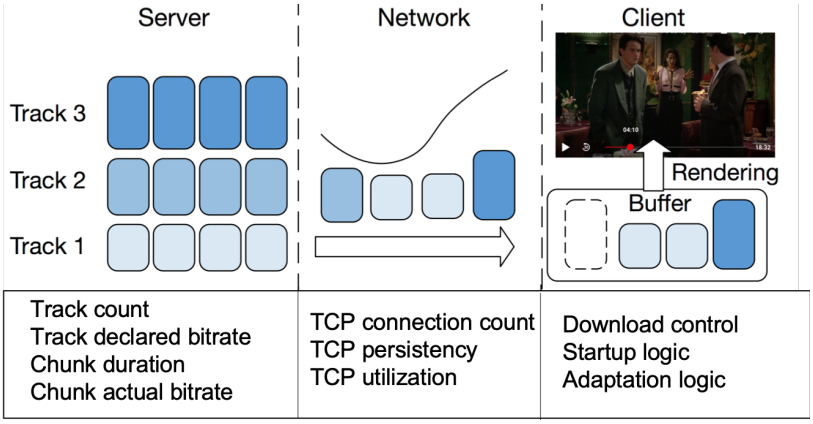

Figure 2.1: ABR streaming and relevant design factors

bitrate.

At the beginning of a session, the player downloads the manifest from the server, and uses the HTTP/HTTPS/QUIC protocol to fetch media chunks from the server. To absorb network variance and minimize stall events, the player usually maintains a buffer and tries to fetch chunks ahead of playback time. During streaming, the client-side adaptation logic (often proprietary) determines what track to fetch next based on a variety of factors, such as the estimated available network bandwidth and playback buffer occupancy.

There exist a number of different implementations of the above high-level ABR design, involving different file format and protocols. HTTP Live Streaming (HLS) [12], Dynamic Adaptive Streaming over HTTP (DASH) [13] and Smooth Streaming [14] are the most well known of these.

Regardless of implementation details, a wide range of factors spanning the server, the network and the client and across the transport and application layers can be customized based on the system designers' considerations around different tradeoffs to optimize streaming performance. For instance, the client can adopt different track selection algorithms to balance video quality and stalls. We summarize the relevant factors in Figure 2.1. While developing objective measures of overall user QoE for video streaming is still an active research area, it is commonly acknowledged that QoE is highly correlated to

a few metrics listed below.

- **Video quality.** One commonly used metric to characterize video quality is average video bitrate, i.e. the average declared bitrate of chunks shown on the screen. A low video bitrate indicates poor video quality, leading to poor user experience. However, the average bitrate by itself is not sufficient to accurately reflect user experience. As we discuss in more detail in § 3.4.1.3, user experience is more impacted by the playback of low quality, low bitrate tracks. It is therefore important to reduce the duration of streaming such tracks. To account for this, another metric is the percentage of playtime when low quality tracks are streamed.

- **Video track switches.** Frequent track switches impair user experience. One metric to characterize this is the frequency of switches. In addition, users are more sensitive to switches between non-consecutive tracks.

- **Stall duration.** This is the total duration of stall events during a session. A longer stall duration means higher interruptions for users and leads to poorer user experience.

- **Startup delay.** The startup delay measures the duration from the time when the users click the "*play*" button to the time when the first frame of video is rendered on the screen and the video starts to play. A low startup delay is preferred.

Each metric by itself provides only a limited viewpoint and all of them need to be considered together to characterize overall QoE.

2.2 Motivation for performing active measurement

Video streaming QoE has a direct impact on user engagement and revenue [15]. For example, a global dataset [16] shows that even a 0.2% increase in stall ratio could reduce play duration by 8 minutes. However, designing a practical ABR streaming system with

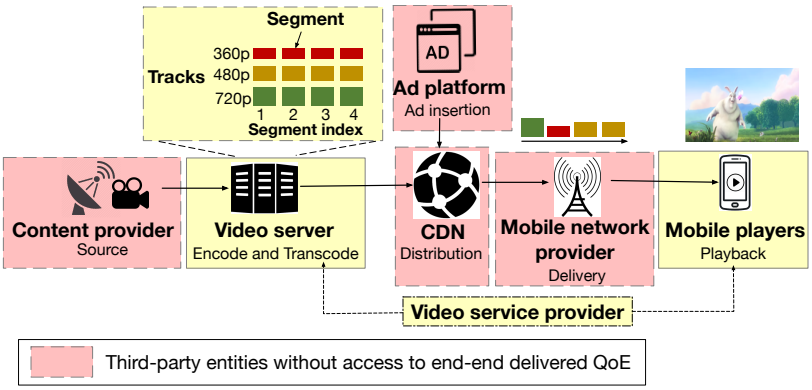

Figure 2.2: ABR video streaming ecosystem.

good QoE properties (e.g., low stall durations, high video quality) is quite complex and challenging. In addition to the streaming services themselves, a variety of entities including original content providers (e.g., Walt Disney), mobile network providers (e.g., Verizon) and CDNs (e.g., Akamai) etc. as depicted in Figure 2.2 are involved in the end-to-end streaming ecosystem. The streaming performance depends on complex interactions among various factors spanning these entities (e.g., content encoding on video servers, network policies and adaptation logic in players). Some design decisions also involve complex tradeoffs. For example, the track selection algorithm should balance the need to deliver higher quality and bitrate videos with the need to reduce the likelihood of stalls. It is nontrivial to realize the best design. As a result, existing designs exhibit high diversity across commercial services and keep evolving over time [4, 17, 18]. Changes at different entities can collectively impact video QoE in ways that may not always be fully anticipated, given the associated complex interactions.

In this dynamic environment, it is important for different entities to monitor delivered QoE, identify emergent QoE issues, and ultimately develop better designs. For video service providers, they typically instrument the server and client to collect QoE information directly. However, for the other entities, acquiring such information is very challenging in practice, as commercial streaming services are closed proprietary systems and do not expose QoE information to other entities such as network providers. To address their need for

realistic QoE measurements, these entities typically conduct their own testing in the wild. This even spawns an industry to offer such testing as a service (e.g., [19, 20, 21]). Specifically, they stream videos on mobile devices in different locations of the cellular network and use blackbox techniques to collect information on the delivered QoE.

Some concrete use cases include (1) mobile network operators hope to understand how well popular video streaming services work in their network and how they develop better network policies to manage video traffic. (2) CDN service providers hope to characterize how well their servers serve video content and diagnose potential performance bottlenecks. (3) Video service providers desire to measure the performance of other competitors and industry benchmarks for improving system design. A theme common to these use cases is that testers need to measure QoE for third-party video streaming services, but typically do not have access to QoE analytical data from these services. Instead, they typically resort to blackbox measurements and perform active testing in different network environments.

2.3 Traffic encryption protocols

Existing video streaming analysis techniques [4, 3, 22] rely on parsing HTTP requests information in the network traffic, e.g., URLs, to identify the identity of downloaded chunks. However, with the adoption of encryption protocols, such information is encrypted and no longer available, making existing techniques no longer viable.

In this dissertation, we focus on the two dominant encryption protocols used in video streaming, i.e., HTTPS (e.g., Netflix [7]) and QUIC (e.g., Youtube [8]). QUIC is a UDP-based encrypted transport protocol with feature enhancements designed for better performance [9, 10]. HTTP-over-QUIC is being standardized as HTTP/3 [23] and attracted wide interest from the industry. These two protocols cover the vast majority of popular commercial streaming services, hence we focus on them in this paper.

As shown in Figure 2.3, HTTPS and QUIC both use Transport Layer Security(TLS) [24] to encrypt application layer data. Only very limited information can be

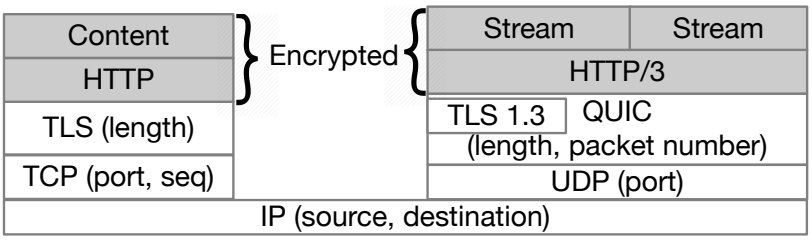

Figure 2.3: HTTPS/QUIC network stack and available information

obtained by in-network third-party monitoring, including IP packet timing, IP addresses, TCP/UDP port number, TLS record length in HTTPS and payload length in QUIC. Additionally, during the TLS handshake phase, the server domain name can be known from the Server Name Indication (SNI) extension sent by the client. However, all application payload information such as HTTP request URL and response cannot be observed, defeating traditional traffic analysis techniques.

It is worth mentioning that compared with HTTPS, QUIC has some unique properties which make its analysis even more challenging: (1) A retransmitted HTTPS packet can be detected from the SEQ number in the underlying TCP header. But for QUIC, each packet carries a new packet number, even for those carrying retransmitted data. This makes it difficult to identify retransmitted QUIC packets and therefore harder to get an accurate estimation of the payload size from network traffic size (§4.3.2). (2) QUIC supports multiplexing multiple streams for multiple objects at the same time within the same connection. This makes it more difficult to get individual object sizes transmitted on a QUIC connection.

Workarounds that decrypt encrypted traffic, including HTTPS MITM (Man-In-The-Middle) [25] proxies, rooting and client instrumentation etc., are fragile and hard to generalize, as various measures are increasingly adopted by the ecosystem to prevent such workarounds for security and privacy considerations. For example, apps on Android 7 onwards by default no longer trust user installed certificates [26], making MITM infeasible. There are no known working solutions to MITM QUIC traffic. Smartphone vendors are

making it much more difficult or even infeasible to root the devices (e.g., Samsung S5 onwards and iPhone).

CHAPTER III

Dissecting VOD Services for Cellular: Performance, Root Causes and Best Practices

In this chapter, we develop a general methodology that leverages common properties of commodity video-on-demand (VOD) apps to derive valuable insights into these proprietary services without access to the source code. We conduct a detailed measurement study of a wide cross-section of popular streaming VOD services to develop a holistic understanding of these services' design and performance. We identify performance issues and develop effective practical best practice solutions to mitigate these challenges.

3.1 Introduction

To build an ABR service, app developers have to determine a wide range of critical components spanning from the server to the client such as encoding scheme, adaptation logic, buffer management and network delivery scheme. The design involves (i) considering various service-specific business and technical factors, e.g., nature of content, device type, service type and customers' network performance, and (ii) making complex decisions and tradeoffs along multiple dimensions including efficiency, quality, and cost, and across layers (application, network) and different entities. It is thus challenging to achieve designs with good QoE properties, especially given the variable network conditions in cellular net-

works.

It is important to develop support for developers to navigate this complex design space. Understanding the performance and QoE implications of their design decisions helps developers make more informed and improved designs. Towards this goal, in this chapter, we conduct a detailed measurement study of 12 popular streaming VOD services to develop a holistic understanding of their respective designs and associated performances.

3.1.1 Contributions

Methodology. The closed, proprietary nature of commercial services makes it very challenging to gain deep visibility into their designs. Approaches like code disassembly suffer from limitations such as code obfuscation. Other approaches that either leverage app-specific features such as URL patterns [3, 27] or rely on deep modifications to the apps [5, 6] cannot be generally applied.

To address these challenges, we develop a general methodology that leverages common properties of commodity VOD apps to derive valuable insights into the proprietary VOD services without access to the source code (§3.2). Based on the observation that most popular VOD services adopt well-known ABR protocols, i.e., HTTP Live Streaming (HLS), SmoothStreaming (SS) and Dynamic Adaptive Streaming over HTTP (DASH) [12, 13, 14], we analyze the network traffic and extract useful information regarding the content download process, including timing, quality and size of video chunks downloaded. In addition, detailed analysis of the displayed User Interface (UI) elements for these apps reveals that they use common methods to inform users about the playback, including playback progress and stall events. We therefore develop techniques to extract this information. Correlating the network and UI, our approach is able to effectively extract critical video QoE metrics such as video quality, stall duration, initial delay and number of track switches. In addition, we can infer the apps' internal buffer state which is critical to gain insights into their behavior.

To derive insights into critical aspects of service design such as the adaptation logic, we craft targeted black-box experiments to stress-test the apps by emulating various network conditions and manipulating the communication between the client and server (e.g., by altering the manifest file). By analyzing the reaction of the apps, we are able to glean critical properties of their design.

In this chapter, we focus primarily on VOD services on the Android platform. However, the measurement methodologies we outline are generally applicable to other platforms (e.g., iOS) and services such as live streaming as they use the same standards (e.g., iOS AV Foundation uses HLS) and substantially similar approaches.

QoE issues and best practices. This study shows i) the different points in the design space adopted by popular services, ii) the different performance tradeoffs they entail. By examining the absolute and relative performances across different points in the design space, developers are able to get more insights into the implications of design decisions they make, and hopefully make more informed design decisions.

Our measurements cover both individual components across the end-to-end delivery path of ABR and their interactions. This is key to developing insights for better designs across components to realize an overall enhanced QoE. In contrast, different entities involved in the streaming system such as the content provider, ISP and app developers have traditionally possessed only partial views and optimized specific factors somewhat independently, based mainly on their limited views. This can sometimes lead to suboptimal performance as end-to-end QoE is ultimately determined by the interplay across all the different factors. Towards filling this gap, this cross-sectional study across different services develops unique insights by revealing QoE implications of different points in the design space, shedding light on industry best practices by comparing across different services and identifying outlier behaviors.

In this study, we observe interesting behaviors that span a wide range of design decisions and further identify a number of QoE-impacting issues and derive best practices for

improvement. We summarize some of the most interesting findings as follows.

- To improve quality, some apps perform Chunk Replacement (CR) – replacing a downloaded chunk with a fresh download for the same position in the video at a potentially different quality. We uncover inefficiencies with existing CR schemes that result in substantial additional data usage, identify root causes, and propose practical CR schemes that achieve better tradeoffs between QoE and data usage (§3.4.1).

- Some services use VBR encoding. However, when determining the next chunk to download, they do not account for the substantial size differences across different chunks in a track, which can be a factor of 2 or more. This can lead to suboptimal video QoE. We propose that apps should expose such chunk information to the adaptation logic and adopt an actual bitrate aware track selection algorithm (§3.4.2).

- Players typically wait until a minimum number of seconds (i.e., startup buffer duration) of video is fetched before initiating playback. We observe that some apps constantly stall at the beginning of playback when network bandwidth is relatively low, even with observed startup buffer values as other apps which don't exhibit this issue. Our evaluation suggests the need for an additional constraint on when playback should begin – a minimum threshold on the number of chunks downloaded (§3.4.3).

- Inadequate synchronization between multiple TCP connections and audio/video downloads can lead to QoE impairments (stalls) for some apps. This highlights the need for better coordination between the parallel download processes for better QoE (§3.3.2).

- A suboptimal buffer-based download strategy waits until the buffer is close to empty, before it restarts downloading. The corresponding app suffered more frequent stalls compared to the others with higher resuming thresholds. Increasing this *resuming threshold* would keep the buffer more occupied and be a practical way to reduce the

chances of stalls and provide the client extra headroom to adapt to transient network variability (§3.3.3.2).

3.2 Methodology

We explore a wide range of popular mobile VOD services, including Amazon Video, DIRECTV, FOX NOW, Hulu, HBO GO, HBO NOW, MAX GO, Netflix, NBC Sports, Showtime Anytime and XFINITY TV. In this chapter, we focus on 12 of these[1] covering a wide diversity of points in the design space, and study them in depth. These services individually have millions of app store downloads, and collectively span a wide range of content types including movies, TV shows and sports videos.

Understanding the design choices and characterizing the QoE of these proprietary video streaming services are challenging, as they do not readily expose such information. To address the challenge, we develop a general methodology to extract information from the traffic and app UI events. To capture important properties of the adaptation logic designs, we further enhance our methodology with carefully crafted black-box testing to stress test the players.

Figure 3.1 shows an overview of the methodology. The proxy between the server and the user device emulates various network conditions (§3.2.4) and extracts video chunk information from the traffic flow (§3.2.1). The on-device UI monitor monitors critical UI components (§3.2.2), such as the seekbar in the VOD apps that advances with the playback to inform users the playback progress and allow users to move to a new position in the video.

We combine information from the traffic analyzer and UI monitor to characterize QoE. In our methodology, the *Traffic Analyzer* obtains detailed chunk information, such as bitrate and duration etc, and therefore can be used to characterize video quality and track

[1]One of the services adopts both DASH and SmoothStreaming. As they have very different design on both server and client side, we treat them as two different services.

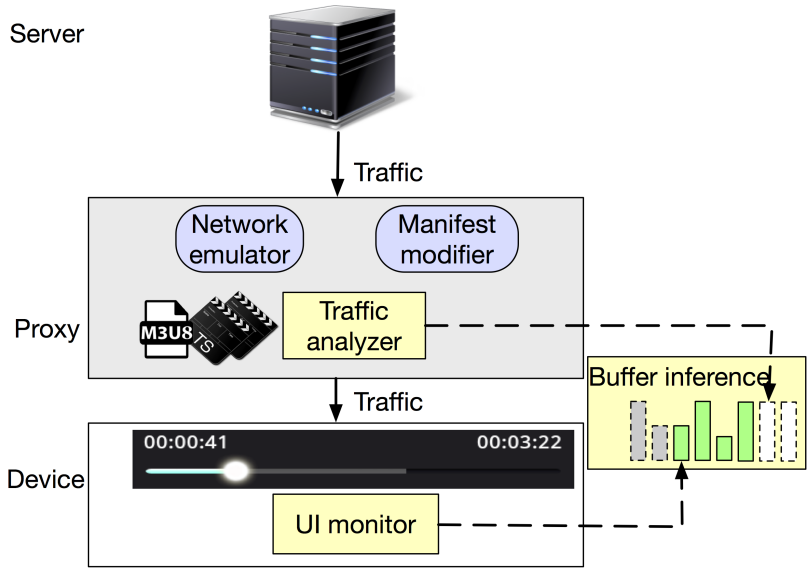

Figure 3.1: Methodology overview

switches. The *UI Monitor* on the device tracks the playback progress from the player's UI, and is able to characterize the stall duration and initial delay. Furthermore, combining the information from both the traffic analyzer and the UI monitor, we can infer the player buffer occupancy across time (§3.2.3), which critically allows us to reason about, identify and unveil underlying causes of many QoE issues.

To understand complex designs such as the adaptation logic, the proxy uses the *Network Emulator* and *Manifest Modifier* to conduct black-box testing. The network emulator performs traffic shaping to emulate various network conditions. By carefully designing the bandwidth profile, we are able to force players to react and understand their design. In some cases, we use the manifest modifier to modify the manifest from the server and observe players' behavior to understand how client side players utilize information from servers. For example, in §3.4.2 with the manifest modification, we are able to explore whether players take actual track bitrate information into consideration when performing track selection.

In the following, we provide details of components used in the measurement methodology.

3.2.1 Traffic analyzer

We develop the network traffic analyzer to perform MITM analysis on the proxy and extract manifest and chunk information from flows between the server and client.

We observe that all the studied apps adopted one or more among the three popular ABR techniques, i.e. HLS, DASH, and SmoothStreaming. We denote the four services that use DASH as *D1* to *D4*, another six that use HLS as *H1* to *H6*, the two services that use SmoothStreaming as *S1* and *S2*.

We specifically developed the traffic analyzer to be generally applied for all VOD services that adopt the three popular standard ABR techniques. The traffic analyzer parses the manifest based on the specification of the ABR protocols, and builds the mapping between HTTP requests and chunks. Since the three streaming protocol implementations have some different properties, the traffic analyzer extracts QoE information with different methodologies based on the protocol each service adopts. We shall mainly describe how the traffic analyzer works with the two most popular protocol implementations HLS and DASH.

HLS vs. DASH HTTP Live Streaming (HLS) [12] is a media streaming protocol proposed by Apple Inc. In HLS, a media presentation is described by a *Master Playlist*, which specifies the resolution, bitrate and the URL of corresponding *Media Playlist* of each track. The URL and duration of media chunks are specified in the *Media Playlist*. Each media chunk in HLS is a separate media file[2]. At the beginning of playback, the client downloads the *Master Playlist* to obtain information about each track. After it decides to download chunks from a certain track, it downloads the corresponding *Media Playlist* and gets the URI of each chunk.

Compared with HLS, the Dynamic Adaptive Streaming over HTTP (DASH) [13] is an international standard specifying formats to deliver media content using HTTP. Media content in DASH is described by the *Media Presentation Description (MPD)*, which specifies

[2]From version 4, HLS also supports using a sub-range of a resource as a media chunk. But none of our studied services use this feature.

each track's declared bitrate, chunk duration and URI etc. Each media chunk can be a separate media file or a sub-range of a larger file. The byte-range and duration of chunks may be directly described in the MPD. The MPD can also put such information in the *Chunk Index Box (sidx)* of each track and specify the URI of sidx. The sidx contains meta information about the track and is usually placed at the beginning of the media file.

To accommodate the differences across the ABR protocol and service variations, the traffic analyzer works as follows. It gets the bitrate of each track from the Master Playlist for HLS, and then extracts the URI and duration of each chunk from it. For DASH, it gets the bitrate of each track from the MPD, and generates the mapping of byte ranges to chunk information using different data sources for different apps. *D2*, *D3* and *D4* put such information into the sidx of each track, while *D1* directly encodes it in the MPD. *D3* encrypts the MPD file in application layer before sending it through the network. However, the sidx is not encrypted and we can still get chunk durations and sizes.

3.2.2 UI monitor

The UI monitor aims at exposing QoE metrics that can be obtained from the app UI on the client. Based on our exploration of all the VOD apps in our study, we identify the seekbar to be a commonly used UI element that indicates the playing progress, i.e. the position of displayed frames in the video in time.

We investigate how to robustly capture the seekbar information. As the UI appearance of the seekbar has a significant difference across different apps, we do not resort to image process techniques. Instead, we use the Xposed framework [28], an Android framework which enables hooking Android system calls without modifying apps, to log system calls from the apps to update the seekbar.

We find that despite the significant difference in visual appearance, the usage of the seekbar is similar across the services. During playback, the players update the status of the seekbar periodically using the Android API *ProgressBar.setProgress*. Thus, we obtain

information about playback progress and stall events from the API calls. The update may occur even when the seekbar is hidden on the screen. This methodology can be generally applied to apps that use the Android seekbar component regardless of the UI layout and visual appearance.

For the all apps we studied, the progress bar was updated at least every 1s and we can therefore get the current playing progress at at least 1s granularity.

3.2.3 Buffer inference

The client playback buffer status, including the occupancy and the information regarding chunks in the buffer, is crucial for characterizing the player's behavior. We infer the buffer occupancy by combining information from the downloading process and the playback process, collected by the traffic analyzer and UI monitor respectively: at any time, the difference between the downloading progress and playing progress should be the buffer occupancy, and the details, such as the bitrate, and duration of the chunks remaining in the buffer, can be extracted from the network traffic.

3.2.4 Network emulator

We use the Linux tool *tc* to control the available network bandwidth to the device across time to emulate various network conditions.

To understand designs such as the adaptation logic, we apply carefully designed network bandwidth profiles. For instance, to understand how players adapt to network bandwidth degradation, we design a bandwidth profile where the bandwidth stays high for a while and then suddenly drops to a low value. In addition, to identify QoE issues and develop best practices for cellular scenarios, it is important to compare the QoE of the different services in the context of real cellular networks. To enable repeatable experimentations and provide apples-to-apples comparisons between different services, we also replay multiple bandwidth traces from real cellular networks over WiFi in the lab for evaluating the services.

To collect real world bandwidth traces, we download a large file over the cellular network and record the throughput every second. We collect 14 bandwidth traces from real cellular network in various scenarios covering different movement patterns, signal strength and locations. We sort them based on their average bandwidth and denote them from Profile 1 to Profile 14 (see Figure 3.2).

We run each of the services with the 14 collected cellular bandwidth traces. Each experiment lasts for 10min and is repeated for several runs to eliminate temporary QoE issues caused by the external environment, e.g., transient server load.

3.3 Service characterization

The interactions between different components of each VOD service across multiple protocol layers on both the client and server side together ultimately determine the QoE. Using our methodology from §3.2, for each service, we identify critical design choices around three key components: the server, the transport layer protocols, and the client, and investigate their QoE implications. We summarize the various designs in Table 3.1.

Our measurements reveal a number of interesting QoE-impacting issues caused by the various design choices (Table 3.2). We shall present the design factors related to these issues in this section and dive deeper into 3 most interesting problems in §3.4.

3.3.1 Server design

At the server-side, the media is encoded into multiple tracks with different bitrates, with each track broken down into multiple chunks, each corresponding to a few seconds worth of video. Understanding these server-side settings is important as they have critical impact on the adaptation process and therefore the QoE.

For each service, we analyze the first 9 videos on the landing page which span different categories. We find that for all studied services, for the 9 videos in the same service, the settings are either identical or very similar. We select one of these videos as a representative

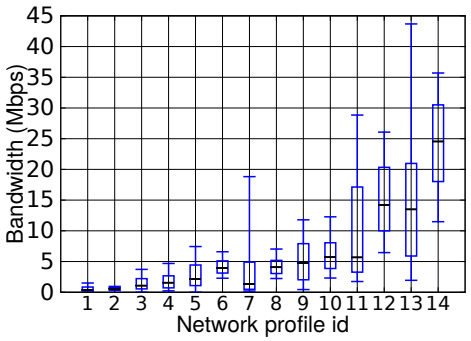

Figure 3.2: Collected cellular network bandwidth profiles

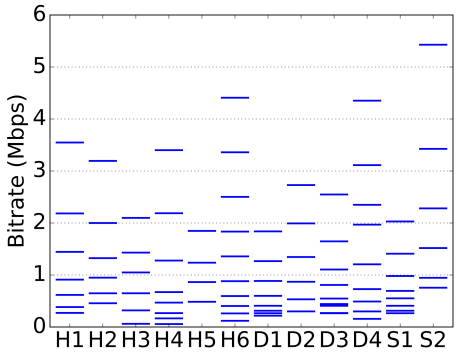

Figure 3.3: Declared bitrates of tracks for different services

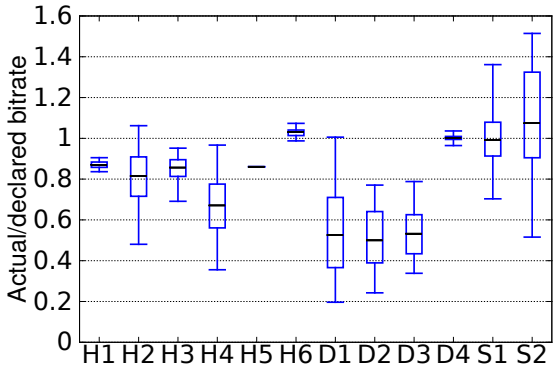

Figure 3.4: The distribution of actual bitrate normalized by declared bitrate

sample to further illustrate the design for each service.

Separate audio track. The server can either encode separate audio tracks or multiplex video and audio content in the same track. Using separate audio tracks decouples video and audio content, and gives a service more flexibility to accommodate different audio variants for the same video content, e.g., to use a different language or a different audio sample rate. We analyze a service's manifest to understand whether the service encodes separate audio tracks. We find that all the studied services that use HLS do not have separate audio tracks, while all services that use DASH or SmoothStreaming encode separate audio tracks.

Track bitrate setting. Track settings such as track count (number of tracks), the properties of the highest and lowest tracks, and the spacing (bitrate difference) between consecutive tracks all impact ABR adaptation and therefore the QoE. We obtain the track declared

Designs	H1	H2	H3	H4	H5	H6	D1	D2	D3	D4	S1	S2
Chunk duration (s)	4	2	9	9	6	10	5*	5	2	6	2	3*
Separate audio track	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y
Max #TCP	1	1	1	1	1	1	6	2	3	3	2	2
Persistent TCP	Y	N	N	Y	N	Y	Y	Y	Y	Y	Y	Y
Startup buffer (s)	8	8	9	9	12	10	15	5	8	6	16	6
Startup bitrate (Mbps)	0.63	1.33	1.05	0.47	1.85	0.88	0.41	0.30	0.40	0.67	1.35	0.76
Pausing threshold (s)	95	90	40	155	30	80	182	30	120	34	180	30
Resuming threshold (s)	85	84	30	135	20	70	178	25	90	15	175	4
Stability	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y
Aggressiveness	N	N	N	N	N	N	Y	N	Y	N	Y	N
Decrease buffer (s)	-	40	-	-	-	-	-	-	30	-	50	-

* The audio chunk duration of *D1* and *S2* is 2s.

Table 3.1: Design choices of 14 studied ABR services

bitrate from the manifest of each service[3].

The highest track represents the highest quality that a service provides. We find across the services the highest track has diverse bitrates from 2 Mbps to 5.5 Mbps. Note that the declared bitrate is not the only factor that determines video quality, as it also depends on other factors such as encoding efficiency.

The bitrate of the lowest track impacts the players' ability to sustain seamless playback under poor network conditions. Apple recommends that the lowest track should be below 192 kbps for cellular network [29]. However, the lowest track of 3 services is higher than 500 kbps and significantly increases the possibility of having stalls with slow network connection. For example, our evaluations show with the two lowest bandwidth profiles, *H5* always stalls for more than 10 s, while apps with lower bit-rate bottom tracks such as *D2* and *D3* do not have stalls under the same network conditions. *Because stalls severely impact QoE, we suggest setting the bitrate of the bottom track to be reasonably low for*

[3]This approach did not work for *D3* as the manifest is encrypted at the application layer and cannot be decrypted. Instead, we use the peak value of the actual chunk bitrates (which can be obtained by parsing the sidx) as the declared bitrate since other DASH services such *D1* and *D2* follow such practice

Design factors	**Problem**	**QoE impact**	**Affected service**
Track setting	The bitrate of lowest track is set high.	Frequent stalls	*H2*, *H5*, *S1*
Encoding scheme	Adaptation algorithms do not consider actual chunk bitrate.	Low video quality	*D2*
TCP utilization	Audio and video content downloading progress is out of sync when using multiple TCP connections.	Unexpected stalls	*D1*
TCP persistence	Players use non-persistent TCP connections.	Low video quality	*H2*, *H3*, *H5*
Download control	Players do not resume downloading chunks until the buffer is almost empty.	Frequent stalls	*S2*
Startup logic	Players start playback when only one chunk is downloaded.	Stall at the beginning	*H3*, *H4*, *H6*, *D2*, *D4*
Adaptation logic	The bitrate selection does not stabilize with constant bandwidth.	Extensive track switches	*D1*
	Players ramp down selected track with high buffer occupancy.	Low video quality	*H1*, *H4*, *H6*, *D1*
	Players can replace chunks in the buffer with ones of worse quality.	Waste data and low video quality	*H1*, *H4*

Table 3.2: Identified QoE-impacting issues in studied services

mobile networks.

Tracks inbetween the highest and lowest track need to be selected with proper inter-track spacing. If adjacent tracks are set too far apart, the client may often fall into situations where the available bandwidth can support streaming a higher quality track, but the player is constrained to fetch a much lower quality, due to the lack of choices. If adjacent tracks are set too close to each other, the video quality improves very little by switching to the next higher track and the higher track count unnecessarily increases server-encoding and storage overheads. Apple recommends adjacent bitrate to a factor of 1.5 to 2 apart [29]. All services we study are consistent with this guideline.

CBR/VBR Encoding. Services can use two types of video encoding scheme, i.e. Constant Bitrate (CBR) encoding which encodes all chunks into similar bitrates, and Variable Bitrate (VBR) encoding which can encode chunks with different bitrates based on scene complexity [30].

We examine the distribution of bitrates across chunks from the same track to determine the encoding. We get chunk duration information from the manifest. To get chunk sizes, for

services using DASH, we directly get chunk sizes from the byte range information provided by the manifest and sidx. For services using HLS and SmoothStreaming, we get the media URLs from the manifest file and use *curl* [31] to send HTTP HEAD requests to get the media size. We find that 3 services use CBR, while the others use VBR with significant different actual chunk bitrates in a single track. For example, the peak actual bitrate of *D1* is twice the average actual bitrate.

With VBR encoding, using a single declared bitrate to represent the required bandwidth is challenging. We look into how services set the declared bitrate. For the highest track of each service, we examine the distribution of actual chunk bitrates normalized by the declared bitrate. As shown in Figure 3.4, *S1* and *S2* set the declared bitrate around the average actual bitrate, while other services set the declared bitrate around the peak actual bitrate. We shall explore further in §3.4.2 the associated QoE implications.

Chunk duration. The setting of chunk duration involves complex tradeoffs [32]. A short chunk duration enables the client to make track selection decision in finer time granularity and adapt better to network bandwidth fluctuations, as chunks are the smallest unit to switch during bitrate adaptation. On the other side, a long chunk duration can help improve encoding efficiency and reduce the server load, as the number of requests required to download the same duration of video content reduces. We find significant differences in the chunk duration across the different services, ranging from 2s to as long as 10s (see Table 3.1). We leave a deeper analysis on characterizing the tradeoffs to future work. In addition, as we find later in §3.4.3, other factors such as startup buffer duration need to be set based on the chunk duration to ensure good QoE.

3.3.2 Transport layer design

In ABR, players use the HTTP/HTTPS protocol to retrieve chunks from the server. However, how the underlying transport layer protocols are utilized to deliver the media content depends on the service implementation. All the VOD services in this study use

TCP as the transport layer protocol.

TCP connection count and persistence. As illustrated in Table 3.1, all studied apps that adopt HLS use a single TCP connection to download chunks. 3 of these apps use non-persistent TCP connections and establish a new TCP connection for each download. This requires TCP handshakes between the client and server for each chunk and TCP needs to go through the slow start phase for each connection, degrading achievable throughput and increasing the potential of suboptimal QoE. *We suggest apps use persistent TCP connections to download chunks.* All apps that adopt DASH and SmoothStreaming use multiple TCP connections due to separated audio and video tracks. All these connections are persistent.

TCP connection utilization. Utilizing multiple TCP connections to download chunks in parallel brings new challenges. Some apps such as *D1* use each connection to fetch a different chunk. Since concurrent downloads share network resources, increasing the concurrency can slow down the download of individual chunks. This can be problematic in some situations (especially when either the buffer or bandwidth is low) by delaying the arrival of a chunk with a very close playback time, increasing the potential for stalls. Different from these apps, *D3* only downloads one chunk at a time. It splits each video chunk into multiple sub-chunk and schedules them on different connections. To achieve good QoE, the splitting point shall be carefully selected based on per connection bandwidth to ensure all sub-chunks arrive in similar time, as the whole chunk needs to be downloaded before it can be played. The above highlights that developing a good strategy to make efficient utilization of multiple TCP connections requires considerations of complex interactions between the transport layer and application layer behavior. We leave further exploration to future work.

When audio and video tracks are separate, the streaming of audio and video chunks are done separately. Since both are required to play any portion of the video, there should be adequate synchronization across the two download processes to ensure that both contents are available by the designated playback time of the chunk. Our evaluations reveal that

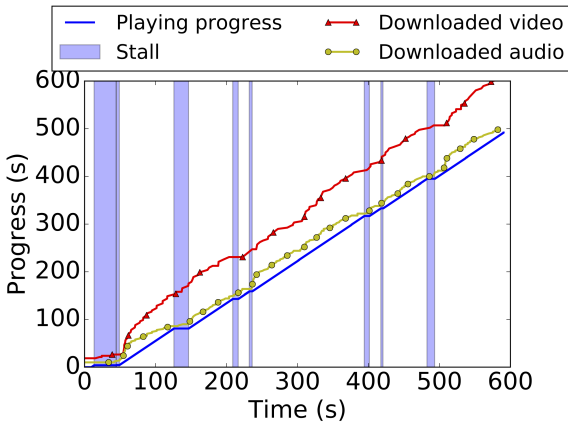

Figure 3.5: The downloading progress of video and audio content of *D1* is out of sync, causing unexpected stalls

uneven downloads for audio and video lead to clear QoE impairments for some apps. For example, we find *D1* uses multiple TCP connections to download audio and video content in parallel, but its download progresses for audio and video content can have significant differences, especially when the network bandwidth is low. For the two network profiles with the lowest average bandwidth, the average difference between video and audio downloading progress is 69.9 s and 52.5 s respectively. In the example shown in Figure 3.5, buffered video content is always more than audio content. When stalls occur, the buffer still contains around 100 s of video content. In this case, the stalls could have been avoided, without using any additional network resources, by just reusing some of the bandwidth for fetching more audio and a little bit less video. *We suggest ensuring better and tighter synchronization between audio and video downloads.*

3.3.3 Client-side design

The client player is a core component that impacts QoE by performing intelligent adaptation to varying network conditions. In this subsection we stress test the different players using the 14 bandwidth profiles collected from various scenarios. By comparing the behavior across different services under identical network conditions, we are able to identify interesting client behaviors and pinpoint potential QoE problems. More specifically, we

use black-box testing to study how players behave at startup, i.e. the startup logic, when they load the next chunk, i.e. the download control policy and what chunk they load, i.e. the adaptation logic.

3.3.3.1 Startup logic

We characterize two properties in the startup phase, startup buffer duration and startup track.

Startup buffer duration. At the beginning of a session, clients need to download a few chunks before starting playback. We denote the minimal buffer occupancy (in terms of number of seconds' worth of content) required before playback is initiated as the startup buffer duration.

Setting the startup buffer duration involves tradeoffs as a larger value can increase the initial delay experienced by the user (as it takes a longer time to download more of the video), but too small a value may lead to stalls soon after the playback. To understand how popular services configure the startup buffer, we run a series of experiments for each service. In each experiment we instrument the proxy to reject all chunk requests after the first n chunks. We gradually increase n and find the minimal n required for the player to start playback. The duration of these chunks is the startup buffer duration. As shown in Table 3.1, most apps set similar startup duration around 10s.

Startup track. The selection of the first chunk impacts users' first impression of the video quality. However, at the beginning the player does not have information about network conditions (eg., historical download bandwidths), making it challenging to determine the appropriate first chunk.

We examine the startup track of different players in practice. We find each app consistently selects the same track level across different runs. The startup bitrates across apps have high diversity. 4 apps start with a bitrate lower than 500 kbps, while another 4 apps set the startup bitrate higher than 1 Mbps. We shall further explore the QoE impact of startup

buffer duration and startup track in sec 3.4.3.

3.3.3.2 Download control

One important decision the client makes is determining when to download the next chunk. A naive strategy is to keep fetching chunks continuously, greedily building up the buffer to avoid stall events. However, this can be suboptimal as (1) it increases wasted data when users abort the session and (2) it may miss the opportunity to get a higher quality chunk if network condition improves in the future. We observe that, even under stable network conditions, all the apps exhibit periodic on-off download patterns. Combining with our buffer emulation, we find an app always pauses downloading when the buffer occupancy increases to a *pausing threshold*, and resumes downloading when the occupancy drops below another lower *resuming threshold*.

We set the network bandwidth to 10 Mbps, which is sufficient for the services to their respective highest tracks. We find 5 apps set the pausing threshold to be around 30 s, while other apps set it to be several minutes (Table 3.1). With a high pausing threshold, the player can maintain a high buffer occupancy to avoid future stall events. However, it may lead to more data wastage when users abort the playback. The different settings among services reflect different points in the decision space around this tradeoff.

The difference between the pausing and resuming threshold determines the network interface idle duration, and therefore affects network energy consumption. 8 apps set the two thresholds to be within 10 s of each other. As this is shorter than LTE RRC demotion timer [33], the cellular radio interface will stay in high energy mode during this entire pause in the download, leading to high energy consumption. *We suggest setting the difference of the two thresholds larger than LTE RRC demotion timer in order to save device energy.*

If either the pausing threshold or the resuming threshold is set too low, the player's ability to accommodate network variability will be greatly limited, leading to frequent stalls. We find that *S2* sets the pausing threshold to be only 4s and has a higher probability of in-

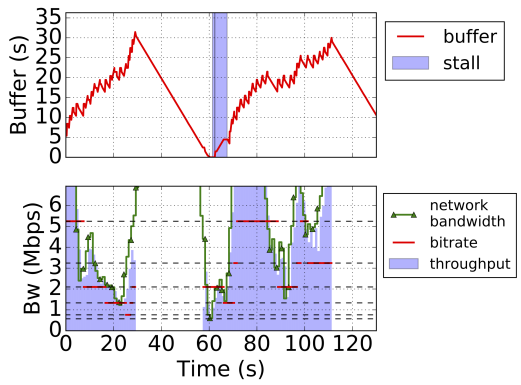

Figure 3.6: *S2* sets the resuming buffer to only 4s, leading to stalls

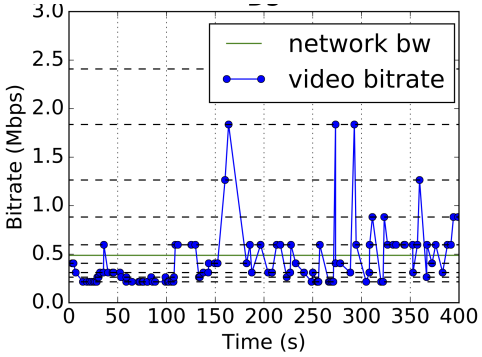

Figure 3.7: *D1* selected track is not stable even with constant bandwidth

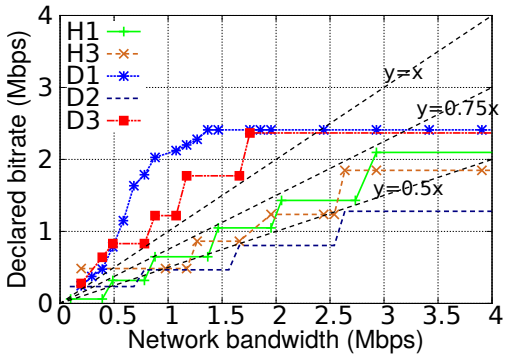

Figure 3.8: Selected declared bitrate given a constant bandwidth

curring stalls than other services under similar network conditions. As the example in Figure 3.6, at 25 s, the buffer occupancy of *S2* reaches to the pausing threshold and the player pauses downloading for around 30 s. When the player resumes downloading chunks, the buffer occupancy is only 4s and drains quickly due to temporary poor network condition. As stalls significantly degrade user experience, *we suggest setting both thresholds reasonably high to avoid stalls*. The exact value will depend on factors like the specific adaptation algorithm and is beyond the scope of this dissertation.

Next, we study the client adaptation logic. A good adaption logic should provide high average bitrate and reduce stall events and unnecessary track switches.

3.3.3.3 Track selection under stable network bandwidth

For each app, we run a series of experiments within each of which we emulate a specific stable network bandwidth for 10 min and examine the resulting track selection in the steady state. A good adaption logic should achieve an average bitrate similar to the network bandwidth without stalls and frequent track switches.

Stability. We find that the selected track of *D1* does not stabilize even with constant network bandwidth. As shown in Figure 3.7, the network bandwidth is constantly 500 kbps. However, *D1* frequently switches between different tracks and tries to improve the average actual bitrate to be close to network bandwidth. However, frequent switches, especially switches between non-consecutive tracks, can impair user experience. In contrast, the other apps all converge to a single track (different for each app) after the initial startup phase. *We suggest the adaptation logic avoid unnecessary track switches.*

Aggressiveness. We find that the track that different apps converge to under the same stable bandwidth condition has significant difference across different services. We term services that converge to a track with declared bandwidth closer to available bandwidth as more aggressive. We show a few examples in Figure 3.8. We find 3 apps are more aggressive and select tracks with bitrate no less than the available network bandwidth. The reason why they are able to stream tracks with a bitrate higher than available network bandwidth without stalls is that they use VBR encoding and the actual chunk bitrate is much lower than the declared bitrate. The other apps are relatively conservative and select tracks with declared bitrates no more than 75% of the available bandwidth. In particular, *D2* even select tracks with declared bitrates no more than 50% of available bandwidth.

3.3.3.4 Track adaptation with varying network bandwidths

To understand the adaptation to varying network condition, we run each app with a simple "step function" bandwidth profile, i.e. the network bandwidth first stays stable at one value and suddenly changes to another value. We test different combinations of

the initial and final bandwidth steps, and when the step occurs. The behavior across the different apps is summarized in Table 3.1.

Reaction to bandwidth increase. When bandwidth increases, all apps start to switch to a track with higher bitrate after a few chunks. In addition, we find some apps revisit earlier track switching decisions and redownload existing chunks in the buffer in an attempt to improve video quality. We further analyze this in §3.4.1.

Reaction to bandwidth decrease. When bandwidth decreases, apps eventually switch to a track with a lower bitrate.

A higher buffer pausing threshold enables more buffer buildup, which can help apps better absorb bandwidth changing events and defer the decision to select a lower track without the danger of stalls. However, among the 7 apps that have a large buffer pausing threshold (larger than 60 s), 4 apps always immediately switch to a low track when a bandwidth degradation is detected, even when the buffer occupancy is high, leading to sub-optimal QoE. In contrast, the other 3 apps set thresholds on buffer occupancy above which they do not switch to a lower track even if the available bandwidth reduces. *We suggest the adaptation logic takes buffer occupancy into consideration and utilizes the buffer to absorb network fluctuations.*

In summary, our measurements show popular VOD services make a number of different design choices and it is important to perform such cross-section study to better understand the current practices and their QoE implications.

3.4 QoE issues: deep dive

Some QoE impacting issues involve complex interactions between different factors. In this section, we explore in depth some key issues impacting the services we study, and use targeted black-box experiments to deduce their root causes. In addition, we further examine whether similar problems exist for ExoPlayer, an open source media player used by more than 10,000 apps [34] including YouTube [35], BBC [36], WhatsApp [37] and

Periscope [38] etc. Exoplayer therefore provides us a unique view of the underlying design decisions in a state-of-the-art ABR player being increasingly used as the base for many commercial systems. The insights and mitigation strategies we develop from this exploration can be broadly beneficial to the community for improving the QoE of VOD services.

3.4.1 Chunk replacement (CR)

Existing adaptation algorithms [39, 40, 41, 5] try to make intelligent decisions about track selection to achieve the best video quality while avoiding stall events. However, due to the fluctuation of network bandwidth in the mobile network, it is nearly impossible for the adaption logic to always make the perfect decision on selecting the most suitable bitrate in terms of the tradeoff between quality and smoothness. We observe that to mitigate the problem, when the network condition turns out to be better than predicted, some players will discard low quality chunks that are in the buffer but have not yet been played, and redownload these chunks using a higher quality track to improve user perceived video quality. We denote this behavior of discarding video chunks in the buffer and redownloading them with potentially different quality as *Chunk replacement* (CR).

While CR could potentially improve video quality, it does involve some complex tradeoffs. As chunks in the buffer are discarded and redownloaded, the additional downloads increase network data usage. In addition, CR uses up network bandwidth which could potentially have been used instead to download future chunks and may lead to quality degradation in the future. Existing works [42, 43, 44] find Youtube can perform extensive CR in a non-cellular setting. However, how common CR is used across popular services and the associated cost-benefit tradeoff for cellular networks is not well understood. We characterize this tradeoff for popular services, identify underlying causes of inefficiencies, and propose improvements.

3.4.1.1 Usage and QoE impact of CR for popular VOD apps

To understand the usage of CR by popular VOD apps, we run them with the 14 collected network bandwidth profiles. We analyze the track and index (the position of the chunk within the video track) of downloaded chunks. As chunks with the same index represent the same content, when multiple chunks with the same index are observed in the traffic, we confirm that the player performs CR. Among the players we study, we find *H1* and *H4* perform CR.

We conduct what-if analysis to characterize the extent of video quality improvement and additional data usage caused by CR. When CR occurs, among the chunks with the same index, only the last downloaded chunk is preserved in the buffer and all previous downloads are discarded. We confirm this using the buffer information in the logcat of *H1*. We emulate the case with no CR by keeping only the first downloaded chunk for each index in the trace and use it as a baseline comparison. Our analysis shows that CR as currently implemented by *H4*, does not work well. The findings are summarized as follows. *H1* shows similar trends.

- CR as currently implemented can significantly increase data usage. With 5 of the bandwidth profiles, the data consumption increases by more than 75%. The median data usage increase is 25.66%.
- For most bandwidth profiles, the video quality improves marginally. The median improvement in average bitrate across the 14 profiles is 3.66%.
- Interestingly, we find CR can even degrade video quality. For one profile, CR decreases the average bitrate by 4.09% and the duration for which tracks higher than 1 Mbps are streamed reduces by 3.08%.

The video quality degradation we observed with CR is surprising, as one would expect CR to only replace lower-bitrate chunks with higher-bitrate ones and therefore improve the average bitrate. Diving deeper, for each experimental run, we emulate the client buffer

over time. When a new chunk is downloaded, if the buffer already contains a chunk with the same index, we replace the previously buffered chunk with the newly downloaded one and compare their quality. A somewhat counter-intuitive finding is that the redownloaded chunks are not always of higher quality. Across the 14 bandwidth profiles, for all CR occurrences, on average respectively 21.31% and 6.50% of redownloaded chunks were of lower quality or same quality as the replaced chunk. These types of replacements are intuitively undesirable, as they use up network resources, but do not improve quality.

To understand why *H4* redownloads chunks with lower or equal quality, we analyze when and how *H4* performs CR. We make the following observations.

- **How CR is performed.** We find that after *H4* redownloads a chunk *seg*, it always redownloads all chunks that are in the buffer with indexes higher than *seg*. In other words, it performs CR for multiple chunks proactively and does not just replace a chunk in the middle of the buffer. In all CR occurrences across the 14 profiles, the 90^{th} percentile of the number of contiguously replaced chunks was 6 chunks.
- **When CR is triggered.** Whenever *H4* switches to a higher track, it always starts replacing some chunks in the buffer. For all runs with the 14 bandwidth profiles, each time CR occurs, we examine the quality of the first replaced chunk among the contiguous replaced ones. We find in 22.5% of CR cases, even the first redownloaded chunk had lower or equal quality compared with the one already in the buffer. This implies that *H4* may not properly consider the video quality of buffered chunks when performing CR.

We show an example of *H4* performing CR in Figure 3.9. At 150 s, *H4* switches from Track 3 to Track 4, which triggers CR. Instead of downloading the chunk corresponding to 580 s' of content, it goes back to redownload the chunk corresponding to 500 s' of content. In fact, that chunk was already downloaded at 85 s with a higher quality from Track 8. As the new downloaded chunk is from Track 4, this indicates CR with lower quality. Even worse, *H4* keeps redownloading all buffered chunks after that. This even causes a stall at 165 s, which otherwise could have been avoided.

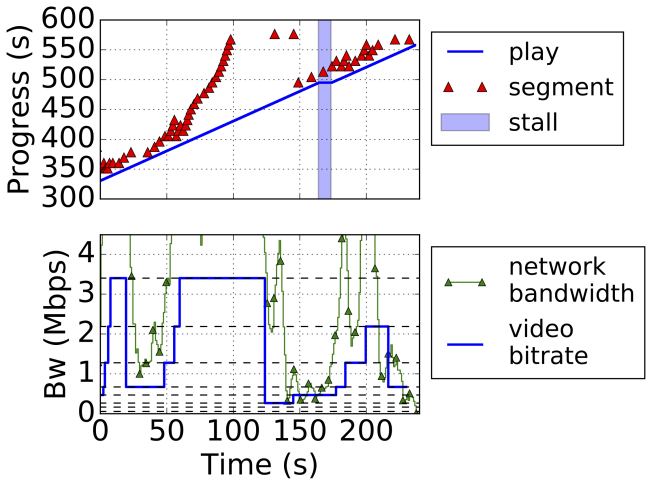

Figure 3.9: *H4* starts CR as long as it switches to a higher track and does not consider the track of chunks in the buffer

Deducing the root causes of such suboptimal CR design from commercial players such as *H4* is challenging due to their proprietary nature. To gain a deeper understanding into the underlying considerations behind CR policies, we next examine the CR design of the popular open-source ExoPlayer and its QoE implications.

3.4.1.2 CR analysis with ExoPlayer

We find that ExoPlayer version 1 uses CR and suffers from some similar issues as *H4*, i.e. it can also redownload chunks with lower or equal quality. To understand this, we first need to understand Exoplayer's adaptation logic. Before loading each chunk the track selection algorithm selects the track based on available network bandwidth and buffer occupancy. When it decides to select a higher track X than the last selected one Y, it initiates CR if the buffer occupancy is above a threshold value. It identifies the chunk with the smallest playback index in the buffer that is from a track lower than the track Y that ExoPlayer is about to select for the upcoming download. Beginning with that chunk, it discards all chunks with a higher index from the buffer. While this strategy guarantees that the first discarded chunk is replaced with higher quality one, the same does not hold for the

following chunks being replaced.

The root cause of these CR-related issues is that the player does not (i) make replacement decision for each chunk individually and (ii) limit CR to only replace chunks with higher quality. To answer the question *why players including H4 and ExoPlayer do not do this*, we study the ExoPlayer code and discover that it does not provide APIs to discard a single chunk in the middle of the buffer. Further investigation shows that this is caused by the underlying data structure design. For efficient memory management, ExoPlayer uses a double-ended queue to store chunks ordered by the playback index. Network activities put new chunks on one end, while the video renderer consumes chunks on the other end, which ensures that the memory can be efficiently recycled. Discarding a chunk in the middle is not supported, and thus to perform CR, the player has to discard and redownload all chunks with higher indexes than the first chosen one.

We find that the underlying data structure and CR logic remain the same in the latest Exoplayer version 2, but that CR is currently deactivated and marked for future activation. To understand the reasons behind ExoPlayer's approach to CR, we contacted its designers. They communicated that they were concerned about the additional complexity and less efficient memory allocation associated with allowing a single chunk in the middle to be discarded, and uncertainty about the benefits of CR. They were also concerned that allowing discard for a single chunk introduces some dependency between the track selection algorithm and other modules such as buffering policy.

3.4.1.3 CR Best practices and improvement evaluation

The response from ExoPlayer developers motivates us to look into how useful CR is when designed properly and whether it is worthwhile to implement it. Intuitively a proper CR logic should have the following properties.

- The logic considers replacing a chunk a time. Each chunk is replaced individually.
- Chunks can only be replaced by higher quality chunks.

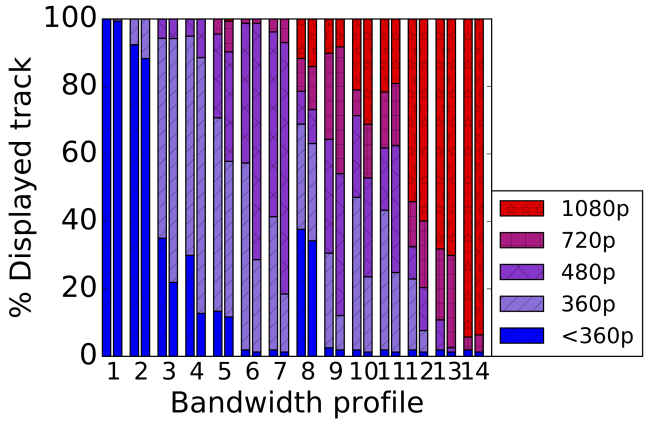

Figure 3.10: The displayed track percentage with/without CR. Each pair of bars are with the same network condition: left is without CR; right is with CR

- When buffer occupancy drops below a threshold, the player should stop performing more replacements and resume fetching future chunks to avoid the danger of stalls.

Changing the Exoplayer memory management implementation to enable discarding individual chunks is a non-trivial endeavor. Instead, for our evaluations, we modify the Exoplayer track selection logic to work with HTTP caching to achieve the same end-results. As an example, when chunks are discarded from the buffer, their track information is recorded. Later if the track selection logic determines to redownloaded them with quality no higher than the discarded ones, we change the track selection to select the track of the discarded chunk so that they can be recovered directly from the local cache on the device without sending traffic to the network. From the network perspective, this would have the same effect as not discarding the chunk.

To evaluate the QoE impact of the improved CR algorithm, we play a publicly available DASH stream [45] using the 14 collected real world bandwidth profiles. We find that, across the profiles, the median and 90^{th} percentile improvements in average bitrate are 11.6% and 20.9% respectively.

Subjective QoE studies(e.g., [46]) show that the video bitrate is not linearly propor-

tional to user QoE. Rather, increasing the bitrate when bitrate is low will cause a much sharper increase in user experience. But when bitrate is already high, further increasing the bitrate does not lead to significant additional QoE improvements. In other words, it is more important to reduce the duration of time that really low quality tracks are streamed. Thus we further break down the track distribution of displayed chunks without and with CR. As shown in Figure 3.10, when network bandwidth shows significant fluctuation and players have chances to switch between tracks, a properly designed CR strategy can greatly reduce the duration of streaming low tracks. For bandwidth profiles 3 and profile 4, the duration of streaming tracks lower than 360p reduces by 32.0% and 54.1% respectively. For profile 7 to profile 12, the duration of streaming tracks worse than 480p reduces significantly, reduction ranging from 30.6% to 64.0%.

CR increases video bitrate at the cost of increasing network data usage. For ExoPlayer with our improved CR algorithm, the median data usage increase across 14 profiles is 19.9%. For 5 profiles, the usage increases by more than 40%. Across the 14 profiles, the median amount of wasted data, i.e. data associated with downloading chunks that were later discarded, as a proportion of the total data usage was 10.8%. This implies that CR should be performed carefully for users with limited data plans.

To better make tradeoff between data usage and video quality improvement, we suggest only discarding chunks with low quality when data usage is a concern. As we shall see, discarding chunks with lower bitrate has a bigger impact on improving QoE and causes less waste data. To evaluate the proposed concept, we change the CR algorithm to only replace chunks no better than a threshold of 720p, and characterize the impact on data usage and video quality. We test with three profiles with the largest amount of waste data. Compared with the case of using no such threshold, for the 3 profiles, the wasted data reduced by 44% on average, while the proportion of time that streaming quality better than 720p was played stayed similar. The results therefore show that this is a promising direction for exploring practical CR schemes. Further work is needed in fine tuning the threshold selection.

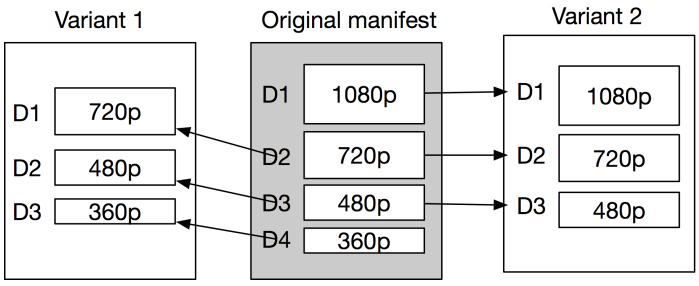

Figure 3.11: Illustration of the manifest modification in the experiment (D in the figure stands for declared bitrate)

In summary, we find proper usage of CR significantly reduces the duration of streaming tracks with poor quality and improves QoE. When making replacement decisions, players should consider each chunk individually and only replace chunk with higher quality. This requires underlying implementation to support discarding a chunk in the middle of the buffer. Due to the implementation complexities, creating a library that supports such operations can greatly benefit the app developer community.

3.4.2 Using declared vs. actual bitrate

Servers specify the declared bitrate for each track in the manifest as a proxy for its network resource needs, to help client players select proper tracks based on the network bandwidth. However, especially for VBR encoding which is increasingly popular, a single declared bitrate value cannot accurately reflect the actual bitrate across the video. For example, as shown in Figure 3.4, the declared bitrate of videos from *D2* can be twice of the average actual bitrate. Despite the potentially significant difference between the declared bitrate and actual bitrate, we find that the adaptation logic in some players such as *D2* relies purely on the declared bitrate to make track selection decisions, leading to suboptimal QoE.

Since *D2* uses DASH, it can in theory obtain actual chunk bitrates from chunk index boxes before playback. To verify whether *D2* takes the actual bitrate into consideration during track selection, we carefully design black-box testing experiments to reveal its internal logic. We modify the manifest to generate two variants with tracks of the same declared

bitrate but different actual bitrates. As illustrated in Figure 3.11, in variant 1 we shift the mapping between the declared bitrate and corresponding media files. We replace the media of each track to the one with the next lower quality level, while keeping the declared bitrate the same. In variant 2, we simply remove the lowest track and keep other tracks unchanged to keep the same number of tracks as variant 1. Thus, comparing these two variants, each track in variant 1 has the same declared bitrate as the track of the same level in variant 2, but the actual bitrate is the same as that of the next lower track in variant 2. We use *D2* to play the two variants using a series of constant available bandwidth profile. We observe that with the same bandwidth profile, the selected tracks for the two variants are always of the same level with the same declared bitrate. This suggests that it only considers the declared bitrate in its decision on which track to select next, else the player would select tracks with different levels for the two variants but with the same actual bitrate.

As the average actual bitrate of videos from *D2* is only half of declared bitrate, failure to consider the actual bitrate can lead to low bandwidth utilization, and thus deliver suboptimal QoE. We use *D2* to play original videos from its server with a stable 2 Mbps available bandwidth network profile. The average achieved throughput is only 33.7% of the available bandwidth in the steady phase. Such low bandwidth utilization indicates that *D2* could potentially stream higher quality video without causing stalls.

There are historical factors underlying the above behavior. HLS was the first widely adopted ABR streaming protocol for mobile apps, and some elements of its design meshed well with the needs of the predominant encoding being used at the time, i.e. CBR. For example, the HLS manifest uses a single declared bitrate value to describe the bandwidth requirements for each track. This is the only information available to the player's track selection logic regarding the bandwidth needs for a chunk in a track, before actually downloading the chunk. HLS requires setting this value to the peak value for any chunk in the track [12]. With CBR encoding, different chunks in a track have similar actual bitrates, making the declared bitrate a reasonable proxy for the actual resource needs. Adaptation

algorithms [47, 39, 40] therefore traditionally have depended on the declared bitrate to select tracks.

More recently, ABR services have been increasingly adopting VBR video encodings as shown in Figure 3.4, which offers a number of advantages over CBR in terms of improved video quality. However, different chunks in a VBR encoded track can have very different sizes due to factors such as different types of scenes and motion.

As the actual bitrate of different chunks in the same track can have significant variability, it becomes challenging to rely on a single declared bitrate value to represent all the chunks in a track. With VBR encoding, setting the declared bitrate to average actual bitrate can lead to stall events [5, 48]. On the other hand, setting the declared bitrate to the peak rate and using that as an estimate for a track's bandwidth (as *D2* seems to do) can lead to low bandwidth utilization and suboptimal video quality. The solution to the above is that (i) more granular chunk size information should be made available to the adaptation algorithm and (ii) the algorithm should utilize that information to make more informed decisions about track selection.

ABR protocols are moving towards making this granular information available, but challenges remain. DASH and newer versions of HLS support storing each chunk as a sub-range of a media file and expose the chunk byte ranges and durations in the manifest file which can be used to determine the actual bitrate for each chunk. HLS also supports reporting the average bitrate in the manifest along with the peak bitrate. Thus, in theory, an adaptation logic should now be able to utilize this information. However, we find that the information may still not be exposed to the adaptation algorithm. We checked the implementation of ExoPlayer version 2, the latest version. It provides a unified interface to expose information based on which an adaptation algorithm selects tracks. However, the interface only exposes limited information including track format, declared bitrate, buffer occupancy and bandwidth estimation. It does not expose the actual chunk-level bitrate information that is included in the manifest file. This implies that even though app develop-

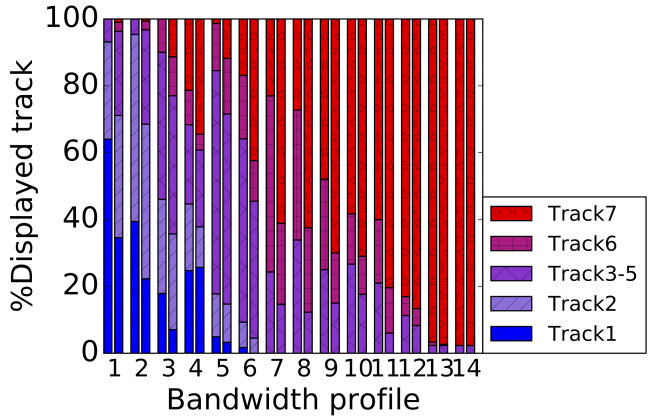

Figure 3.12: The displayed track percentage without/with considering actual chunk bitrate.

ers can implement customized sophisticated adaptation algorithms, in Exoplayer, currently they still can not leverage actual bitrate information to select tracks.

We next demonstrate that even a simple adaptation algorithm that considers actual chunk bitrates can improve QoE. We adjust ExoPlayer's default adaptation algorithm to select the track based on the actual chunk bitrate instead of the declared bitrate. To evaluate the performance, we VBR-encode the Sintel test video [49] and create an HLS stream consisting of 7 tracks. For each track we set the peak bitrate (and therefore the declared bitrate) to be twice of the average bitrate. We play the video both with the default adaptation algorithm and the modified algorithm that considers actual bitrate using the 14 collected network profiles.

We show the distribution of displayed track with and without considering actual chunk bitrate in Figure 3.12. Each pair of bars are with the same network condition. The left one the is the distribution only considering declared bitrate. The right one is the distribution considering actual bitrate. As we can see, when actual bitrate is considered, the duration of playing content with low quality reduces significantly. Across the 14 network profiles the median of average bitrate improvements is 10.22%. For the 3 profiles with the lowest

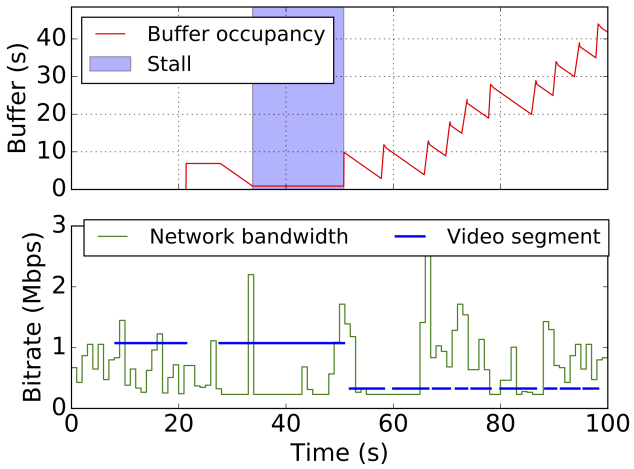

Figure 3.13: *H3* encounters a stall soon after starting to play

average bandwidth, the duration for which the lowest track is played reduces by more than 43.4% compared with the case of considering only the declared bitrate for track selection. Meanwhile, for all profiles we observe the stall duration stays the same, except for one profile, where it increases marginally from 10 s to 12 s. Note that the above results just illustrates the potential of using fine-grained chunk size information. The development of superior ABR adaptation schemes for VBR to make better tradeoff between video quality and stalls is a separate research topic in itself.

In summary, we suggest the services should expose actual chunk bitrate information to the adaptation logic, and that the adaptation logic should utilize such information to improve track selection.

3.4.3 Improving startup logic

We find that some apps such as *H3* always have stalls at the beginning of playback with certain network bandwidth profiles, while other apps do not have stalls under the same network condition. This indicates potential problems with the startup logic. As shown in Figure 3.13, *H3* first selects the track with a bitrate around 1 Mbps, which is higher than

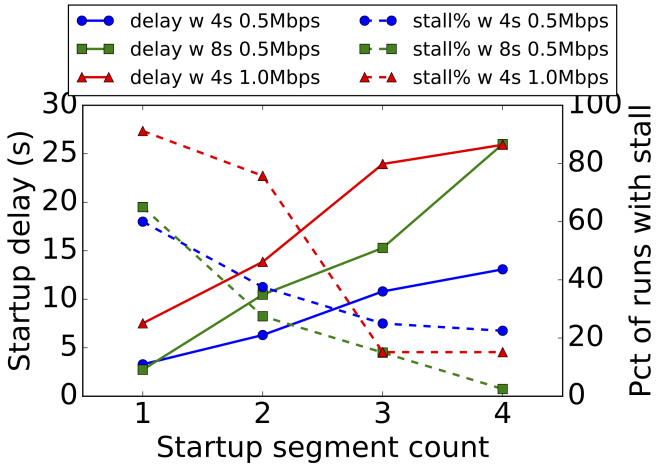

Figure 3.14: Startup delay and stall ratio with different chunk durations, startup tracks and startup chunk count.

the available network bandwidth . It starts playback after downloading the first chunk. For the second chunk it keeps selecting the same track as it may not yet have built up enough information about the actual network condition. As the network bandwidth is lower than selected bitrate, the buffer goes empty before the second chunk is downloaded, leading to a stall.

The investigation into the design difference between apps with and without QoE problems can give us hints on potential causes and solutions. We find *H3* and *H2* set similar startup buffer durations. However, *H2* does not encounter stalls with the same network, while *H3* does. Further analysis shows that each chunk of *H2* is only 2 s long and it downloads 4 chunks before starting playback, while the chunk duration for *H3* is 9 s and it starts playback once a single chunk is downloaded. Based on this observation, we hypothesize that the likelihood of having stalls at the beginning of playback does not only depend on the startup buffer duration in seconds but also on the number of chunks in the buffer. Using just 1 chunk as startup buffer introduces a high possibility to have stalls at the beginning of playback.

To validate this hypothesis and identify improvements, we characterize the tradeoff

brought by startup buffer duration setting between resulting startup delay and stall likelihood at the beginning of a video session, and propose suggestions for determining the setting empirically. We instrument ExoPlayer to set different startup buffer durations and play the Testcard stream, a publicly available DASH stream [45], with different chunk durations. We also configure the player to use different startup track settings. We calculate the average startup delay and the stall ratio, i.e. ratio of runs with stalls, with 50 bandwidth profiles of 1 min generated by dividing the lowest 5 10-min bandwidth profiles. As shown in Figure 3.14, we have the following observations.

- The stall ratio depends on both the startup buffer duration and the chunk duration. With the same startup buffer duration of 8 s, the stall ratio with chunks of 4 s is only 57.7% of the ratio with chunks of 8 s.
- Compared with using 1 chunk as startup buffer, using 2 or 3 chunks significantly reduces the stall possibility. In all video settings, the stall ratio for using 3 chunks is less than 41.7% of the stall ratio for 1 chunk.
- Using a higher bitrate track as startup track can significantly increase stall possibility, especially when startup buffer is only 1 chunk. With the startup buffer duration set to be 4s, when increasing the startup track bitrate from 0.5Mpbs to 1Mbps, the stall ratio increases from 60.0% to 91.1%.

Our findings suggest that apps should set the startup buffer duration to 2 to 3 chunks. We check the implementation of ExoPlayer. The startup buffer duration is a static value in seconds which developers can configure. *We suggest the player should enforce the startup buffer threshold both in terms of duration and chunk count. The startup track bitrate should also be relatively low to avoid stalls.* Similar suggestions can be also applied to the logic when the player recovers from stall events.

3.5 Summary

We conduct a detailed measurement study of a wide cross-section of 12 popular mobile streaming VOD services to develop a holistic understanding of their design and performance. Using carefully crafted measurements, we tease out important component designs across the end-end pipeline, including track settings, startup behavior, track switching behavior etc., and identify a number of QoE issues and their underlying causes. Using what-if-analysis, we develop best practice solutions to mitigate these challenges. By extending the understanding of how elements of service design impact QoE, our findings can help developers better navigate the design space and build mobile ABR services with improved performance.

In this chapter, we focus on VOD services. As live streaming uses same ABR protocols as VOD, the measurement methodology can be extended to study live streaming as well. We leave performing a detailed measurement study on live streaming to future work.

The proposed measurement methodology in this chapter relies on HTTP request information in the traffic. With the increasing adoption of traffic encryption, such a methodology is no longer applicable. In the next chapters, we explore techniques to study ABR adaptation behavior and measure streaming QoE even in the presence of traffic encryption.

CHAPTER IV

CSI: Inferring Mobile ABR Video Adaptation Behavior under HTTPS and QUIC

In the previous chapter, we analyze HTTP information in the network traffic to understand the adaptation behavior of ABR systems under different network conditions. However, end-to-end traffic encryption protocols such as HTTPS and QUIC are increasingly adopted by streaming services, defeating traditional traffic analysis approaches. To address this, in this chapter, we develop CSI (Chunk Sequence Inferencer), a general system supporting third-parties to perform active measurements and infer mobile ABR video adaptation behavior based on packet size and timing information still available in the encrypted traffic.

4.1 Introduction

As described in Chapter II, different mobile streaming systems adopt different ABR strategies and tradeoffs. Their clients exhibit substantially different adaptation behaviors even under the same network conditions, and they keep evolving over time.

In this chapter, we develop a novel, general system CSI (Chunk Sequence Inferencer) that provides the capability to independently conduct active measurements and infer the adaptation behavior and delivered QoE of third party commercial mobile video services,

for the increasingly common but challenging use case where these services use encrypted (HTTPS/QUIC) communications between the client and server. For a specific streaming service and video asset, CSI streams the video under specific network conditions of interest (§4.4). A key feature is that CSI analyzes the associated network traffic to infer (1) the *identity* of each downloaded chunk, i.e., the index, the track it belongs to, whether it is an audio or video chunk and (2) the time when each chunk is downloaded. From such information, QoE information including displayed video quality and variance, and stall occurrences can be further analyzed.

CSI should be particularly useful to a wide range of third-party entities who desire to independently evaluate and understand the adaptation behavior of popular mobile video services. Without a tool like CSI, it is extremely challenging to study the adaptation behavior of commercial streaming services: in addition to the complex ABR logic, these commercial systems are proprietary and close source, and very little is known about their inner workings.

One key challenge CSI addresses is that popular streaming apps [7, 50] are increasingly adopting end-to-end encryption protocols like HTTPS and QUIC and encrypt packet payloads. Existing active measurement approaches depend on being able to extract application level information from the network traffic between the client and server, and determine the identity of each downloaded chunk using information in the corresponding HTTP request URL ([43, 11, 4, 3]). This approach is no longer viable in the presence of traffic encryption as all higher level information, including the request URL information is encrypted. Even workarounds such as Man-In-The-Middle (MITM [25]) proxies are becoming increasingly less effective (see §4.2). Machine learning based proposals [17, 51, 52, 53, 54, 55] also have various limitations, including requiring labeled QoE data to train models, which is hard to obtain in general (§4.2).

To address this challenge, CSI infers chunk identities as follows (§4.3). It leverages the key insight that for common traffic encryption, the data volume sent over the network is

similar to the corresponding object size before encryption. Before running the active measurement, CSI obtains the sizes of all chunks across all tracks for the target test video. After running the video streaming session, it analyzes the IP-level information still available in the encrypted traffic - specifically the packet sizes and timing information. Conceptually, it analyzes the encrypted traffic to first infer the packets corresponding to client requests for chunks, and then estimates the size of each downloaded chunk based on the traffic downloaded between consecutive requests. It then uses the chunk size as a fingerprint to identify the corresponding playback index and track of each downloaded chunk.

Our key contributions are:

- Foundational insights (§4.3). We perform extensive measurements and develop two key insights that demonstrate the feasibility of inferring chunk identities from encrypted traffic. (1) Downloaded object size can be accurately inferred from associated encrypted packets (§4.3.2). (2) For the increasingly common Variable Bitrate (VBR) encoding, even with a relatively short sequence of chunk sizes, consisting of a mixture of chunks from different tracks, the identity of each chunk in the sequence can still be identified with high accuracy (§4.3.3).

- Design of CSI (§4.4). CSI enables automated and repeated active measurements to understand the adaptation behavior and delivered QoE of commercial mobile video streaming under various network conditions, which is useful for situations that requires large-scale testing. It automates the measurement process including performing network emulation, player UI instrumentation, data collection and analysis.

- Inference algorithm that is a key component of CSI (§4.5). To efficiently identify the chunk sequence that matches size information from the traffic, CSI formulates the matching problem as a shortest path graph search. CSI also addresses additional challenges introduced by QUIC's unique properties, such as the stream multiplexing feature.

- Evaluation (§4.6). We perform extensive evaluations and demonstrate that CSI achieves high inferencing accuracy (1) across different chunk size variability across 6 popular services (2) across ABR systems with different designs. In addition, the analysis is fast, typically taking only a few seconds to analyze a 10 min long video session.

- Use case (§4.7). We use Hulu as an example service and illustrate how CSI can be used in practice to help understand the QoE implications of parameter settings in token-bucket based traffic shaping policies and derive optimized shaping policies for mobile networks (§4.7).

We are working with the developers of a popular mobile video streaming analysis toolkit from a large mobile network operator to integrate CSI into the toolkit. The toolkit is widely used in the industry and a new version including CSI is being prepared for public release.

The design of CSI was primarily motivated by the need to analyze complex adaptation behavior of closed-source mobile apps in highly variable network conditions typical of cellular networks. However, CSI can also be used for less challenging scenarios such as more stable broadband home networks and web-based ABR streaming.

4.2 Design requirements

We describe the requirements in designing CSI for analyzing ABR video streaming in the presence of traffic encryption and where existing work fails.

- The system should require minimal knowledge on the adaptation behavior or QoE about the tested service, as that is what the system tries to measure. [17, 51, 52, 53, 54, 55] use network traffic data with labeled video QoE to train ML models to monitor video QoE from encrypted network traffic charateristics such as throughput. However, such labeled QoE data is hard to obtain without active measurement.

- The system should be robust and generalizable. It should (1) only use information generally available in the encrypted network traffic, (2) leverage common characteristics in video streaming independent of specific track adaptation algorithms. Workarounds that decrypt encrypted traffic, including HTTPS MITM [25] proxies, rooting and client instrumentation etc., are fragile and hard to generalize, as various measures are increasingly adopted by the ecosystem to prevent such workarounds for security and privacy considerations. For example, apps on Android 7 onwards by default no longer trust user installed certificates [26], making MITM infeasible. There is no known working solutions to MITM QUIC traffic. Smartphone vendors are making it much more difficult or even infeasible to root the devices (e.g., Samsung S5 onwards and iPhone). The measurement system should work in cases where such workarounds are not applicable. [56] uses very specific assumptions on ABR adaptation logic to estimate QoE metrics, which do not hold in general.

- The system should provide fine-grained information on how players adapt to various network conditions and resulting QoE. This is essential for many use cases such as performing QoE diagnosis and deriving better designs. Previous mentioned machine learning approaches only provide coarse-grained binary classification or qualitative labels on the QoE, which is insufficient for use cases such as deriving better traffic policies or diagnosing QoE issues.

4.3 CSI overview: using sizes as fingerprint

In this section, we describe the high-level approach of CSI and the key insights that enable it. We will describe practical challenges and more CSI design details later in §4.4 and §4.5.

In the following, we denote the chunk corresponding to the i^{th} request as C_i, its media

Symbol	Description
C_i	Chunk corresponding to the i^{th} request
M_i, T_i, I_i, S_i	Media type, track, index and size of C_i
$\tilde{S}_i$	The estimated size of C_i based on traffic
S_{ak}	The size of an audio chunk in the kth audio track

Table 4.1: The notation used in this chapter

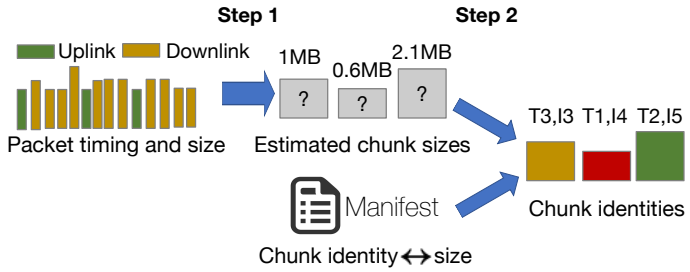

Figure 4.1: Proposed analysis approach for encrypted traffic (T: track, I: index, Tx,Iy means the y^{th} chunk in the x^{th} track.)

type (audio or video[1]), track, index and size as M_i, T_i, I_i and S_i respectively (in Table 4.1).

4.3.1 High-level solution

In advance of running the actual streaming experiment, CSI gathers sizes of all chunks from all tracks of the test video. Note this only need to be performed once for each test video. Then CSI streams the tested video on the target service in certain network conditions. During the test, CSI captures encrypted traffic going through the device. After the test, it infers downloaded chunk identities from encrypted traffic following below 2 steps.

Step 1. Combining packet sizes and timing, CSI identifies (i) packets corresponding to HTTP requests from the player to the server in the upstream direction, and (ii) the set of packets in the downstream direction that correspond to the response (i.e., chunk) for each request. From this CSI estimates sizes of downloaded chunks. We denote the estimated chunk sizes as $(\tilde{S}_i)_{i=1}^{n}$ (they may have inaccuracy compared with actual chunk sizes

[1]Some ABR services multiplex audio and video content together and each chunk contains both video and associated audio content. For such services, we consider all chunks as video chunks.

$(S_i)_{i=1}^n$).

Step 2. Given the estimated downloaded chunk sizes $(\tilde{S}_i)_{i=1}^n$, CSI identifies the chunk sequence $(C_i)_{i=1}^n$ (where different chunks can be from different tracks) whose size sequence $(S_i)_{i=1}^n$ most closely matches $(\tilde{S}_i)_{i=1}^n$ as the likely set of chunk downloaded in the session.

The feasibility of such an inference approach depends on two key insights. (1) For encrypted traffic, given a group of packets associated with downloading a chunk, we can estimate chunk sizes with relatively high accuracy. (2) Given the achievable accuracy of chunk size estimation, we can accurately identify the chunk identity based on the size. Works [57, 58, 59, 60, 61, 62, 63, 64, 65] infer details like which website the user visited, which video is played and which app the user used etc. from HTTPS traffic. However, there is no study on the feasibility of inferring the downloaded chunks during ABR video streaming. In addition, existing works focus on HTTPS and do not examine QUIC. In the next, we perform measurements to demonstrate the insights.

4.3.2 Accuracy of chunk size estimation

We first investigate estimating chunk size from a set of encrypted packets associated with downloading the chunk.

We try to reduce potential inaccuracy as much as possible. For HTTPS, we remove retransmitted packets based on SEQ number in underlying TCP header. We then estimate the chunk size as the sum of the TLS payload lengths in the remaining packets (excluding IP/TCP/TLS headers in Figure 2.3). For QUIC, we estimate the chunk size as the sum of the QUIC payload lengths in all packets(excluding IP/UDP/QUIC headers).

To evaluate the accuracy of the size estimation, we build an Android app with Cronet [66], an HTTP library that supports both HTTPS and QUIC, and download chunks with sizes ranging from 50KB to 1MB using the two protocols. We capture associated network traffic and estimate download file sizes following the above steps. We repeat the experiment in different mobile network environments and each object is downloaded 100

times in total.

We compare the estimated size $\tilde{S}_i$ with the ground truth S_i to measure the estimation accuracy. We find the estimation is quite accurate for both protocols: the maximal error is only 1% and 5% for HTTPS and QUIC respectively. For HTTPS, this error is mainly caused by potential TLS overheads. For QUIC, the error rate is slightly higher due to (1) we cannot differentiate retransmitted QUIC packets based on non-encrypted packet headers, (2) QUIC is built on top of UDP and implements signaling such as congestion control and flow control within the encrypted QUIC payload which we also cannot distinguish.

Based on the measurements, we have the following relation between $\tilde{S}_i$ and S_i.

$$S_i \leq \tilde{S}_i \leq (1+k)S_i \qquad \text{(Property (1))}$$

k represents the maximal estimation error (1% for HTTPS and 5% for QUIC).

[67, 58, 59, 60, 62, 63, 64] also show that traffic analysis can be performed to infer HTTPS payload size, as it does not use TLS padding [68, 69] to obfuscate payload size, likely due to significant associated data overhead and network resource inefficiencies [70]. Our results validate the observation for HTTPS and suggest a similar conclusion for QUIC.

4.3.3 Accuracy of chunk identification

Next, we perform analysis on chunk size variability of video streaming services to understand whether it is possible to use estimated chunk sizes (given the achievable estimation accuracy in §4.3.2) to accurately identify downloaded chunks among all encoded chunks.

Traditionally streaming services mainly adopted Constant Bitrate (CBR) encoding and encode the video with fixed bitrates for each track. The track of downloaded chunks can be trivially identified based on their sizes, as each track has a distinct chunk size. More recently services increasingly adopt VBR encoding [71, 59, 11, 72] due to its higher encoding efficiency [73]. The encoder allocates higher bitrates to encode the chunks corresponding to complex scenes and lower bitrates to chunks corresponding to simpler scenes. This re-

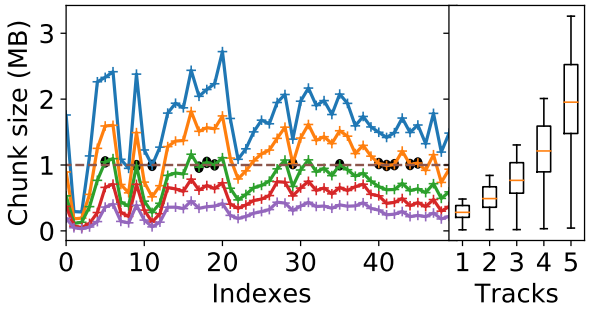

Figure 4.2: Chunk sizes of a Youtube video (PASR 2.6).

sults in size variance even for chunks in the same track. As an example, we plot the chunk sizes of a popular Youtube video (*Adele-Hello*) in Figure 4.2. In the video, some chunks from track 3 can even have similar sizes with chunks from track 5. This makes it challenging to identify chunk tracks based on its size. However, such size diversity also gives us opportunity to identify each chunk: assuming each chunk has a different size, we can build a unique mapping between the size and chunk identity.

To evaluate the feasibility of using chunk sizes as a fingerprint to identify chunks with different VBR encodings, we create videos with different size variability and perform analysis. We define *PASR* (peak-to-average size ratio) to be the ratio between the 95^{th} percentile chunk size and the average chunk size within a track. We use FFmpeg to encode the commonly used Big Buck Bunny (BBB) test video [74] into 10 different ABR streams (each with a ladder of tracks) with PASR values ranging from 1.1 to 2.0 (increasing at 0.1). For each stream, we encode the video into six tracks with resolutions ranging from 144p to 1080p following the setting of Netflix [75]. When encoding the tracks, we follow the three-pass encoding procedure in [76] and configure parameters `-maxrate` and `-b:v` to achieve desired PASR in each setting. We then use MP4Box [77] to split each track into 5-sec chunks.

Q1: Can we uniquely determine the identity of a single chunk given its estimated size (considering potential inaccuracy in size estimation)?

If chunk sizes can be accurately obtained without any error, two chunks C_i and C_j are

indistinguishable based on the size information only when their sizes $S_i = S_j$. When there is potential inaccuracy in size estimation (which is the case for encrypted traffic), assuming the maximum error in size estimation is k, two chunks C_i and C_j are indistinguishable based on the size information if $\frac{S_j}{1+k} \leq S_i \leq (1+k)S_j$, as they can be potentially estimated to have the same size. We define such two chunks to be *similar* with threshold k. Recall that k is 1% for HTTPS and 5% for QUIC. We define a chunk to be *unique* if there is no other chunk similar to it in the video.

We find that even with a k of 1%, all encoded videos have less than 0.1% of unique chunks regardless of the encoding PASR. In other words, 99.9% of chunks have at least 1 other chunk with similar sizes. With a relatively low PASR, there is less variability in sizes of chunks in the same track, and therefore multiple chunks in the same track are more likely to have similar sizes. With a relatively high PASR, sizes in the same track span larger ranges, and thus chunk sizes in different tracks are more likely to overlap. In either case, it is hard to guarantee that a single chunk has a unique size among all the chunks across all tracks in the video. Taking the the video in Figure 4.2 as an example, we highlight chunks with size 1MB ($k = 1\%$). We can see that multiple chunks in both the same track and different tracks have similar sizes.

The above analysis shows that given a certain estimated size for a single chunk, it is very challenging to uniquely identify the corresponding chunk regardless of the encoding.
Solution. To reduce the ambiguity in chunk identification, we leverage one common property during ABR streaming: the indexes (i.e., playback positions in the track) of the downloaded chunks should grow contiguously.

$$I_i = I_{i-1} + 1 \qquad \text{(Property (2))}$$

With this constraint, we can combine the estimated size information of multiple consecutive chunks to jointly determine their identities. Note that we do not assume I_1 to be 1 as the

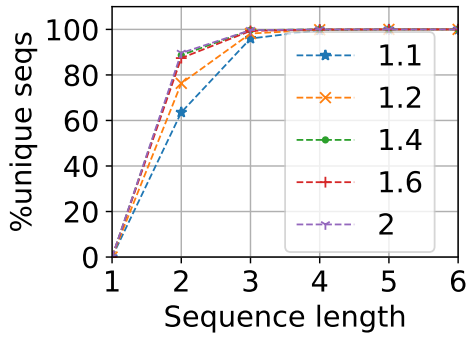

Figure 4.3: Relation between chunk sequence length and uniqueness (HTTPS $k = 1\%$)

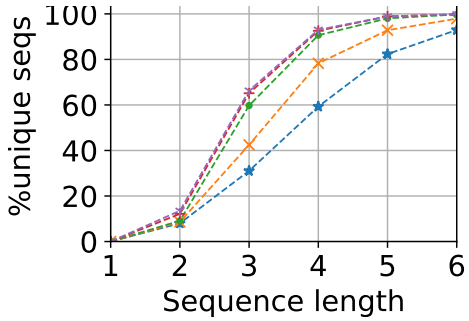

Figure 4.4: Relation between chunk sequence length and uniqueness (QUIC $k = 5\%$)

playback might not start from the beginning of the video in the test (e.g., resuming from the end point of last test).

Q2: Can we uniquely determine the chunk identities given the estimated sizes of multiple consecutive chunks? We denote a *chunk sequence* as a series of chunks $(C_i)_{i=1}^{n}$ where the indexes of the chunks grow contiguously (they can be from different tracks). We consider two chunk sequences $(C_{1i})_{i=1}^{n}$ and $(C_{2i})_{i=1}^{n}$ to be *similar* if every pair of chunks C_{1i} and C_{2i} in these two sequences are *similar*. A sequence is *unique* if it does not have a similar sequence.

The total number of chunk sequences increases exponentially when the sequence length increases[2]. However, as shown in Figure 4.3 and 4.4, for all VBR encodings, the percentage of unique sequences decreases dramatically when the sequence length increases. Even with a PASR as low as 1.1, 99.9% of 3-chunk sequences are unique with k of 1%, 92.6% of 6-chunk sequences are unique with k of 5%. This implies that for many video services, even those with relatively small size variance in a track, given a sequence of only a small number of contiguous chunks, we can uniquely determine the identity of each chunk in the sequence with a high probability.

[2]Each sequence is uniquely determined by the index of the first chunk and the tracks of all chunks in the sequence

In summary, our measurement results demonstrate the feasibility of the proposed approach. We will perform similar analysis on the encoding of commercial streaming services later in §4.6 to validate the generality of above conclusion. We next build CSI as a practical system based on these insights.

4.4 CSI system design

Further to the description in §4.3.1, we present concrete system design of CSI (Chunk Sequence Inferencer) in Figure 4.5. The key components include the *controller*, the *gateway* and the *mobile device*. The controller automates the measurement and analyzes collected data to infer streaming behavior of mobile video services. The gateway performs traffic shaping to emulate different network conditions and collects traffic passing through. CSI also leverages *web browsers* to facilitate collecting information on encoding chunk sizes of the tested video. CSI works following below procedures.

4.4.1 Collecting chunk sizes from all tracks

Recall in §4.3.1, in advance of running the actual streaming experiment, CSI need to gather sizes of all chunks from all tracks of the test video. Such information is essential for later chunk identity inference based on their estimated sizes. CSI gets such information from the manifest which specifies information on encoding tracks and chunks (see Figure 2.1). Clients downloads the manifest at the beginning of playback to get necessary information to fetch chunks. Many manifests directly specify the sizes of all chunks[11, 72]. CSI parses the manifest to get the size information. In other cases, manifests only provide URLs of all chunks. CSI sends HTTP HEAD requests to query chunk sizes given the chunk URL information in the manifest. Note this only need to be performed once for each test video.

Depending on the streaming service, CSI obtain the manifest using one of the following approaches.

Approach 1: If the streaming service supports browser-based streaming (most commercial services do), CSI plays the tested video using a browser instead of mobile apps to get the manifest. The reason is that browsers provide developer tools (e.g., Chrome [78], Firefox [79]) to inspect all network activities including encrypted traffic. Unlike browsers, mobile apps do not provide such access, but they typically share the server-side setting (e.g., track encoding and chunk sizes) with browser-based players. Thus, even though CSI focuses on revealing the adaptation behavior of mobile apps which are the predominant vehicle for consuming video content on mobile devices [80, 81, 82], CSI leverages the browser to glean the manifest.

Note that the browser-based streaming here is used just for getting the manifest metadata, and we cannot use ABR testing on web to understand the behavior of native apps, as Browser-based streaming are likely to have very different implementations with different client adaptation logic designs (e.g., Java-based player libraries such as ExoPlayer [48] on Android, AV Foundation framework on iOS, and Javascript-based libraries such as Shaka Player [83] in browsers) and thus different performance characteristics. As an illustration, we stream a video on Youtube with a stable network bandwidth of 1 Mbps on and observe the player behavior using information from stats-for-nerds displayed on the screen [84]. We find Youtube on different platforms selects tracks with different resolution (web 480p, Android and iOS 360p) and has different maximum buffer duration (web 90 s, iOS 60 s, Android 120s), which could lead to very different performance. Therefore, we only make use of the increased access offered by the browser to get the manifest file for the video.

Approach 2: An alternate approach is to glean the manifest from native apps using protocols (or platforms or devices) where intercepting encrypted connections is feasible (§2.3). For example, there is no known technique to intercept QUIC connections. But when QUIC traffic is blocked, players typically fall back to providing service on HTTPS. Thus, CSI blocks QUIC traffic on the gateway and uses MITM approaches to intercept HTTPS connections to get the manifest if feasible, and use the manifest information to help analyze

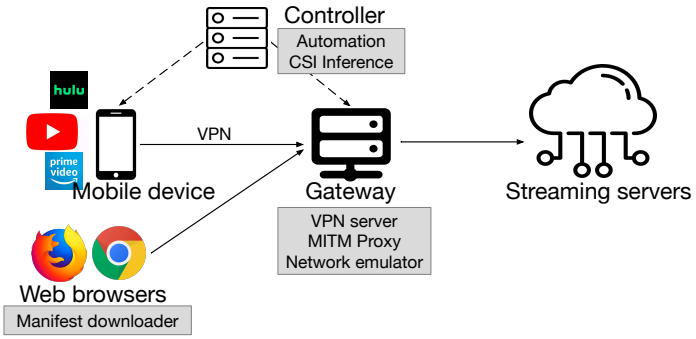

Figure 4.5: The system architecture of CSI

the streaming behavior over QUIC, which can have substantially different performance [9]. Another example is that newer Android systems no longer trust user installed certificates (see §4.2) and prevent MITM. CSI could perform MITM interception on older systems to get the manifest, then use it to study the adaptation behavior of newer versions of players on latest Android systems.

4.4.2 Streaming video and collecting data

After per-chunk size information is collected for all tracks, CSI next conducts streaming experiments for the target service for which testers desire to understand the adaptation behavior and quantify QoE. CSI leverages the UI Automator testing framework [85] to interact with the video player and play the tested video. The controller also controls the gateway to perform traffic shaping using the Linux tool *tc* according to the bandwidth trace provided by testers. The traffic from the device is routed through the gateway using VPN. The gateway captures passing-through encrypted traffic (and therefore packet size and timing information).

For some services, CSI also collects the identities of displayed chunks by analyzing the device screen. This is optional and helps improve the inference accuracy of CSI. We will perform evaluation to show the inference accuracy with/without such information in §4.6. The information on displayed chunks can be collected in different approaches. For

example, the stats-for-nerds player overlay option in YouTube shows the resolution of the currently displayed track [84]. Netflix has test patterns that encode the track number and frame number as an overlay on the videos [86]. Combining with OCR techniques to extract such information from the screen, CSI gleans information on when and what chunks are displayed on the device. CSI will later infer when each displayed chunk was actually downloaded and identities of chunks that are downloaded in the buffer but not yet displayed.

4.4.3 Performing analysis

Given collected information, CSI infers when and what chunks are downloaded by the client player. It then computes the client buffer occupancy across time and analyzes streaming QoE including video quality and stalls. Combining with the network bandwidth information, testers can understand how clients react to different network condition and gain more insights on the design of the tested service. We detail the inference algorithm in the next section.

4.5 CSI inference

Earlier in §4.3, we described the high-level analysis approach of CSI to infer downloaded chunk identities from encrypted traffic. In this section, we describe the challenges in developing it into a practical algorithm and present our solution.

4.5.1 Challenges

To implement the inference into a practical technique, we need to surmount the following challenges.

Challenge 1: QUIC MUX. In Step 1 (§4.3), we desire to identify packets associated with each chunk and then estimate individual chunk sizes. However, for ABR systems using QUIC and with separate audio tracks, when players send requests to download audio and video chunks, video and audio traffic are multiplexed on the same QUIC connection,

Design type	Audio track	Protocol	Transport MUX
CH	Combined	HTTPS	N
SH	Separate	HTTPS	N
CQ	Combined	QUIC	N
SQ	Separate	QUIC	Y

Table 4.2: The ABR streaming system design types

making it challenging to separate corresponding packets. In contrast, on each connection, HTTPS does not send the next chunk request until the current chunk is finished downloading, making it easier to separate traffic for different chunks.

Challenge 2: Large search space. In Step 2, we desire to search for likely chunk sequences that closely match with estimated sizes from the network traffic. Existing traffic analysis work [11, 4, 3, 22] builds fingerprints for each website or app. However, such approaches cannot be directly applied to our problem, as the search space consisting of all possible chunk sequences increases exponentially with the sequence length n. For example, if we stream a 10 min video with 5 video tracks and 5 s second chunk duration, the player downloads 125 chunks and there can be $5^{125} = 7 \times 10^{83}$ potential chunk sequences. Building fingerprints for all combinations and perform exhausted searches quickly becomes infeasible as the number of chunks grows. It is therefore essential to be able to efficiently hone in on the chunk sequence that matches (as per Property (1)) with the sequence of estimated sizes $\tilde{S}_i$ from Step 1.

4.5.2 Algorithm

Popular ABR streaming systems show high diversity in various aspects of the system. We categorize all ABR streaming designs into 4 different types based on the choice on the following 2 key factors.

• Combined (**C**) or Separate(**S**) audio/video: whether the server muxes the audio and video content into combined tracks, or encode them as separate tracks.

- HTTPS(**H**) or the QUIC(**Q**) protocol.

This leads to 4 different design types we shall refer to as {S/C}{H/Q} in Table 4.2. Various popular video streaming services fall into different types, e.g., Amazon Video iOS (SH), Hulu Android(SH) and YouTube Android (both SH and SQ).

When separate audio tracks exist, we observe that popular services commonly use CBR to encode audio content, resulting in almost fixed size chunks in each audio track (typically there are only 1 or a few audio tracks). In this chapter, for simplicity of exposition, we assume all audio chunks in the kth audio track have constant size S_{ak}.

For ABR systems that use QUIC (SQ and CQ), CSI assumes players send at most 1 outstanding video chunk request and 1 outstanding audio chunk request at any time, in line with our observation of behaviors of popular apps using QUIC including Youtube and ExoPlayer. The likely reason for such behavior is that QUIC multiplexes/MUXes requests to the same server over a single connection. As QUIC's congestion control is performed at the connection level, downloading multiple video chunks at the same time does not increase the connection's share of available network bandwidth but instead increases the contention and slows down the download of each chunk. This can increase the potential for stalls and is therefore not preferable. Although the player downloads video and audio chunks concurrently, the player does not start download a chunk until the previous chunk with the same media type finishes downloading. We shall leave extending CSI to other hypothetical uses of QUIC to future. Note that CSI does not make such assumptions for designs using HTTPS. When HTTPS is used, apps in practice may open multiple TCP connections and download multiple chunks concurrently.

Next we first present CSI for the 3 design types that do not have transport MUX (i.e., SH, CH and CQ) and are relatively easier to analyze. Then we present CSI for the remaining more complex design type that use transport MUX (i.e., SQ).

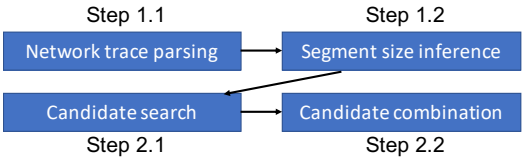

Figure 4.6: Algorithm overview

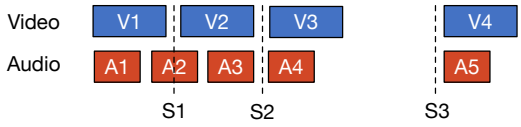

Figure 4.7: Types of split points: *SP1* (e.g., S3), *SP2* (e.g., S2)

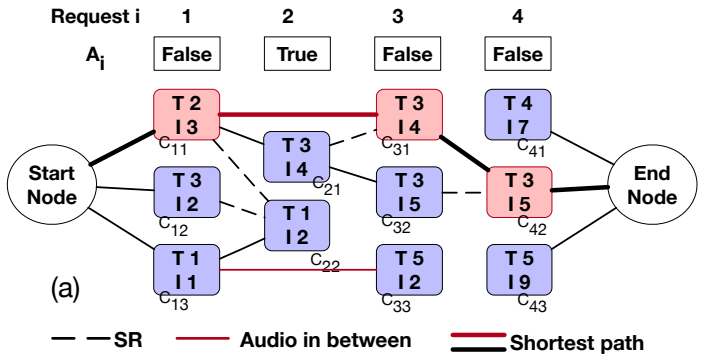

Figure 4.8: CSI inferencing example for the 6 types without transport multiplexing

4.5.3 ABR designs without transport MUX

Figure 4.6 (above Figure 4.7) presents a more detailed breakdown of the 2 steps mentioned in §4.3 for ABR design types that do not use transport MUX.

Step 1.1 CSI parses the network trace and collects the video streaming related packet information. It identifies video connections using the server hostname from the SNI during the handshake, e.g. "googlevideo.com" for YouTube.

Step 1.2 In the absence of transport MUX, on each connection, the player does not send the next request until the current chunk is fully downloaded. Thus CSI group downlink traffic between the two consecutive requests and estimate the chunk size as in §4.3.2. For HTTPS, the request packets can be differentiated from uplink ACK packets using the SEQ number. For QUIC traffic, using the instrumentation and setup in §4.6, we find ACK packets have sizes smaller than 80 bytes, while the request packets are much larger. Thus, CSI uses packet size to differentiate them.

Step 2 Now that CSI has the estimated size sequence $(\tilde{S}_i)_{i=1}^{n}$, it needs to determine the

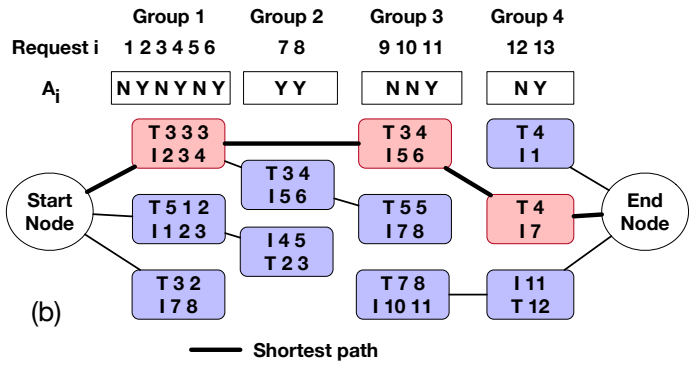

Figure 4.9: CSI inferencing example for type SQN

chunk sequence $(C_i)_{i=1}^{n}$ that satisfies the size constraints in Property (1) for each i and the contiguous index constraint in Property (2). Notice that the search space size of all possible contiguous chunk sequences increases exponentially ($O(T^n)$, T: total number of tracks) with the sequence length. To perform the search efficiently, CSI solves it using a two-level hierarchical approach. It first searches for chunks matching each individual estimated size $\tilde{S}_i$ separately, then combines chunks for different requests into contiguous sequences by modeling the search into the shortest path problem in a graph. Using Dijkstra's algorithm [87], the problem can be solved in $O(n^2)$.

We use SH as a representation to explain in detail how CSI performs the two-layer search.

Step 2.1 For each $\tilde{S}_i$, CSI searches across all video tracks and locates chunks with actual sizes satisfying Property (1). We denote the m video chunks that match $\tilde{S}_i$ as chunk candidates $\{C_{i1}, ..., C_{im}\}$. As SH has separate audio chunks, C_i could also be an audio chunk. We mark the possibility of C_i to be an audio chunk (A_i) as true if $\tilde{S}_i$ and a certain audio chunk size S_{ak} satisfies Property (1). The actual downloaded chunk C_i will be among all these possible candidates.

Step 2.2 CSI combines the candidates for different requests and find the chunk sequence with contiguous indexes satisfying Property (2). The search problem becomes selecting a candidate C_i from $\{C_{im}\}$ for each i, so that the indexes of combined chunk sequence

$(C_i)_{i=1}^n$ are contiguous. CSI convert these candidates into nodes in a graph and formulate the search as the shortest path problem.

As the example in Figure 4.8a, for each request, CSI identifies multiple chunk candidates (each as a node in the graph) in the previous step. CSI adds an edge between candidates corresponding to two consecutive requests if their indexes I_i grow contiguously. For example, an edge is added between C_{11} and C_{21} because I_{11} and I_{21} grows contiguously from 3 to 4. Also, some requests have A_i marked as true and could potentially correspond to audio chunks. In that case, CSI also adds edges between video chunk candidates corresponding to requests surrounding this request if their indexes grow contiguously. For example, an edge is added between C_{13} and C_{33} because A_2 is true and I_{13} adn I_{33} grow contiguously from 1 to 2. After edges are added, each connected path represents a chunk sequence with contiguous indexes. To search for the contiguous chunk sequence corresponding to all requests, CSI assigns all edge length to 0 and use Dijkstra's algorithm to find connected paths covering all requests.

Notice that in the search process CSI does not add assumptions on the client adaptation algorithms and outputs all possible sequences matching with the traffic. We will evaluate how many sequences the algorithm usually outputs and what are the accuracies of the output in §4.6. We find that in many cases CSI finds a unique sequence. Even in the case of multiple sequence candidates, the multiple sequences typically are similar and result in similar QoE.

4.5.4 ABR designs with transport MUX

For system types performing transport MUX, i.e. SQ, CSI needs to address one more challenge: the traffic corresponding to multiple chunks could be transmitted concurrently on the same connection, making it difficult to analyze individual $\tilde{S}_i$ for each chunk. To address this additional challenge, CSI works slightly differently from §4.5.3.

Step 1.2 CSI intelligently detects time points when there is no outstanding request (i.e.,

all issued requests so far are fully downloaded), and splits traffic into small groups at these time points. With such splitting, each group includes the traffic for a smaller set of complete chunks. For example, in Figure 4.7, CSI splits the traffic at time S2 and S3, resulting in 3 groups containing 5, 2 and 2 chunks.

A key question is *how to find proper split points* for the groups. We hope to make each group as small as possible, as it reduces the search complexity in later Step 2. In the extreme case, assume each group only contains 1 chunk, it is equivalent to the design types without transport MUX. But meanwhile, the splitting needs to make sure all the traffic for the same chunk are in one group and thus cannot be performed arbitrarily. For example, S1 in Figure 4.7 cannot be used as a split point, as otherwise chunk A2 is split into two groups. CSI leverages common properties in video streaming and identify two types of split points for QUIC traffic.

The first type of split point *SP1* is based on the common ON-OFF traffic pattern that is widely observed in popular players [71, 88, 11]. Due to buffer management, the client typically pauses fetching chunks if the video buffer occupancy is higher than some threshold, and waits until the buffer occupancy drops below another lower threshold, resulting in a periodical ON-OFF pattern in the traffic. Thus CSI splits traffic when the OFF period is observed. S3 in Figure 4.7 is such a split point. In the implementation, the OFF period can be detected using an idle period longer than some threshold. This threshold typically can be set as a few seconds and can also be tuned for each service.

The second type of split point *SP2* is based on the practice that with QUIC, players only downloads at most 1 video and 1 audio chunk concurrently. Thus CSI splits the traffic when it observes the player sends out two requests at the same time, as this indicates all previous downloads are finished. S2 in Figure 4.7 is such a splitting point.

After the splitting, CSI gathers the packets for each traffic group and estimates total chunk size $\sum_1^n \tilde{S}_i$ in the downlink traffic and the total number of requests n in the uplink traffic.

Step 2 CSI performs a two-layer hierarchical search similarly to §4.5.3 to identify chunk sequences matching with the traffic. It first searches short contiguous chunk sequences matching each traffic group, then combines short sequences for different groups into long contiguous sequences by modeling the search into the shortest path problem in the graph.

Step 2.1 For each group CSI searches for contiguous chunk sequence candidates given the chunk count and total estimated size constraints. As long as the number of chunks in each group is small, we can practically do an exhaustive search over combinations to find the sequence candidates. We evaluate the splitting in previous Step 1.2 using Youtube with various network bandwidth profiles using the setup in §4.6. Combining these two types of splitting points, 99.7% of groups are no larger than 10 requests including both video and audio chunks, which can be easily searched.

Step 2.2 Similar as §4.5.3, CSI combines the candidates from different groups into a contiguous sequence by formulating the search into the shortest path problem in a graph. The only difference is that each node in the graph is a chunk sequence candidate for a traffic group, instead of a single chunk candidate for a single request. As Figure 4.9(b) shows, one candidate sequence for traffic group 1, C_{11}, consists of 3 video chunks with index 2,3,4 respectively and 3 audio chunks (omitted in the figure). Another candidate sequence for traffic group 2, C_{21}, consists of 2 video chunks with index 5 and 6. CSI adds an edge between them as their chunk indexes grow contiguously. After adding all edges, CSI searches for a connected path that covering all requests.

4.6 System evaluation

In this section, we perform evaluation on CSI. We first evaluate the generality of CSI across encodings of popular streaming services. We then demonstrate that CSI achieves high accuracy across different ABR designs.

Service	#Videos	% unique sequences ($k = 1\%$)		% unique sequences ($k = 5\%$)	
		3 chunk	6 chunk	3 chunk	6 chunk
Amazon	111	96.9 (98.0)	100.0 (100.0)	16.0 (27.3)	92.8 (96.7)
Facebook	144	99.4 (100.0)	100.0 (100.0)	58.6 (93.9)	99.5 (100.0)
HBO Now	30	98.0 (98.4)	100.0 (100.0)	24.6 (35.5)	97.3 (98.2)
Hulu	30	97.3 (98.6)	100.0 (100.0)	20.9 (32.2)	90.3 (96.4)
Vudu	46	99.1 (99.9)	100.0 (100.0)	45.6 (81.9)	99.1 (100.0)
Youtube	1920	99.5 (99.9)	100.0 (100.0)	68.8 (89.7)	99.8 (100.0)

Table 4.3: The chunk size variability of popular video services and the percentage of chunk sequences with unique sizes. In cells with format "A(B)", A and B are the median and 95^{th} percentile value across videos.

4.6.1 Different encodings

CSI bases on the insights that there is enough variability in chunk sizes and that given estimated sizes (with certain errors) of multiple consecutive chunks, the identities of these chunks can be uniquely determined. We analyze the generality of the insights on popular video streaming services.

We analyze a number of ABR videos on 6 popular video streaming services, including Youtube, Facebook Watch, Amazon Video, Vudu, HBO Now and Hulu, and collect the individual chunk sizes across all tracks for each video. For Youtube, we use its data API [89] to query videos with more than 1 million views from different categories.For other services, we analyze videos on their landing page.

We find that for all the video services, there exists significant size variability across chunks (Table 4.3). For all services, more than half of videos have a PASR value higher than 1.41. The prevalence of such high PSAR values is due to 2 factors: (1) the wide adoption of VBR encoding in the industry. (2) Newer proposed shot-based encoding schemes [90] perform encoding and segmentation on a shot (scene) basis, leading to variable chunk durations and thereby to variable chunk sizes. We envision such size variability persists in the future, considering the industry trend towards encoding schemes supporting higher efficiencies.

We next analyze the feasibility of using chunk sizes as a fingerprint to identify chunks.

We find that with k of 1%, for every studied service, in more than half of videos, more than 96.9% of 3-chunk sequences are unique. When the sequence length increases, more percentage of sequences have unique sizes. 100% of 6-chunk sequences are unique for every studied service. For QUIC where k is 5%, the percentage of unique sequences is relatively lower, but still, more than 90% of 6-chunk sequences are unique.

Summary. Our evaluations show that size-based chunk identity inferencing has high accuracy for video encodings across a range of popular streaming services.

4.6.2 Different ABR designs

We evaluate the accuracy of CSI for the 4 ABR designs outlined in §4.5. To control the tested system type, on the client side, we use ExoPlayer, a popular open-source Android media player used by more than 10,000 apps [34, 36, 37, 38]. The default ExoPlayer does not support QUIC. We added Cronet as the underlying network stack in ExoPlayer and control it to use HTTPS or QUIC as needed. On the server side, we leverage Youtube servers and stream popular videos from Youtube. The reason we choose Youtube is that videos from most other services are protected by Digit Right Management (DRM) and cannot be rendered using ExoPlayer. However, given the video encoding analysis shown in §4.6.1, evaluation results from other services should be similar for Youtube. One challenge is that YouTube does not provide open interfaces to obtain the video manifest file. To address this, we recreate the manifest file using information obtained from youtube-dl [91], such as video bitrate, resolution and URL etc. YouTube stores video and audio chunks in separate tracks. As we cannot control YouTube encoding and the main focus is the video chunks, to emulate the design types where video and audio content are combined in a single track, we do not add the audio track when generating the manifest and only keep the video tracks.

To ensure repeatable experiments, we collect 30 bandwidth traces from mobile networks in various scenarios covering different signal strength and locations by performing

Case	Without displayed chunk information				With displayed chunk information			
	Best output		Worst output		Best output		Worst output	
	100% match	>95% accuracy	100% match	>95% accuracy	100% match	>95% accuracy	100% match	>95% accuracy
CH	100.0	100.0	64.3	93.6	100.0	100.0	94.9	100.0
SH	100.0	100.0	5.0	91.7	100.0	100.0	9.4	99.4
CQ	100.0	100.0	84.1	95.6	100.0	100.0	92.1	100.0
SQ	98.0	100.0	4.0	52.8	98.5	100.0	91.5	98.0

Table 4.4: The evaluation of inference accuracy with ExoPlayer. "100% match" means the percentage of experimental runs with 100% accuracy. ">95% accuracy" means the percentage of runs with accuracy higher than 95%.

throughput measurement, and let CSI perform traffic shaping based on these traces during experiments. These traces have an average bandwidth ranging from 600kbps to 40Mbps and different bandwidth variability, which triggers different adaptation behavior on ExoPlayer. Note that CSI itself does not make any assumption on the track selection and does not bias towards certain adaption bebavior. We test 5 popular videos covering different genres with durations from 13 min to 1 h. We test 5 runs for each video and bandwidth trace combination. In each test run we stream the video for 10 min. In total we test around 125h of video playback.

To obtain ground truth on the downloaded chunk identities for measuring the accuracy of the inferred results, we instrument ExoPlayer to log chunk request timing and URL. As part of our evaluations, we explore how information on displayed chunks obtained from screen analysis helps improve the accuracy. To get such information, we also add instrumentation in ExoPlayer to log the displayed chunks.

4.6.2.1 ABR without transport MUX

For the 3 design types without transport MUX (Table 4.2), we analyze the inference accuracy as follows. For each experiment, the inference might return multiple candidate chunk sequences. We calculate the accuracy of each inferred sequence and get the high-

est and lowest accuracy across the sequences. The accuracy of an inferred sequence is calculated as the percentage of correctly inferred chunks.

As shown in Table 4.4, for CQ and CH, CSI always find the ground truth as one of the inferred chunk sequences for every run. For 84% of runs, the inferred sequence is unique. Even when multiple inferred sequences are found, the worst identified sequence still achieves high accuracy: higher than 95% for 95.6% of runs. With additional displayed chunk information, the accuracy can be further improved: CSI uniquely infers the ground truth sequence for 92% of all runs.

For SH where audio content is encoded into separate tracks, some requests are for audio chunks, increasing the challenges for the inference. Our evaluation shows that CSI still infers the downloaded chunk identities with high accuracy. For SHN, the accuracy of the best-inferred sequence is 100% for every run. Even the worst candidate for 91.7% of runs had an accuracy exceeding 95%. With additional displayed chunk information, the accuracy improves: even the worst candidate sequence accuracy for 99.4% of runs is higher than 95%.

4.6.2.2 ABR with transport multiplexing

We next evaluate the system design that uses transport MUX, i.e., SQ (Table 4.2). As discussed in §4.5, transport MUX makes it difficult to estimate the size of each chunk and further infer their identities. CSI finds multiple matching sequences for 96% of experiment runs. But for 52.8% of runs even the worst candidate has an accuracy higher than 95% and for 69.8% of experiments with the worst candidate has an accuracy higher than 90%. With the help of displayed chunk information, there is further improvement and CSI can determine the ground truth as the only output for 91.5% of runs.

In terms of computation time, for system designs without transport MUX, CSI typically takes a few seconds to analyze a 10 min's trace on a commodity desktop. For system designs with transport MUX, the analysis time can increase up to around a minute due

to the larger search space involved. Such short response times are reasonable for active testing.

Summary. Our evaluations show that CSI achieves high accuracy for a wide range of system designs, and with reasonable computation times.

4.7 Demonstrating ABR analysis using CSI

We discussed earlier the need for third-party entities including mobile network operators and app developers to perform active measurement and study the adaptation behavior of commerical mobile streaming systems. In this section, we shall illustrate one such important use case for mobile network operators, i.e., designing traffic management polices. It is worth mentioning that the purpose of this section is to demonstrate the need for using active measurement to derive complex design decisions and how CSI helps support such use cases. Determining the optimal traffic management policy itself requires careful considerations between various tradeoffs and large-scale evaluations and is out of the scope of this chapter.

Due to the large traffic volume, mobile network providers commonly desire to perform traffic management policies on video traffic (e.g., [92] and [93]). A typical approach involves using rate-limiting such that players to deliver Standard Definition (SD) quality video to smartphones. The underlying consideration is that especially given the limited screen size of mobile devices, streaming videos at too high quality and resolutions would at best bring only marginal QoE improvements, while consuming substantially more network data, leading to draining the user's limited data budget much faster, and also potentially significantly increasing the load on the network.

A good traffic shaping policy needs to balance the data usage and delivered QoE. To design such a policy, it is essential to understand the interaction between various parameters in the policy design space and app adaptation logic. In the following, as an illustration, we leverage CSI to explore designing a token-bucket based shaping policy. Token bucket

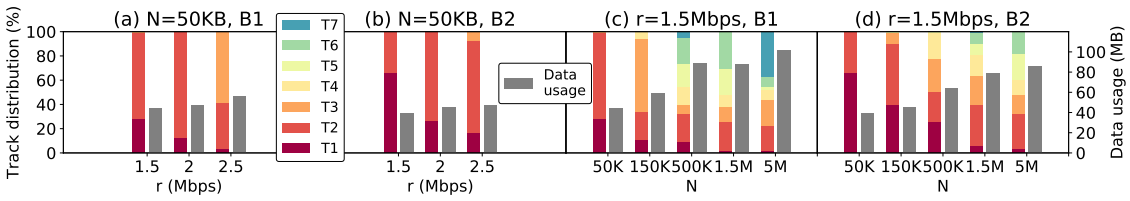

Figure 4.10: The track distribution and data usage for Hulu with different shaping policy and network conditions.

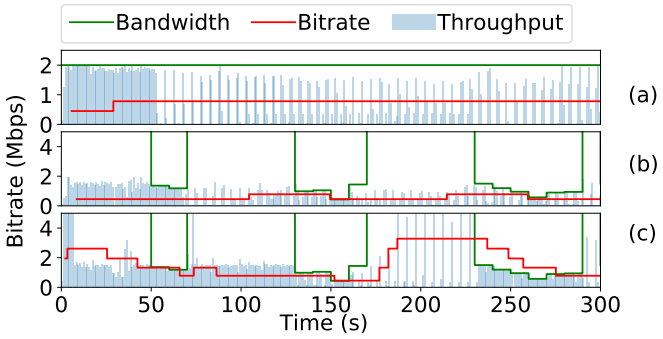

Figure 4.11: Hulu behavior under (a) 2Mbps, (b) profile B2, r=1.5Mbps, N=50KB, (b) profile B2, r=1.5Mbps, N=5MB.

is widely used for traffic shaping and performs shaping based on the expenditure of tokens [94]. It has 2 key parameters: token generation rate r and bucket size N. Tokens fill in the bucket at the rate of r until the bucket is full. When a packet of size s arrives, if there are more than s tokens available, the shaper forwards the traffic and consumes s tokens. Otherwise it queues the packet until enough tokens are generated. As we shall show later, the selection of each of these parameters has to be done carefully, as it can substantially impact the streaming QoE. In the next, we use Hulu as an example service to illustrate how CSI can be leveraged to determine the QoE impact of different parameters of the token bucket.

We first use CSI to gain some basic insight on the adaption design of Hulu, which is essential for later understanding the QoE impact of various shaping configurations. We select a popular video on Hulu. It has 7 tracks with bitrate 0.46/0.79/1.35/1.99/2.67/3.36/4.37 Mbps respectively (we refer to T1-T7 respectively). We perform a series of experiments

within each of which we emulate a stable network bandwidth ranging from 1 Mbps to 4 Mbps. As the example shown in Figure 4.11(a), we find that at the beginning of playback, Hulu always starts by downloading chunks from T1, i.e., the lowest track. After downloading a few chunks, Hulu ramps up to a proper track and stays steady on the track when the network is stable. The bitrate of selected track in always no higher than half of available network bandwidth. We infer buffer occupancy in Hulu across time using the total duration of downloaded chunks that are not played yet. We find Hulu pauses downloading the next chunk when the buffer occupancy increases to around 145s until the buffer occupancy drops below it, generating periodically "ON-OFF" traffic download patterns (after 50s in Figure 4.11(a)).

We now explore how to properly configure r and N to implement a traffic shaping policy that works well with Hulu. To perform evaluations, we chain a Linux machine at the upstream of gateway (in Figure 4.5). On the machine, we configure tbf [95], a tc module, to implement traffic shaping with token bucket with rate r and size N. In the meantime we use tc to emulate cellular network bandwidth on the gateway. In particular, we test 2 different bandwidth profiles as an illustration: a profile with stable bandwidth 10Mbps (denoted as B1), and a profile with bandwidth 10Mbps in most of time but occasionally low bandwidth such as 1Mbps (denoted as B2, see Figure 4.11).

To start with, we fix N to be relatively small (50KB) and test different r values. As shown in Figure 4.10 (a)(b), for both network bandwidth profiles, when r increases, the player spends more time streaming better quality tracks and the fraction of streaming low quality tracks (e.g., T1) reduces. The data usage also increases accordingly. The reason is that higher r indicates higher network bandwidth and Hulu is able to stream tracks with better bitrates.

We next illustrate how the bucket size N affects app behavior. As an example, we fix r to be 1.5Mbps. As shown in Figure 4.10 (c)(d), for both network profiles, with larger N, the percentage of low quality tracks reduces and the percentage of high quality tracks

increases. The reason is that the bucket accumulates tokens at startup or when the cellular network condition is poor. Also, Hulu streams tracks with bitrate no higher than half of available bandwidth, thus its buffer is frequently full and needs to enter ”OFF” periods and pause for some time, leading to tokens acculating in the bucket. With larger N, the bucket accumulates more tokens and allows for larger bursts when the player continues to download chunks. For example, as shown in Figure 4.11, compared with (b), in (c) where N is higher, the achieved instantaneous throughput is much higher when the player resumes from OFF periods. As a result, the player ramps up from low quality tracks faster and play higher quality tracks. However, bigger N introduces higher user data usage. The data usage when N is 5MB is 2.2 times of the data usage when N is 50KB (Figure 4.10(d). It also leads to more frequent track switches. In Figure 4.11(b), the player often selects tracks with rate much higher than 1.5Mbps and quickly consumes all tokens in the bucket. As a result, the player ramps down to T1. Such dramatic change in video quality might severely degrade user experience and should be avoided.

As shown above, both parameters r and N jointly have complex interactions with the app adaptation behavior. To come up with the optimal design, one would need to perform extensive active measurements to test combinations of these parameters under a wide range of cellular network conditions for different videos and streaming services. It needs to carefully consider the tradeoff between various QoE metrics as well as data usage, to achieve data savings while delivering a good user QoE. As we illustrated above, in the presence of encrypted traffic, CSI can help allow testers to evaluate the impact of different token bucket configurations on the ABR streaming QoE.

4.8 Summary

We presented a novel scheme CSI for analyzing ABR streaming behavior of mobile apps in the presence of traffic encryption (HTTPS and QUIC). CSI does not depend on specific designs of certain services or platforms. Extensive evaluations using real videos

and network conditions show that CSI achieves high inference accuracy and can effectively analyze complex player behaviors such as adaptation logic and SR, even for very challenging conditions like encrypted transport multiplexing in QUIC.

When there is transport multiplexing and it is challenging to separate the packets corresponding to different chunks, currently CSI leverages displayed chunk information to perform analysis. We leave exploring techniques such as machine learning techniques and further improving CSI to work without such information to future work.

CHAPTER V

What You See Is What You Get: Measure ABR Video Streaming QoE via On-device Screen Recording

In previous chapters, we focus on network traffic analysis. In this chapter, we explore a conceptually very different approach to QoE measurement — utilizing the on-device recording capability to record the video displayed on the mobile device screen and measuring delivered QoE from this recording. We design a novel system *VideoEye* to conduct such screen-recording-based QoE analysis.

5.1 Introduction

Existing works [4, 17, 11, 43, 51, 53, 54] measure video QoE of commercial proprietary services by analyzing information available in the collected network traffic (e.g., HTTP request URLs). However, with the increasing adoption of end-to-end encryption protocols such as HTTPS and QUIC [23], information such as HTTP requests are no longer visible in the traffic. In such scenarios, existing traffic analysis techniques either can only perform coarse-grained QoE classification or simply do not work. Also, such QoE inferencing attempts from network traffic can be inherently inaccurate due to complexities in the video playback.

In this chapter, we explore a conceptually very different direction — *utilizing commonly*

available standard on-device recording capabilities of smartphones to record the video displayed on the device screen, and measuring the delivered QoE directly from this recording. Intuitively, the approach appears appealing, as the displayed content is what users actually observe and so should accurately reflect user QoE, i.e., "what you see is what you get". It works regardless of the underlying network protocol or player logic. Also, using only on-device recording obviates the need for additional, potentially expensive and cumbersome recording equipment(§5.2.2), and therefore aligns well with the needs of conducting scalable in-the-wild mobile measurements.

While intuitively promising, we identify a number of practical technical concerns, including: (1) potential impact that on-device recording has on the viewing experience, e.g., slow down of the rendering process during playback and perturbation of the streaming QoE, distortation of QoE measurements; (2) the possibility that the recording process introduces distortions (e.g., due to compression) in the recorded video, making it challenging to analyze the original displayed QoE. (3) the challenge to design the analytics to accurately measure QoE from the recordings in the presence of such distortions, where the recordings can be a mixture of multiple tracks with potential stalls.

Our key contributions are summarized below.

- We design VideoEye, to our knowledge the first system to analyze streaming QoE including stalls and displayed tracks from on-device screen recordings (§5.2). It leverages common properties in ABR video encodings and is generally applicable for any videos. It consists of three components, i.e., frame alignment, stall detection and track detection.
- We conduct a measurement study (§5.3) to characterize and quantify the overhead and distortion of screen recording — this understanding was lacking until now. We find that screen recording does not perturb the ABR video playback process, but does introduce significant distortions involving compression artifacts and color space distortions in recorded videos. These distortions are complex and hard to eliminate.
- We develop techniques to measure streaming QoE based on video properties invariant

of recording distortions (§5.4). In addition, we propose optimizations for each analysis component to reduce errors and increase robustness to imperfect inputs. Extensive evaluations show that VideoEye measures streaming QoE with high accuracy across different devices (§5.5-§5.6) even in the presence of recording distortions. A strawman stall detection algorithm had high inaccuracies (e.g., on average estimating 13 sec of more stalls than the ground truth in a 5-min experimental run). In comparison, the maximum error of our stall detection algorithm is only 0.5 sec (§5.6). In addition, even with a recording bitrate of only 5 Mbps, the minimal track detection accuracy for chunks across all runs is 96.9%.

In this chapter we mainly focus on measuring the QoE of streaming Video-On-Demand (VOD) content. This work is being utilized in a large scale production drive testing effort to measure video QoE in a large mobile network. While we focus on Android here, we perform preliminary evaluations on iOS and show that VideoEye is generally applicable to other mobile platforms (§5.7).

5.2 VideoEye system design

We develop VideoEye, a screen-based video QoE measurement platform. As shown in Figure 5.1, VideoEye performs QoE measurement in three steps. (1) Before the in-the-wild testing, testers acquire materials regarding the video to be tested (§5.2.1). This only needs to be done once for each test video and could be performed in the lab. (2) During the testing, testers stream the ABR video in the wild and record the screen display on the device (§5.2.2). We denote generated videos from screen recording as the *recorded video*. This test can be performed repeatedly under various network conditions. In each run, the recorded video could consist of different tracks with different quality levels and might have stalls. (3) After the test, devices upload the recorded videos to the server for QoE analysis (§5.2.3).

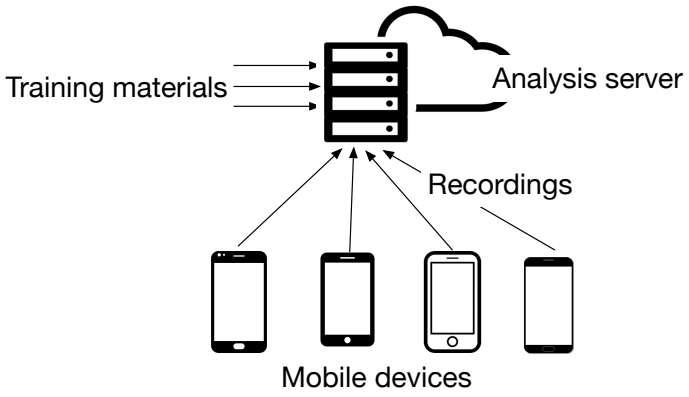

Figure 5.1: VideoEye system overview.

5.2.1 Training material acquisition

Measuring QoE metrics such as stalls and displayed tracks without information on the video content and track encoding is difficult. For example, when the recorded video shows the same content for multiple contiguous frames, it might be caused by a stall, it could also be static scenes in the original video content. To assist accurate QoE analysis without requiring manual inspection, VideoEye requires testers to obtain the following two types of materials for the specific pre-selected video for training purposes, (1) a *content reference*, i.e., a high-quality version of the video content, and (2) the *track references*, i.e., the separate encoded tracks.

To get high-quality content reference, (a) high quality versions of commonly used test videos such as the Big Buck Bunny video (BBB) [96] are typically publicly available, (b) for commercial content such as popular movies, testers can purchase a high quality blue-ray disc, (c) testers can download the highest-quality track as the reference.

To get individual encoded track references, (a) for many popular video services, there exist tools to download track videos, e.g., *youtube-dl* [91] for Youtube, *FBDOWN.net* [97] for Facebook Video and *clipr* [98] for Twitch. (b) If feasible, testers can direct client players to play a certain track by instrumenting the ABR manifest using the following approaches to include only 1 track each time using proxies and record the track. If players use HTTPS

to download the manifest, testers can adopt MITM techniques to instrument the manifest. If players use QUIC, for which there is currently no known way to intercept the connection, testers can block UDP traffic to the device and force players to fall back to HTTPS [99]. Note that server-side track encodings are invariant irrespective of transport protocol used to stream the content. Thus, track references we obtained via HTTPS can be applied to analyze QoE from recordings of streaming over QUIC in the wild as well. (3) In some use cases, it might be possible for the video providers to make the video tracks available to testers to conduct independent evaluations.

Note that the acquisition of above materials only needs to be done once for each test video. In addition, VideoEye automates obtaining the track references for popular services including Youtube (through (a)), Amazon Video and Hulu (through (b)) etc, and minimizes manual effort.

5.2.2 On-device screen recording

During the test, testers stream the tested video on mobile devices and record the screen. The playback on the screen can be recorded via different approaches. Some existing works (e.g., [100]) use a camera to record the display. However, recording via an external camera introduces various challenges. For example, colors in the camera-recorded video can be significantly different from actual displayed colors due to factors such as external lighting environment and camera settings. Also, performing QoE analysis requires pixel-level alignment between the camera window and device display, but this is hard to achieve and maintain, especially for mobile test cases. Another approach is to mirror the device display to HDMI port and record it using additional specialized hardware (e.g., capture cards [101]). We denote this approach as hardware-based recording. It provides high-quality recording. However, requirements for such bulky hardware severely limit the mobility and scalability of the testbed, making it difficult to perform measurements in the wild. To address such constraints, in this chapter we explore what can be achieved minimalistically, using only

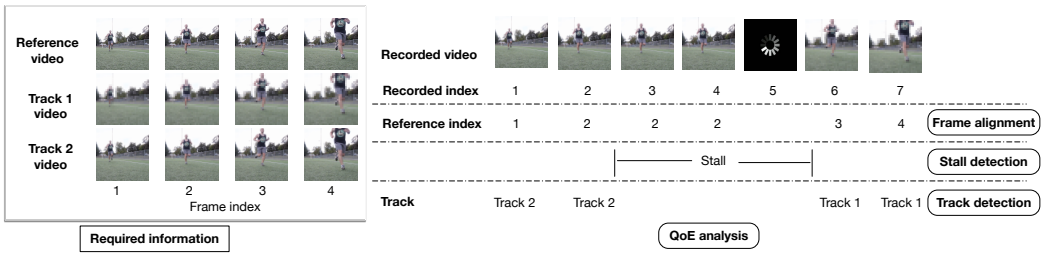

Figure 5.2: Analysis procedure of VideoEye

on-device software and no additional hardware.

Popular mobile OSes such as Android and iOS provide the capability for apps to record the device screen. Considering the operational simplicity and mobility provided by app-based recordings, we focus on designing techniques to work with app-based recordings. in this chapter, we focus on Android devices. But similar principles could generally apply to other OSes such as iOS.

We overview how the Android system supports screen recording functionality. To record the device screen, apps first use the API `MediaProjection.createVirtualDisplay` to mirror the screen to a virtual display, then use `MediaRecorder` to record and encode the displayed content into video files. Underlying the application framework, Android uses a media engine (Stagefright) [102] that uses either built-in software-based or integrated hardware-based codecs to encode the video.

We develop an Android recording app using the system APIs. It records the display and stores the recorded video on the device storage for later analysis. It also provides interfaces to configure the recording quality via the API `MediaRecorder.setVideoEncodingBitRate`. A higher bitrate indicates potentially better quality, but would also lead to higher storage requirements. For example, recording at 5 Mbps for an hour generates around 2GB of video, while recording at 20 Mbps generates 8GB.

5.2.3 QoE analysis based on recordings

After the test in the wild, mobile devices connect to WiFi network and upload recorded videos stored on the sdcard to the server. The server analyzes recorded videos and generates streaming QoE, including stalls and displayed tracks, for each of the runs in four sequential steps (Figure 5.2).

(1) Preprocessing: As we shall show later in §5.3, recorded video generated on mobile devices typically have a variable frame rate and may occasionally miss/drop some frames. To address this, the server preprocess the recorded video and converts it into a frame sequence with a fixed frame rate same as the content reference. It duplicates frames if necessary to keep a constant frame rate. This ensures the time elapsed between two consecutive recorded frames is the same as the time between two consecutive content reference frames, which later facilitates analysis such as stall detection.

(2) Frame alignment: Each recorded frame during playback corresponds to a frame in the content reference. However, the recording does not necessarily start exactly when playback starts. In addition, there might be stalls during playback and frame drops during recording. Thus, the index of a frame in the recorded video (we denote as *recorded index*) could have an offset compared to the frame index in the content reference (we denote as *reference index*). For example, in Figure 5.2, after the stall, the frame with a reference index of 3 is recorded as frame 6 in the recorded video. Therefore, the server performs frame alignment and finds the reference index of each recorded frame (§5.4.1).

(3) Stall detection: After frame alignment, the server performs stall detection to measure stall durations and frequencies (§5.4.2). In the above example, it identifies there is a stall between recorded frame 2 and frame 6.

(4) Track detection: each displayed frame outside stalls comes from one of the multiple tracks which have the same content but different quality levels. Different frames can be from different tracks due to ABR adaptation. After the reference indexes of the recorded frames are identified, the server identifies which track they come from (§5.4.3). For ex-

ample, in Figure 5.2, we detect recorded frame 1 comes from track 2 which has relatively better video quality. This analysis helps measure video quality and variance across time.

We next present the challenges involved in developing these techniques (§5.3) and our proposed solution (§5.4).

5.3 Challenges

Despite the benefits of app-based on-device recording such as low-cost and mobility, on-device recording could potentially bring new challenges including significant overhead and limited video quality. There is little existing work on these challenges. We next perform measurements to gain such understandings. All the measurements are repeated on 4 commodity devices, i.e., Nexus 6P, Pixel 2, Samsung S7 and Samsumg S8. We refer to them as N6, P2, S7 and S8.

5.3.1 Recording overhead

Smartphones typically offload video encoding/decoding to dedicated hardware chips (e.g., DSPs) [103]. As video playback and screen recording share such hardware resources, the recording might perturb playback. For example, if displayed frames are not rendered in time, the playback will have extra stalls. It is thus important to understand and quantify the impact of screen recording to playback.

To characterize the impact of screen recording, we compare the performance of video rendering with and without screen recording. We use `MediaRecorder` to decode the commonly used 1080p BBB video and render it on the screen as quickly as possible. We focus on two metrics, i.e., decoding throughput (the number of frames decoded in 1 s) and delay (the time to decode the first frame). We repeat the test with different recording bitrate including 10 Mbps and 20 Mbps. We find that on all tested devices, the impact of screen recording on video playback is negligible: no matter whether screen recording is enabled or not, the decoding throughput is 60 fps (frame per second), which is higher than the frame

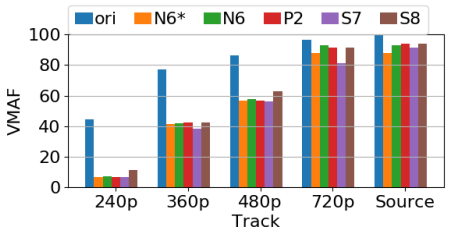

Figure 5.3: The VMAF of Youtube encoded tracks and recordings (recording bitrate: N6* 5Mbps, others 20Mbps).

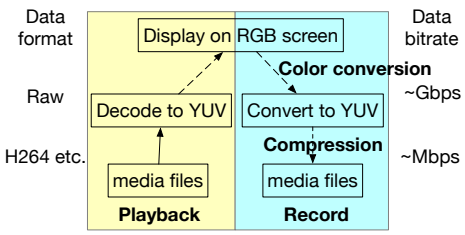

Figure 5.4: Illustration of the playback and recording process.

rate of most popular videos (movie 24fps, TV content 30fps). In other words, even with screen recording enabled, commodity devices are able to decode 1080p videos in time for playback. The reason why the tested throughput cannot exceed 60fps is that these devices are equipped with screens with a refresh rate of 60Hz. The decoder is blocked by the screen to consume decoded frames. We also measure the decode delay and find that the time to decode the first frame is 3-5ms for all devices no matter whether screen recording is enabled or not. This demonstrates that commodity devices are able to support screen recording without impacting video playback performance.

To further validate that screen recording does not perturb streaming QoE, we use ExoPlayer to stream the same ABR video under the same network condition once with, and once without screen recording. We repeat the comparisons under various network conditions, and confirm that for each case the screen recording does not impact streaming QoE.

5.3.2 Recording quality

Another challenge brought by app-based recording is that distortions might be introduced in the recording process. It is critical to understand how the quality of the video played on the screen compares with the quality of the screen recording of that playback. Understanding whether, what kind of, and how much distortions are introduced in the recording process can provide valuable insights that in turn help drive the design of appropriate techniques to analyze the streaming QoE from the screen recordings.

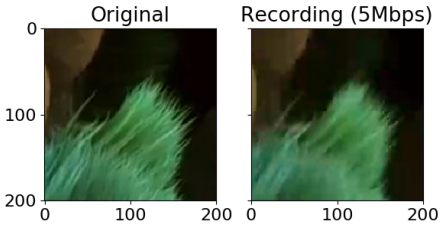

Figure 5.5: Example of the compression artifacts in the recording (6P)

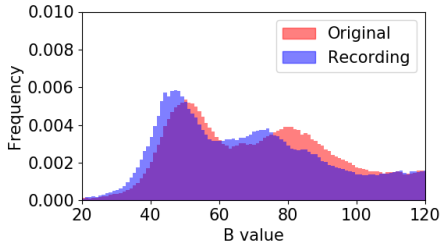

Figure 5.6: The color histogram shifts after the recording (S7)

We perform the following measurements to characterize recording quality and understand how the recording affects the quality analysis of ABR tracks. We upload the high-quality BBB source video to Youtube. Youtube encodes the uploaded video into multiple tracks, each with a different bitrate and resolution. We download encoded tracks, store them on the device storage and use ExoPlayer to play it. We record the playback and later compute the perceptual quality of the recorded frames in terms of the VMAF metric [104]. VMAF is a video quality metric on a scale 0-100 that is demonstrated to have a strong correlation with the user experience. A VMAF difference of 6 can be noticed by users [105]. If the screen recording introduces minimal distortion and users cannot notice any artifacts on the recording, the resulting VMAF value of the recorded video would be close to the original played video. As shown in Figure 5.3, the average VMAF value for recorded frames is significantly lower than that of original frames on all devices even with a recording bitrate as high as 20 Mbps. For instance, the 240p track has a VMAF of 45. But after recording, on all devices the VMAF is less than 10. Recordings of the 360p track have a similar VMAF with the original 240p track. This demonstrates that the recording video quality cannot be directly used to represent the quality of the original played video.

To understand the cause of distortions, we look into the video playback and recording process. As shown in Figure 5.4, during playback, players need to first decode video files (typically several Mbps) into raw pixel streams (typically several Gbps). To express colors, there are different color spaces. Typically video files are encoded in YUV color space to

achieve higher compression efficiency [106]. However, displays use RGB color space and therefore the decoder output needs to be converted to RGB color space before displaying on the screen. When recording the screen, the media encoder converts displayed frames back to YUV color space [107, 108], compresses them and returns the resulting video, which we use for our QoE analysis. Through the process, there are two types of distortions.

• Compression artifacts [109]. This is caused by the loss of high-frequency information in the encoding compression process. As the example in Figure 5.5, after the recording, the details in the image such as the grass become less clear.

• Color space distortion. Colors in the recorded video differ from colors in the original video due to multiple conversion between different color spaces. Such color conversions can cause a mismatch between colors, as different color spaces have different color ranges and there is no simple one-to-one back and forth mapping between colors in different spaces. In addition, in practice the color space conversion is performed with limited precision [110], contributing to the mismatches. To illustrate an example of the color distortion, we plot the color histogram in the blue channel[1] of a original frame and its corresponding recorded frame in Figure 5.6. We can see that the distribution of blue colors shifts towards smaller values. In other words, after recording, the pixel colors are mapped to different colors with lower blue values.

The distortions introduced in the recording process are very complex and hard to compensate. (1) A high recording bitrate cannot eliminate the distortions. As shown earlier in Figure 5.3, even after increasing the recording bitrate from 5 Mbps to 20 Mbps, VMAF of the recorded video is still different from that of the original track. (2) Modeling the color space distortion is challenging. We perform analysis to create a static color mapping before and after the recording. However, we find the same color in different coordinates of a displayed frame could distort to more than 50 shades of the same color in the recorded frame.

[1] In RGB color space, each pixel in the frame is represented using values in 3 color channels, i.e. red, green and blue.

In addition, the distortions are not deterministic: multiple recordings of the same video playback can still be different. We leave exploring how to fully eliminate such distortions to future work.

In addition to the above distortions, screen recording does not always keep a constant frame rate and sometimes drops frames. We compare the recorded frames and original frames to identify dropped frames. The frame drop rate is 1%, 2%, 3% and 0.5% on N7, P2, S7 and S8 respectively.

5.4 Screen-based QoE analysis

The distortions introduced in the recording process make it challenging to analyze video QoE from the recorded video. In the next, we describe how VideoEye addresses such challenges and perform QoE analysis in the procedures mentioned earlier in §5.2.3.

5.4.1 Frame alignment

In frame alignment, we aim to find the reference index of each recorded frame. Compared to frames in the content reference, recorded frames have two types of distortions, i.e. (1) displayed frames can be from one of the ABR tracks which have different levels of compression artifacts compared with reference frames, (2) recorded frames are different from displayed frames due to recording distortions. To perform frame alignment, we need to identify a reference-index-specific signature that is robust to these distortions, but is substantially different for frames with a different index. A candidate to serve as such a signature is the frame thumbnails which are low-resolution images scaled down from the frames. The intuition is that details in the frame are more sensitive to distortions, but the frame outline should be kept. Thus, as shown in Figure 5.7, we can perform frame alignment as follows: extracting the thumbnail for each recorded frame as the signature and searching among the reference thumbnails to find the frame with the smallest difference to know its reference index. The difference between two thumbnails is computed as the sum

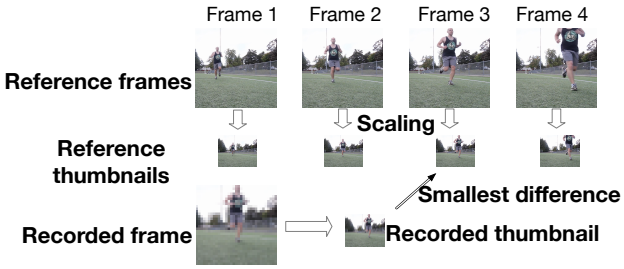

Figure 5.7: Frame alignment algorithm

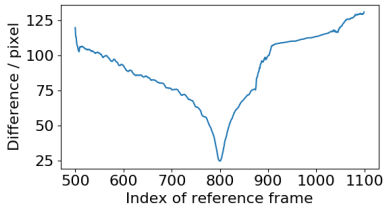

Figure 5.8: The difference between reference thumbnails and a track thumbnail

Seq 1	1	2	5	5	3	4	5	6	7	8	9
Seq 2	7	8	9	10	3	4	12	13	14	3	16
Seq 3	1	2	2	2	5	6	7	8	9	6	11

Figure 5.9: Examples of longest increasing sequence. The grey frames are filtered out.

of the absolute difference between each pixel in them. We show an example of the alignment process for a track frame thumbnail with reference index 800 in Figure 5.8. Among all reference frames, the difference with the reference frame with index 800 is the smallest. Thus we can accurately determine the reference index of the track frame is 800.

Accuracy enhancement. The frame alignment results unavoidably could have errors. For example, video content could have similar frames in scenes with little movement or even between different scenes. Differentiating these similar frames would be challenging. In addition, when stalls occur, the screen typically presents some additional UI animations such as rotating progress bars, making it challenging to detect the reference frame. To reduce errors in the alignment results, we leverage the fact that reference indexes should only increase during playback, and thus extract the longest increasing subsequence (LIS) from the detected reference index sequence. The longest increasing subsequence is defined as the subsequence (which could be not contiguous) where the elements are ranked from low to high and the subsequence is as long as possible [111]. Figure 5.9 shows 3 examples. In the first sequence, the 3th and 4th recorded frames (reference indexes both detected as 5) will be filtered out from the longest increasing subsequence. Our later evaluation in

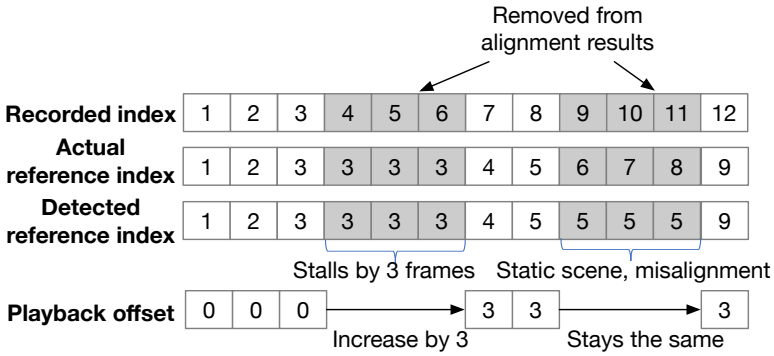

Figure 5.10: An increase in the playback offset indicates the occurrence of stalls.

§5.5 confirms that the longest increasing subsequence effectively reduces the misaligned frames.

5.4.2 Stall detection

After performing frame alignment, we get a sequence of recorded frame indexes and their corresponding reference indexes. Note that the detected reference indexes might have some errors. Based on such information, we next infer the occurrence as well as the start and end time of stalls.

To reliably detect stalls, we leverage the insight that players typically do not skip frames during playback. Recall that the recorded frame sequence is preprocessed to have the same frame rate as the reference video. If we define the difference between the recorded index and reference index of a frame as its *playback offset*, the playback offset will remain constant in the absence of stalls during playback, as the reference indexes of recorded frames will increase at the same speed as recorded indexes. But when a stall occurs, the playback offset of frames will increase, as the reference index will stay the same, while the recorded index keeps increasing. The increase in the offset is equal to the number of frames experiencing stalls. For example, in Figure 5.10, the playback stalls for 3 frames' worth of time at recorded index 4 to 6. At the end of the stall, the playback offset also increases by 3. From another perspective, we can use the increase in the playback offset as an indicator

to detect stalls.

This proposed algorithm can also differentiate stalls from situations where the reference video has frames of similar content. For example, in Figure 5.10, recorded frame 9 to 11 are all wrongly detected to have a reference index of 5, while their actual reference indexes are 6 to 8. However, our stall detection algorithm is robust to such alignment errors, and correctly identify this is not a stall, as the following recorded frame 12 is correctly detected with a reference index of 9 and the playback offset does not change.

Accuracy enhancement. To further increase the robustness of stall detection, we combine results from multiple frames to determine the occurrence of stalls. The insight is that the increased playback offset will persist after the stall. In other words, all frames after the stall will have a larger playback offset compared with frames before the stall. Thus, to further reduce detection errors caused by occasional frame misalignment, we check the next several frames after detecting a potential stall and ensure the majority of them confirm the increase of this offset. The number of frames to check can be tuned by users based on the use case. A larger number of frames will provide better robustness in the stall detection. in this chapter, we use a heuristic of checking the next 10 frames. We shall perform an evaluation of the accuracy of our stall detection technique in §5.6.

5.4.3 Track detection

Each displayed frame can be from one of the multiple ABR tracks. To analyze QoE such as displayed track distribution, after performing frame alignment, we further determine the track each frame comes from.

In the encoding process, tracks with lower bitrate are compressed more and tend to have more compression artifacts. Therefore, a frame in a low-bitrate track will be more different from the corresponding high-quality content reference frame, compared to the corresponding frame in a high-bitrate track. We define the *distance* between two frames as the sum of the absolute difference between each pixel in them. In the example shown

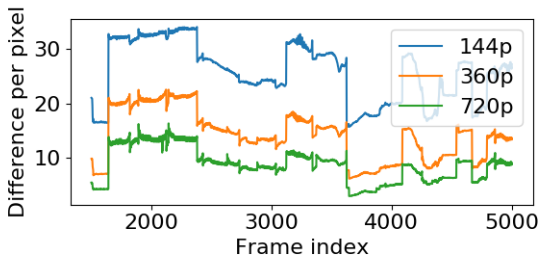

Figure 5.11: The difference between the high-quality content reference and encoded track references

in Figure 5.11, for frames with the same reference index, the distance between frames in the 720p track (the track with the best quality) and the content reference is always smaller than the distance between frames in the 144p track (the track with worst quality) and the reference video. Thus, we can use this distance as a signature to differentiate each track and perform track detection as follows. For a given video we build a distance table, where the ith row corresponds to the ith track and the jth column corresponds to the jth frame in the video. The entry for cell (i, j) in the table is the distance between the jth frame in track i and the corresponding frame in the content reference. Later when analyzing a recording, for every frame we first perform frame alignment to get its reference index, then compute its distance to the corresponding content reference frame. We check the distance table at the specific reference index and then find the entry with the most similar distance to the reference among all tracks. The corresponding track is recognized as the track the frame corresponds to.

When building the distance table for track detection, we have two options: (1) based on the original track references, (2) based on recordings of the track references. We will empirically evaluate the accuracy of both approaches.

Accuracy enhancement. When chunk durations are known, the detected tracks of frames can be further combined to improve accuracy. As track switches typically occur at the chunk boundary, within each chunk (which can be a few seconds long) the track should be the same. We can group the track information of all frames within a chunk and find the

most frequent result as the chunk track.

5.5 Micro-benchmark

The QoE analysis of VideoEye has 3 key sequential steps, i.e. frame alignment, stall detection and track detection. In this section, we focus on the performance of each step in isolation and perform an evaluation of the accuracy of each of these individual steps assuming the previous steps are 100% accurate. More specifically, we focus on evaluating (1) the accuracy of frame alignment and (2) the accuracy of track detection assuming all frames are aligned correctly. It is clear that the stall detection will always give the correct results if all frames are perfectly aligned (§5.4.2), so we do not perform an evaluation on its own in this section. We will combine the 3 steps and perform evaluation on the overall accuracy to measure video QoE later in §5.6.

5.5.1 Experiment setup

We download the top track of 10 popular high resolution (4K) videos from Youtube covering various categories including sports, nature documentary, music and animations. These set of videos cover scenes with different amount of motions and details. To get the ground truth for our evaluation, we use these high-quality videos as the source and embed the frame number using *FFmpeg* as an overlay on each frame. Then we upload the annotated videos to Youtube to encode ABR tracks. In our experiment, we focus on the 5 encoded tracks with resolution 144p, 240p, 360p, 480p and 720p, as they are suitable for playback on mobile devices which have relatively small screen sizes.

We use the Youtube Android app to stream the 10 annotated videos from Youtube servers and record the screen. In each experiment, we constrain Youtube to play a certain track of the video by specifying corresponding track resolution in the app UI. We test 3 different recording bitrates, i.e., 5 Mbps, 10 Mbps and 20 Mbps on 4 devices, i.e., N6, P2, S7 and S8. For each tuple of (video, track, recording bitrate, device), we perform 5

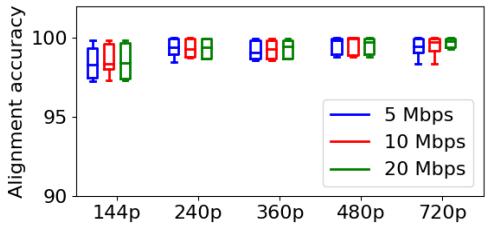

Figure 5.12: Frame alignment accuracy on S7 with various tracks

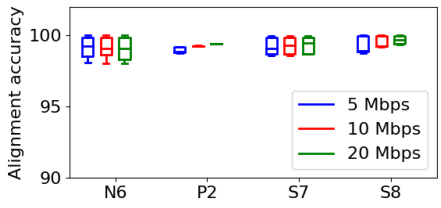

Figure 5.13: Frame alignment accuracy of 360p track on various devices

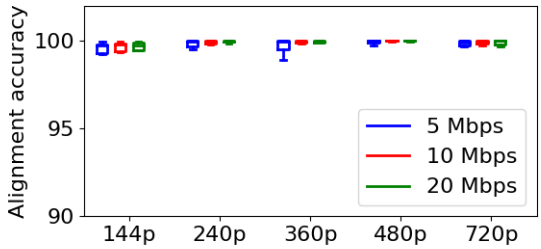

Figure 5.14: Frame alignment accuracy on S7 with LIS optimization

experimental runs. The experiments are automated using Android's UIAutomator framework [85].

To evaluate the accuracy of frame alignment and track detection in each run, we need the ground truth reference indexes and displayed tracks. We derive the ground truth as follows. To get the ground truth reference indexes of recorded frames, we use OCR to extract the frame indexes from the annotation we added earlier on the overlay. To get the ground truth tracks, recall in each run we constrain Youtube to play a certain track, thus the displayed track can be known from the experiment setting. In this section we select Youtube for evaluation, as it allows uploading the specific annotated video and we can more easily get ground truth for evaluation. But the proposed analysis technique is generally applicable and independent of specific video services.

5.5.2 Frame alignment

For each run we mask the annotation of ground truth reference indexes on recorded frames and use our frame alignment algorithm to align the recorded frames. Then we compare the inferred reference index with the ground truth. We compute the percentage of frames that are detected with the correct reference index as the alignment accuracy.

We first examine the accuracy of the core alignment algorithm before performing additional optimization such as extracting the LIS. We plot the distribution of the alignment accuracy across different runs for each setting on Samsung S7 in Figure 5.12 and across different devices in Figure 5.13 respectively. The boxplot in the figures shows the 10pct, 25pct, median, 75pct and 90pct accuracy across experimental runs. Overall we find that on all devices with any of the recording bitrates, the frame alignment achieves high accuracy for all tracks: the median accuracy is always higher than 98% and all runs have an accuracy higher than 90%.

Impact of track: We find that the alignment accuracy for low-quality tracks, especially the 144p track, is relatively lower than other tracks. The reason is that low-quality tracks typically tend to have more compression artifacts, making it more challenging to identify the correct reference index.

Impact of recording bitrate: We find that the recording bitrate does not have much impact on the alignment accuracy. Recording at a bitrate of 5 Mbps yields similar results with recording at a much higher bitrate of 20 Mbps. The reason is that VideoEye performs alignment based on the frame thumbnail and does not require preserving the details. This allows highly accurate alignment even with low bitrate recording and offers various practical advantages, including (1) reducing the requirement on the device storage due to a smaller file size, (2) reducing the associate network transmission overhead to the analysis server.

Impact of device: We find that on all devices, the alignment achieves high accuracy. The difference in alignment accuracy across devices is not significant.

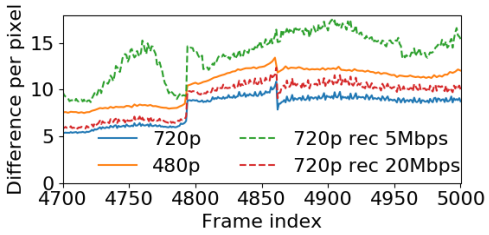

Figure 5.15: The distance between track references and the content reference

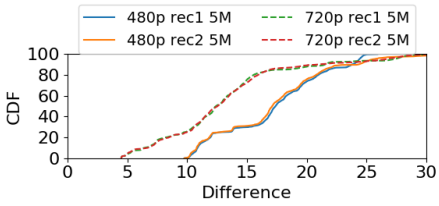

Figure 5.16: The distance between track recordings and the content reference

With LIS optimization: Recall that VideoEye extracts the longest increasing subsequence from the aligned frame indexes to further increase alignment accuracy. We plot the accuracy of the LIS optimization in Figure 5.14. We can see that compared with Figure 5.12, the alignment accuracy is further improved. For example, on Samsung S7, the median alignment accuracy of the LIS across different runs is higher than 99.5% for any track and recording bitrate setting.

5.5.3 Track detection

We evaluate the accuracy of track detection given the correct reference indexes. Recall in §5.4.3 that we have two options to build the distance table for track detection. We evaluate the accuracy using these two types of distance tables.

5.5.3.1 Based on track references

We first evaluate the track detection based on distance table built directly using downloaded track references. We find VideoEye can accurately detect the low-quality tracks. However, high-quality tracks reveal a low detection accuracy. For example, with a recording bitrate of 5 Mbps, the median accuracy of detecting the 480p and 720p track is only 29.5% and 2.7% respectively on S7.

We explore the cause and find that the recording distortions change the distance of

recorded tracks to the content reference significantly. To illustrate the challenge, we plot the distance of the 720p and 480p track reference to the high-quality content reference for the BBB video in Figure 5.15. We can see that at any frame index, the 720p track reference always has a smaller distance from the high-quality content reference compared to the 480p track reference. Such a difference in the distance is the key enabling the track detection. However, as shown in the figure, due to distortions introduced in the recording process, the distance between the track recording to the content reference is significantly larger than the distance between the original downloaded track reference to the content reference. With a recording bitrate of 5 Mbps, the distance of the 720p track recording to the content reference is even larger than the distance between the original 480p track reference to the content reference. As a result, for a 720p track frame recorded at 5 Mbps, the track detection using distance table built from original track references will mis-identify the frame to be from the 480p track.

In summary, due to distortions in screen recording, VideoEye cannot reliably detect tracks from the recorded video using distance table built based on the downloaded track references. Thus, this option should not be used.

5.5.3.2 Based on track recordings

To account for recording distortions, we can instead use the player to play the downloaded track references and record each track. Later we build distance tables using track recordings. The intuition is that if the recording distortion is stable across experiment runs, the distance value in the distance table would be similar to the distance value of a later recording in the experiments, thus we could accurately detect the track of recorded frames. As an example, we record the 720p and 480p track of the BBB video with a recording bitrate of 5 Mbps multiple times and compute the distance of each recording to the high-quality content reference. We can see in Figure 5.16, the distance of track recordings with the same bitrate to the content reference is similar across multiple runs. Note that even

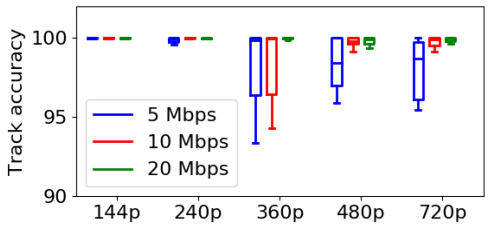

Figure 5.17: Track detection accuracy on S7 with various tracks

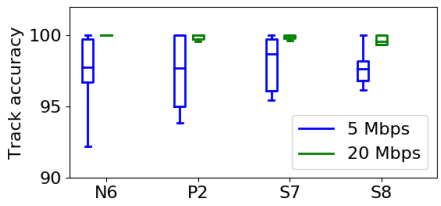

Figure 5.18: Track detection accuracy of 720p track on various devices

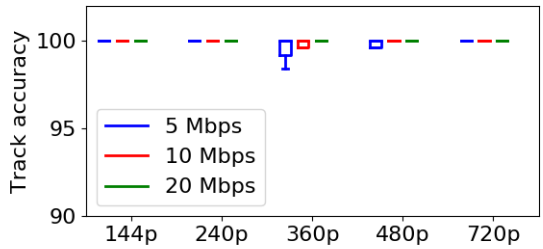

Figure 5.19: Track detection accuracy on S7 with chunk optimization

though this table-building process requires extra steps to record each track, it only needs to be done once for each test video on a device. Furthermore, it can be fully automated and manual efforts are minimized.

We next measure the track detection accuracy using the distance table built from track recordings. As all tracks of the test videos are recorded multiple times, we use one of the recordings for each track to build the distance table, then use the constructed table to perform track detection on the other recordings. As shown in Figure 5.17, the track detection achieves high accuracy. Even with a relatively low recording bitrate of 5 Mbps on S7, the median detection accuracy across runs is higher than 98% for any track.

Impact of tracks: The track detection error increases slightly for high-quality tracks. The reason is that the difference between multiple high-quality tracks is relatively smaller. For example, in Figure 5.11, the difference between the 720p and 360p track is much smaller than the difference between 360p and 144p track. This makes it relatively more prone to errors. But even for the high-quality tracks, the median detection accuracy across

runs is still higher than 98%.

Impact of recording bitrate: The track detection accuracy increases slightly with higher recording bitrate. On Samsung S7, the 5pct detection accuracy of the 720p track is 95.6% with a recording bitrate of 5 Mbps. With a recording bitrate of 20 Mbps, it increases to 98.6%. The cause is that higher recording bitrate causes smaller compression artifacts and better preserves the details in the tracks. This indicates that while even a recording bitrate of 5 Mbps already gets high accuracy and may be sufficient for some use cases, in cases requiring higher track accuracy, testers could choose a high recording bitrate such as 20 Mbps.

Impact of device: As shown in Figure 5.18, the track detection accuracy shows some differences across devices. However, even on the device with the lowest accuracy, with a relatively low recording bitrate of 5 Mbps, the median detection accuracy is higher than 97%.

Combining results for frames in the same chunk: Recall that track detection results for frames in each chunk can be combined to detect the chunk track. As shown in Figure 5.19, this effectively increases the detection accuracy. In more than half of experimental runs, the tracks of 100% chunks are detected correctly, regardless of the track resolution. Note that even without this optimization, the track detection accuracy is already very high.

The evaluation demonstrates that using distance table built from track recordings, VideoEye can accurately detect tracks even with a relatively low recording bitrate.

5.6 QoE Analysis Evaluation

We have evaluated the accuracy of each of the three techniques described in §5.4 assuming previous steps have 100% accuracy. In this section we evaluate the accuracy of combining the three techniques to measure streaming QoE including stall occurrences and displayed tracks during ABR streaming process. As stall detection and track detection de-

pend on the results of frame alignment, this evaluation demonstrates the robustness of our techniques in the presence of alignment errors.

5.6.1 Experimental setup

For evaluation purposes, we need ground truth video QoE information to evaluate QoE measurements from VideoEye. However, it is difficult to obtain QoE information when streaming from third-party proprietary video services. Our experimental methodology therefore works as follows. We encode the 10 high-quality video sources in §5.5 into ABR tracks using FFmpeg based on Netflix's track ladder setting [76]. Then for each track, we use FFmpeg to annotate both the frame index and track number as an overlay on the frames. Later in the recorded video we use OCR to analyze the annotation on the overlay of frames to get their ground truth reference indexes and tracks they come from for evaluation. We use MP4Box [77] to create chunks from each track (each chunk is 2 s) and generate corresponding DASH manifest files. We use ExoPlayer on the mobile devices to stream the video under various network conditions. The screen is recorded with different recording bitrate during the playback. Note that our specific setup (using ExoPlayer to stream annotated videos) is only for getting accurate ground truth QoE for evaluation. VideoEye does not make any specific assumptions about the player adaptation logic and server track encodings, and we therefore expect the findings from the evaluations to be generally applicable irrespective of player adaptation logic and track encodings.

As cellular network is highly variable, it is difficult to do repeatable experiments in the wild. To ensure repeatable experiments, as bandwidth is the key network characteristic that determines player track selection and video QoE, we collect 20 representative cellular network bandwidth traces in various locations. Figure 5.20 shows the distribution of throughput values in each bandwidth trace. We use tc to perform traffic shaping and replay the bandwidth traces during streaming to emulate these cellular network conditions.

We play the 10 DASH streams over the 20 bandwidth traces on 4 devices. Each exper-

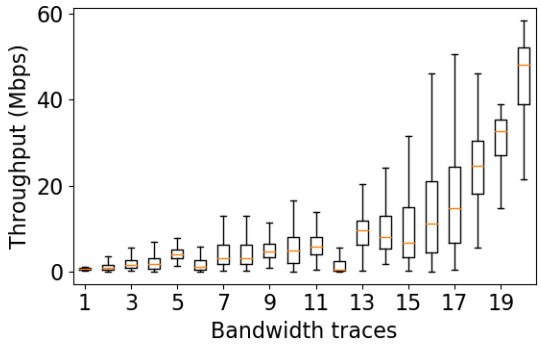

Figure 5.20: The throughput distribution of collected bandwidth trace

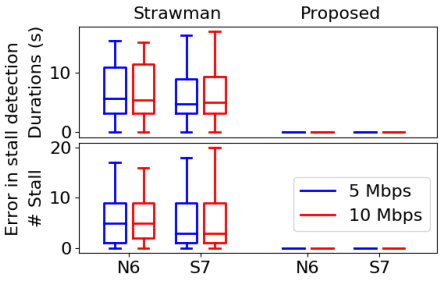

Figure 5.21: The stall detection error of strawman approach and our proposed algorithm.

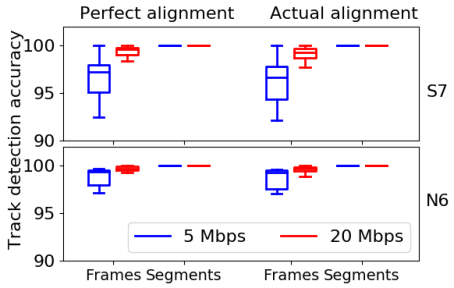

Figure 5.22: The track detection accuracy for ABR video streaming.

iment setting is repeated twice and recorded with two bitrates, i.e. 5 Mbps and 20 Mbps respectively, representing 2 very different points in the space of recording bitrates. In total we recorded more than 260 hours' playback.

5.6.2 Stall detection

As the stall detection depends on the results of frame alignment, we first evaluate the alignment accuracy for the encoded ABR streams. Across all runs we find the alignment accuracy is high. Even with a low recording bitrate of 5 Mbps, the median alignment accuracy on N6, P2, S7 and S8 is 95.6%, 98.1%, 97.5% and 97.9% respectively before applying the LIS optimization. Recall we show our frame alignment algorithm achieves high accuracy on Youtube tracks as well in Figure 5.12, demonstrating that its generality

across different encodings and players.

The above results are encouraging. However, high accuracy in frame alignment does not necessarily guarantee the accuracy of the stall detection. In theory, even with high frame alignment accuracy, if the stall detection algorithm is sensitive to errors in the frame alignment result, even the 2% errors in the frame alignment could lead to low stall detection accuracy. In our evaluation, we focus on stalls longer than 400ms, as very short stalls might not be noticed by users and thus do not have a significant impact on QoE.

A strawman approach is to detect a stall starts whenever the aligned reference indexes of 10 consecutive displayed/recorded frames (around 400ms) do not increase, and ends whenever the aligned frame indexes resume increasing. We evaluate this strawman approach and find that it has high error rate in detecting both stall events and their durations. For example, on N6 with a recording bitrate of 20 Mbps, 82.8% of runs without stalls are inaccurately detected to have stalls and 42.3% of runs without stalls are detected to have total stall duration longer than 5 s. On average the strawman approach detects 8.7 more stalls (13.6 sec of longer stall duration) compared to the ground truth (Figure 5.21). This demonstrates that it is critical for the stall detection algorithm to be robust to occasional frame alignment errors, even if average frame alignment accuracy is high.

We next evaluate the accuracy of our proposed stall detection technique. Recall that our proposed stall detection algorithm (1) leverages the increase in playback offset in the LIS to differentiate stalls with static scenes in the video content, (2) performs a majority vote using a number of frames when a potential stall is detected. These steps were explicitly designed to make the scheme more robust to frame alignment errors. Our evaluations show that in contrast to the strawman solution, our stall detection indeed achieves high accuracy, validating the design decisions. From our evaluation, for all runs (including runs with-/without stall occurrence) on the 4 tested devices, the number of stalls is detected correctly. The stall duration of 75.5% of runs with stall occurrence is detected perfectly. Even though there are errors in detecting the stall duration for 24.5% of experimental runs with stall

occurrence (6.9% of all experimental runs), the error is only a few frames (Figure 5.21). The maximum error in detected stall duration for all experiments is 0.5 s. This illustrates that our proposed stall detection algorithm is robust to occasional errors in frame alignment and achieves high accuracy.

5.6.3 Track detection

In this section, we evaluate the track detection accuracy based on imperfect frame alignment results. We first check the track detection accuracy assuming all frames are perfectly aligned. We find that similar to the benchmark shown earlier in §5.4.3, on all devices, with a recording bitrate of 5 Mbps, the median track detection accuracy for frames is no less than 97.2%. The accuracy further increases with a recording bitrate of 20 Mbps. When combining the track detection results for all frames in a chunk, the median track detection accuracy for chunks is 100% on all devices regardless of the recording bitrate (see Figure 5.22).

We next check how the errors in frame alignment affect the track detection accuracy. We find that the track detection accuracy for frames only marginally decreases. For example, as shown in Figure 5.22, the median accuracy only drops by 0.6% and 0.1% on N6 and S7 respectively with a recording bitrate of 5 Mbps. The reason is that the errors in frame alignment is relatively rare (¡5% on all devices). In addition, we find the errors in frame alignment does not have any impact on the track detection accuracy for chunks. The median track detection accuracy is still 100% on all devices. The reason is that our algorithm selects the most frequently detected track of all frames in a chunk as the chunk track, thus is robust to occasional errors.

In summary, the evaluations demonstrate that our proposed technique can accurately measure video QoE metrics such as stall number and track distribution etc.

5.7 Discussions

iOS platform. in this chapter we focus on evaluations on the Android platform. However, we perform some preliminary evaluations and show that RecVQ applies to iOS platform as well. We leverage iOS's built-in screen recording functionality to record video playback. We find that the default recording bitrate is around 9 Mbps. Similar to Android, the recording also introduces significant distortions. We repeat the evaluation in §5.5 with one Youtube video and find that both the frame alignment accuracy and track detection accuracy of VideoEye is higher than 99% for all tracks. We leave a larger-scale evaluation on iOS devices to future work.

Machine learning techniques. As the first work to develop systems to analyze screen recordings for streaming QoE, VideoEye show that even with relatively simple solutions, video QoE can be accurately measured. We encourage follow-up work to apply machine learning techniques to further improve the system.

5.8 Summary

In this chapter, we explored the idea of measuring ABR streaming QoE via recording and analyzing the video displayed on the mobile device screen. We identified the technical challenges involved, and developed a practical proof-of-concept solution to address these. Extensive evaluation demonstrates that VideoEye has low overhead, and can accurately detect important QoE metrics like displayed tracks and measure stalls.

There are a few directions to explore in the future to further improve VideoEye. First, currently VideoEye requires the high-quality content reference and all encoded tracks as the training materials. VideoEye can be further improved to relax such requirements by exploring techniques such as machine learning. For example, it could require only some frame samples from each track. Second, in this chapter, we develop VideoEye to work with the distortions introduced in the recording process. Future work can explore model-

ing and eventually eliminating such distortions to further increase the analysis accuracy. Finally, while the current system performs analysis on the server, the overall approach can be implemented using on-device processing instead. We leave the actual implementation to future work.

CHAPTER VI

WIQ: Understand QoE Impact of QUIC on ABR Video Streaming with Minimal Effort

Chapter III, IV, V proposes techniques to measure the streaming QoE of existing ABR systems, the majority of which use HTTP/HTTPS to transport video chunks. In this chapter, we develop a system, WIQ (What-If-QUIC), to help developers understand the QoE impact if they change the network protocol to QUIC in their ABR systems before actually devoting effort to upgrade the client and server. It can help guide developers to make decisions on whether to adopt QUIC and also develop ABR design that works better with QUIC.

6.1 Introduction

QUIC, a new transport protocol to replace the traditional TCP/HTTPS stack, has elicited a strong interest in the field. Compared with TCP/HTTPS, QUIC offers many enhanced features, e.g., 0-RTT handshake, better loss recovery and built-in support for stream multiplex etc. to improve network performance. HTTP-over-QUIC is now being standardized by IETF as the next generation of HTTP protocol, i.e., HTTP/3 [23]. It also attracts increasing adoption by the industry. For example, popular mobile apps including Youtube, Snapchat and Uber are already using QUIC for data transmission [112, 113].

However, there is very limited understanding on whether and how much QoE improve-

ment QUIC can bring to commercial ABR video streaming systems. Youtube reports that QUIC reduces their stall rates by more than 15% [9]. Some preliminary studies [114, 115] perform measurements using their own emulated players. However, as the streaming QoE is determined by complex interactions between a wide range of factors spanning different entities such as server encoding and client adaptation algorithm etc., the results from one service cannot be easily generalized to other services.

There is no existing solution for video service providers to understand the QoE impact of QUIC before actually changing the server and client implementation. This causes a 'chicken-and-egg' problem: such modification may require significant engineering efforts and they want to gain understanding on the performance improvement they could get before investing such efforts. To address this problem, in this chapter, we develop a platform called WIQ to perform what-if analysis and characterize the QoE impact of existing ABR streaming systems assuming they switch to QUIC. The system uses two proxies to seamlessly convert between QUIC and HTTPS. It eliminates the need to change the server and client implementation and requires minimal efforts from developers. Using this platform, video service providers can also easily test different existing ABR designs and understand best practices on streaming over QUIC.

We carefully design the system to make sure it introduces minimal overhead and the behavior (such as connection management) is consistent with native QUIC applications. Our key contributions are summarized as follows.

- We design WIQ, a system to perform what-if analysis on ABR streaming systems and characterize the QoE impact of adopting QUIC without the need to change the system client or server.

- We perform evaluations on WIQ and demonstrate that it introduces minimal overhead. The extra delay is no more than 30 ms. To showcase the use of WIQ, We apply it to ExoPlayer and find that QUIC effectively reduces the startup delay of ExoPlayer by up to 991 ms.

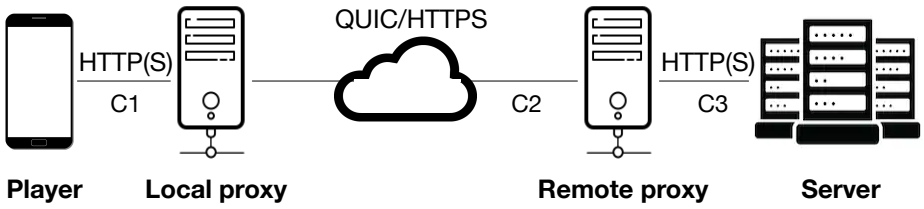

Figure 6.1: System overview.

6.2 System design

To allow developers to measure the video QoE over QUIC without the need to change the server or client implementation, as shown in Figure 6.1, we propose a two-proxy approach. Near the device, we deploy a *local proxy* which accepts HTTP/HTTPS requests from the client app, but forwards requests over QUIC to the remote proxy. Near the video server, we deploy a *remote proxy* which accepts requests from the local proxy over QUIC, but sends requests to the video server via HTTP/HTTPS.

In the system, the connection between the client and server is split into 3 connections: the connection between the client device and the local proxy *C1*, the connection between the local proxy and remote proxy *C2* and the connection between the remote proxy and the server *C3*. The end-to-end performance is determined by C2 which runs QUIC traffic, as C1 and C3 have a good network connection with high bandwidth and low latency (the local proxy is deployed close to the client, the remote proxy is deployed close to the server).

We implement the two proxies on top of GoProxy [116], a popular HTTP proxy implemented in Go language. To add QUIC support to the proxy, we integrate it with quic-go [117], a QUIC implementation widely used in previous studies [118, 119] and the Caddy Web Server [120]. The proxies work slightly depends on whether the client uses HTTP or HTTPS. (1) When the client uses HTTP to send requests, as there is no encryption, the local proxy directly reads the request and forwards it over QUIC to the remote proxy. The remote proxy reads the request and forwards it to the video server. (2) When the client uses HTTPS to send requests, the local proxy performs Man-In-The-Middle (MITM) to inter-

cept the traffic and extracts the HTTP payload. It then sends the HTTP payload over QUIC to the remote proxy. We choose this option instead of the alternate: directly forwarding the encrypted TLS payload over QUIC, as it would encrypt the payload twice and does not match with real QUIC usage in the wild. The remote proxy reads requests from the QUIC connection and sends HTTPS requests to the video server. In both cases, the video server receives requests in the same protocol as the client sends them: if these two protocols are different, it might lead to failure in the system. For example, if the client sends the request using HTTP, while the remote proxy sends HTTPS, the video server might not support HTTPS and causes failure. On the other hand, if the client sends the request using HTTPS, but the remote proxy sends HTTP, the video server might respond by redirecting the client to HTTPS, while the remote proxy keeps HTTP and causes infinite redirections. To avoid such failures and notify the remote proxy which protocol clients initially use, the local proxy appends the protocol name to the end of the hostname when sending the request to the remote proxy. The remote proxy reads the protocol information from the domain name and send the request to the original host.

We try our best to minimize the overhead incurred by the proxies and shall perform the benchmark in the following §6.3. Meanwhile, to take the overhead into consideration when doing comparison, we add the option to let the local proxy and remote proxy communicate in HTTPS. When doing comparison, we do not compare the case where WIQ is not used with the case where WIQ is used. Instead, we compare the case where WIQ uses HTTP to the case where WIQ uses QUIC. In both cases, the end-to-end performance includes the overhead caused by WIQ. The performance difference would only be due to the protocol difference between QUIC and HTTPS.

The traffic from the client can be redirected to the local proxy via one of the following approaches: (1) specifying the proxy in the system setting of the device, (2) using a VPN to route the traffic from the device through the local proxy machine and using ‘iptable‘ to redirect it to the proxy port.

When the client uses HTTPS, the local proxy performs MITM and uses a self-signed certificate to generate certificates to the client. To make clients trust these generated certificates, the self-signed certificate needs to be installed on the test device as a trusted CA (certificate authority). From Android 7, apps no longer trust user-installed certificates. In this case, developers can explicitly change the security configuration of the app to trust the certificate for testing purposes.

To facilitate performance analysis, we instrument the local proxy to generate logs including essential information to analyze the performance, including request URL, request and response size, the time when the local proxy receives the request from the client, the time when it receives response from the remote proxy, and the time it sends the response back to the client etc.

We also change the connection management of the local proxy to make it match the behavior of native QUIC apps. By default QUIC opens one connection for each host and multiplexes all requests to that host on this single connection. In WIQ, as the local proxy always communicates to the remote proxy, it will multiplex all requests for all hosts through one connection, which does not match with the behavior of native apps. To fix this, we instrument the local proxy to read the request host and sends the requests to each host on a separate connection.

6.3 Overhead benchmark

The encryption/decryption operation and HTTP parsing when WIQ performs MITM introduces extra delay. In this section, we perform measurements to characterize such overhead of WIQ. In particular, we compare the time it takes to download objects with and without WIQ.

We set up an HTTP server and host files with different sizes. We develop an Android app to download these objects and measure the download time. As shown in Figure 6.2, all traffic from the mobile device is routed to the local proxy machine using ShadowSocks

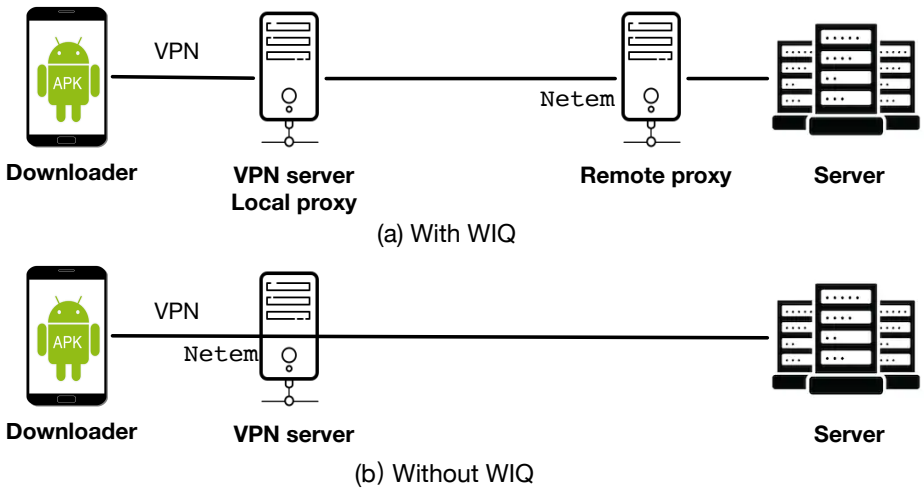

Figure 6.2: Benchmark setup.

VPN [121]. When WIQ is not enabled, the traffic sent to the VPN server is directly sent to the HTTP server without actually processed by any of the two proxies. When WIQ is enabled, the traffic sent to the VPN server is routed through the local proxy using "iptables". The local proxy further sends the data to the remote proxy.

We use *tc-netem* [122] to perform traffic shaping and emulate networks with different bandwidth and latency. In the case without WIQ, we perform traffic shaping on the egress of the VPN server. In the case with WIQ, we perform traffic shaping on the egress of the remote proxy. The other two connections, i.e., C1 between the mobile device and the local proxy and C3 between the remote proxy and the HTTP server, have very good network conditions: the latency is less than 1ms and the bandwidth is no less than 40 Mbps.

We download objects of 10KB, 100KB and 1MB, which are close to typical chunk sizes. The client test both HTTP and HTTPS to download the objects. For WIQ, there are 2 settings: (1) WIQ disabled (we denote as *noproxy*), (2) WIQ enabled and the two proxies communicate using HTTPS (we denote as *httpsproxy*), (3) WIQ enabled and the two proxies communicate using QUIC (we denote as *quicproxy*. For the network condition, we test bandwidth of 1 Mbps, 5 Mbps or 10 Mbps. For each (object size, WIQ setting, network bandwidth) combination, we download the object 100 times.

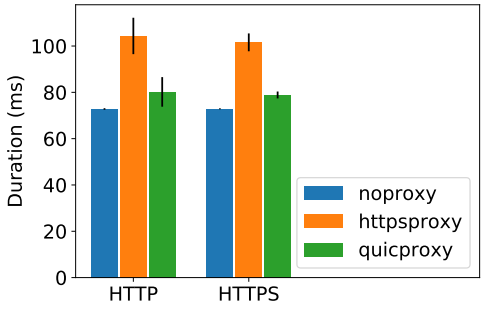

Figure 6.3: The time it takes to download 10 KB (bandwidth 1 Mbps).

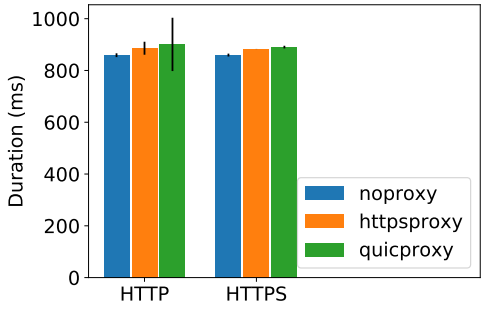

Figure 6.4: The time it takes to download 1 MB (bandwidth 10 Mbps).

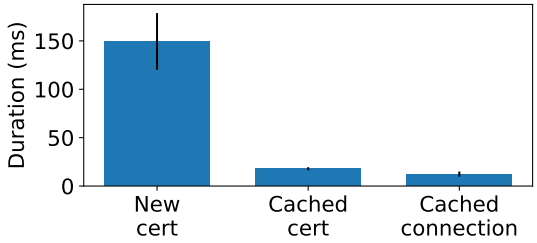

Figure 6.5: The extra time it takes to generate a new certificate and establish a new connection (object 10KB, bandwidth 10Mbps)

We find that the overhead introduced by WIQ is less than 30 ms for all object sizes and network conditions. We show two representative examples in Figure 6.3 and 6.4. The increase in download time is less than 30 ms for both small (10 KB) and big (1 MB) objects in network with high bandwidth (10 Mbps) or relatively low bandwidth (1 Mbps).

When performing the above benchmark, the client uses persistent TCP connections to send requests. We next evaluate the overhead of establishing a new connection. The first time WIQ receives a request to a certain host, it needs to generate a new certificate for that host and sends it to the client to finish the handshake process. This involves extensive computation and introduces extra delays. As shown in Figure 6.5, it takes 149 ms to download an object of 10 KB in a high-bandwidth network. Compared with this, for later requests to the host, where the certificate is already generated and the connection is already estab-

lished, it only takes 12 ms to finish the request. To minimize this overhead, WIQ caches generated certificates to disk and reuses them for new connections to known hosts. When the certificate is cached, it only takes 18 ms to establish a new connection and download an object of 10 KB. When performing what-if analysis using WIQ, before the actual run, testers could first perform a test run to "warm up" WIQ and let it generate the certificates.

6.4 Showcase use case

In this section, we showcase how WIQ can help understand the QoE impact of QUIC on ABR streaming systems. From Android 7 onward, apps by default no longer trust user-installed certificates. As a result, without the help of video service providers, it is challenging to intercept the connections of commercial streaming services. In this section, we perform the demonstration using ExoPlayer [123], a popular open-source Android player. Note that app developers of commercial streaming services can change their security configurations and generate test versions of the app to work with WIQ.

We encode the Big Buck Bunny [74] video into an ABR stream with 5 tracks using FFmpeg. We host the video chunks on an HTTP server and use ExoPlayer to play it. We collect 5 network bandwidth profiles and use Netem to perform traffic shaping based on these bandwidth profiles. The latency between the local proxy and remote proxy is set to be 100 ms. For each bandwidth profile, the experiment is repeated 5 times.

We find that for ExoPlayer, QUIC does not consistently improve streaming video quality compared with HTTPS. As shown in Figure 6.6, in some cases such as bandwidth profile 4, the duration of streaming low quality tracks (e.g., 240p) slightly reduces. However, in some other cases such as bandwidth profile 3, the duration of streaming low quality tracks increases. As shown in Figure 6.7, it does not consistently reduce stall durations either.

We find that QUIC effectively reduces the startup delay. As shown in Figure 6.7, the startup delay is reduced by up to 991 ms. This is likely due to QUIC's 0-RTT handshake feature.

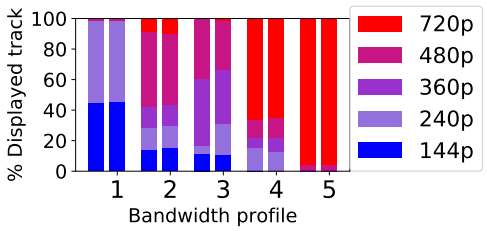

Figure 6.6: Displayed tracks of ExoPlayer over HTTPS vs QUIC.

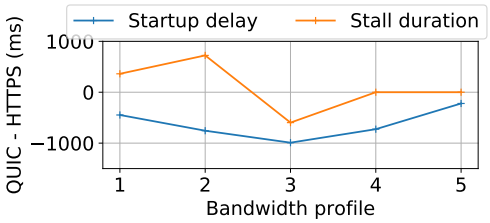

Figure 6.7: Startup delay and stall duration of ExoPlayer over HTTPS vs QUIC.

6.5 Summary

In this chapter, we present WIQ, a system to help video service providers understand the QoE impact of QUIC on their ABR streaming system without the need to modifying the server or client. We demonstrate that the system incurs minimal overhead and use it to show that QUIC can help reduce the startup delay of ExoPlayer.

As future work, we can use WIQ to perform what-if analysis on a wide range of commercial services. Based on the measurement findings, we can develop best practices on ABR design to fully take advantage of new features provided by QUIC and improve streaming QoE.

CHAPTER VII

Related Work

7.1 QoE characterization of commercial ABR systems

Some existing works make effort to characterize streaming QoE of video services. However, none of the existing methodologies can be generally applied to the mobile VOD services we study.

HTTP Parsing. Some studies [3, 39] extract bitrate information from the request URL based on certain URL patterns. However, the URL pattern differs between services and many services even do not have such patterns. For example, we find that Netflix and Amazon do not directly put bitrate information in the URL. Akhshabi et al. [4] estimate the chunk duration based on their sizes. However, we find that many video services use variable bitrate (VBR) encoding. Even for the same track, the actual bitrates of chunks vary significantly. In this dissertation, we develop techniques to parse the HTTP request based on standard ABR protocols, which can be generally applicable to different services. In addition, when traffic encryption is adopted, HTTP information is no longer available, defeating such approaches. In this dissertation, we develop two novel techniques to analyze streaming QoE even in the presence of traffic encryption.

ML Classification. To extract QoE information from encrypted network traffic, some recent work [17, 51, 53, 54] trains ML models to predict video QoE from traffic characteristics such as network throughput etc. However, the predicted QoE information is

coarse-grained qualitative results, typically binary decisions on whether the QoE is good or not. It does not give information on the individual stalls or displayed tracks, making it impossible to get a comprehensive understanding of user experience. In addition, the model is application-specific and requires labeled data for training, which can be hard to get in the first place. In this dissertation, we develop techniques that give detailed information on app adaptation behaviors and provide fine-grained QoE metrics such as the displayed tracks over time.

Service-specific analysis. Some analysis techniques leverage service-specific features to extract video QoE. For example, Youtube provides special modes [84] that display streaming track information on the UI. Some apps output QoE debugging information in logcat. However, such approaches only work for specific services with such support and are not available for general services. In contrast, the systems developed in this dissertation are generally applicable for ABR services with different designs.

7.2 Proposal of novel adaptation algorithms

Many prior works [18, 39, 40, 41, 5, 124, 125] have investigated the opportunities for optimizing the rate adaptation algorithms. Jiang et al. [39] propose an adaptation algorithm that improves fairness between multiple video streaming applications. Li et al. [40] use a TCP-like probe approach to select video bitrate. PiStream [41] leverages physical layer information in the LTE network to help predict network bandwidth and adapt video bitrate. Huang et al. [5] select video bitrate based on buffer occupancy. These state-of-the-art algorithms can help improve video streaming performance. In this dissertation, we investigate the algorithms deployed in commercial mobile VOD systems in practice.

7.3 Diagnosis of QoE issues in video streaming systems

Prior efforts [126, 127, 128, 129, 130, 131, 132, 133, 134, 135]have also emphasized the importance and challenges of diagnosing the performance problems of video streaming. Jiang et al. [126] propose to use clustering methods over client attribute to identify root causes of video problems. A recent work [136] builds a machine learning model to perform root cause analysis for poor video QoE based on network characteristics. These works focus on identifying problems caused by the external environment such as poor network conditions. In this dissertation, we build systems to understand the QoE implications of various ABR designs.

7.4 ABR streaming over QUIC

Youtube is the only known commercial ABR service that already deploys QUIC in production till now. Work [9, 10, 137] perform measurements to understand how QUIC affects Youtube's streaming QoE. Some other work [114, 115, 138, 139] performs measurements using an emulated player or open-source players to understand the impact of QUIC on video QoE such as startup delay etc. Different from these work that focus on a specific service or only works for open-source players, in this dissertation, we develop measurement techniques that work for general close-sourced commercial ABR services already using QUIC and provide support for services that are considering the adoption of QUIC to perform what-if analysis.

Some other work looks into how to improve QUIC to better work with video streaming applications. Work [140] proposes to extend QUIC to support unreliable streams. [141] proposes a new congestion control mechanism using QUIC to vary downloading rate based on buffer status. This is parallel to our work which tries to develop ABR designs that better work with the various features provided by QUIC.

CHAPTER VIII

Conclusion

Designing an ABR streaming system with good QoE properties is challenging, as the QoE is determined by complex interactions between a wide range of factors across different layers spanning different entities. To help identify QoE issues and derive better designs, it is important to provide support for both first-party video service providers and other third-party entities such as network providers to perform continuous measurements and understand the QoE implications of various design decisions.

In this dissertation, we focus on developing techniques and building platforms to perform such measurements for general ABR streaming systems and understand the interaction between different layers (e.g., network layer transport and application layer adaptation). Specifically, we develop four techniques or platforms to address various challenges in performing the measurements, including the proprietary nature of commercial services, the adoption of traffic encryption and the use of the new protocol QUIC.

- We develop a general measurement platform to analyze the various design factors and resulting QoE of ABR streaming systems based on standard ABR protocols and common UI designs. We carefully craft black-box experiments to stress-test 14 commercial services and glean critical properties of their design. From the study, we identify a number of QoE-impacting issues and derive best practices for improvement.

- We design CSI, a novel system to support third-parties to perform active measurements and infer ABR behavior even in the presence of traffic encryption. It infers downloaded chunk sizes from associated encrypted packets and uses the sizes as a fingerprint to infer the chunk identities. We develop a novel algorithm to efficiently perform the inference. Extensive evaluation demonstrates that CSI achieves high accuracy.

- We develop VideoEye to analyze streaming QoE from on-device screen recordings. It is generally applicable for any videos and does not depend on specific network protocols. We perform measurements to understand the distortions introduced in screen recordings and develop techniques to measure video QoE based on properties invariant of such distortions. Our evaluation shows that it can accurately measure the stall duration and detect displayed tracks.

- We design WIQ to perform what-if analysis on ABR streaming systems and characterize the QoE impact if they adopt QUIC. It eliminates the need to change the system server or client, and thus requires minimal effort from developers. We perform evaluations and demonstrate the WIQ introduces extra overhead to the system performance.

When designing these measurement systems, we leverage general properties that commonly apply to a wide range of ABR streaming systems. (1) For the black box measurement techniques, we parse network traffic based on standard ABR protocol specifications. (2) CSI relies on the insight that common encryption protocols do not apply extensive padding techniques to hide payload size due to the associated data overhead. It also leverages the fact that popular services widely adopt VBR encoding due to its higher efficiency compared with CBR encoding. (3) VideoEye relies on fundamental properties in the video encoding process. For example, the tracks with lower bitrate have more distortions and are thus more different from the reference video. (4) WIQ performs protocol conversions at

the transport layer and works with different ABR designs. None of these systems depends on service-specific logic such as a particular adaptation behavior. Following this principle, we guarantee that our systems are generally applicable to different ABR services.

In the future, when new ABR protocols, encryption protocols or encoding techniques are proposed, our measurement systems can be extended to work with them. For example, the HTTP network parser can be updated based on the specification of the new ABR protocol. CSI can work with new encryption protocols as long as they do not apply extensive padding to hide payload size. WIQ can perform what-if analysis for ABR streaming over new transport protocols by integrating support for the new protocol. Even in the case where there are unseen fundamental changes in the general properties our measurement techniques depend on, the measurement philosophy our techniques follow can still be generally applicable: understanding what chunks are downloaded and what chunks are displayed over time. With these two pieces of information, we can gain valuable insights into the ABR system design and measure streaming QoE.

8.1 Future work

There are a few directions that this dissertation does not dive in. We encourage future works for further exploration.

- *Live streaming and 360-degree video*. In this dissertation, we mostly focus on VOD content. In recent years, other forms of video streaming such as live streaming and 360-degree video streaming have been increasingly popular. Many of these video streaming systems use similar streaming protocols as VOD (e.g., DASH and HLS). The techniques developed in this dissertation can be directly applied or extended to them with some modifications. We leave a detailed study to future work.

- *iOS platforms*. Many systems developed in this dissertation such as CSI and VideoEye are general for different device platforms. However, our evaluation mostly fo-

cuses on the Android platform. We leave performing a measurement study on the iOS platform to future work.

- *Machine learning*. To perform tasks such as identifying identities of downloaded chunks from the encrypted traffic and detecting displayed tracks from recordings, in this dissertation, we develop some simple but effective techniques. We believe that machine learning techniques are also suitable to solve these problems and we encourage future work to explore leveraging machine learning to further improve the accuracy of the measurement.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Cisco Visual Networking Index: Forecast and Methodology, 2016-2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html, 2017.

[2] Experience Shapes Mobile Customer Loyalty - Ericsson. https://www.ericsson.com/thinkingahead/consumerlab/consumer-insights/experience-shapes-mobile-customer-loyalty, 2017.

[3] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, timid, and unstable: picking a video streaming rate is hard. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 225–238. ACM, 2012.

[4] Saamer Akhshabi, Ali C Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 157–168. ACM, 2011.

[5] TY Huang, R Johari, N McKeown, M Trunnell, and M Watson. A buffer-based approach to rate adaptation. In *Proc. 2014 ACM Conference on SIGCOMM, SIGCOMM*, volume 14, pages 187–198, 2014.

[6] Qi Alfred Chen, Haokun Luo, Sanae Rosen, Z Morley Mao, Karthik Iyer, Jie Hui, Kranthi Sontineni, and Kevin Lau. Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 151–164. ACM, 2014.

[7] Protecting Netflix Viewing Privacy at Scale. https://medium.com/netflix-techblog/protecting-netflix-viewing-privacy-at-scale-39c675d88f45, 2016.

[8] Openwave Mobility Mobile Video Index, Dec 2017. https://owmobility.com/whitepapers/, 2017.

[9] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196. ACM, 2017.

[10] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. Taking a long look at quic: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017 Internet Measurement Conference*, pages 290–303. ACM, 2017.

[11] Shichang Xu, Subhabrata Sen, Z Morley Mao, and Yunhan Jia. Dissecting vod services for cellular: performance, root causes and best practices. In *Proceedings of the 2017 Internet Measurement Conference*, pages 220–234. ACM, 2017.

[12] Roger Pantos and William May. HTTP live streaming. `https://tools.ietf.org/html/rfc8216`, 2017.

[13] ISO/IEC 23009-1, Information technology - Dynamic adaptive streaming over HTTP (DASH). `http://standards.iso.org/ittf/PubliclyAvailableStandards/c057623_ISO_IEC_23009-1_2012.zip`, 2012.

[14] Smooth Streaming Protocol. `https://msdn.microsoft.com/en-us/library/ff469518.aspx`, 2017.

[15] Content providers use QoE monitoring to grow ad revenue from VOD content. `http://www.witbe.net/2018/08/28/11142/`.

[16] Conviva's continous measurement annual census report. `https://www.conviva.com/blog/2017-ott-streaming-market-year-review/`.

[17] Giorgos Dimopoulos, Ilias Leontiadis, Pere Barlet-Ros, and Konstantina Papagiannaki. Measuring video QoE from encrypted traffic. In *Proceedings of the 2016 Internet Measurement Conference*, pages 513–526. ACM, 2016.

[18] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: near-optimal bitrate adaptation for online videos. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE, 2016.

[19] GL core product catalog: drive testing. `https://www.gl.com/drive-testing.html`.

[20] Rohde & Schwarz Mobile Network Testing - video quality test solution. `https://www.rohde-schwarz.com/ae/solutions/test-and-measurement/mobile-network-testing/expertise/video-quality-testing/video-quality-testing_232054.html`.

[21] TEMS Portfolio: Where superior mobile network experiences begin. https://www.infovista.com/products/tems-portfolio.

[22] Tarun Mangla, Emir Halepovic, Rittwk Jana, Kyung-Wook Hwang, Marco Platania, and Mostafa Ammar. Videonoc: Assessing video qoe for network operators using passive measurements. In *Proceedings of ACM Multimedia Systems Conference*. ACM, 2018.

[23] IETF QUIC working group. https://datatracker.ietf.org/wg/quic, 2018.

[24] draft-ietf-quic-tls-13 - Using Transport Layer Security (TLS) to Secure QUIC. https://tools.ietf.org/html/draft-ietf-quic-tls-13.

[25] Man-in-the-middle attack. https://en.wikipedia.org/wiki/Man-in-the-middle_attack.

[26] Android Network Security Configuration. https://developer.android.com/training/articles/security-config.html, 2017.

[27] Luca De Cicco and Saverio Mascolo. An experimental investigation of the Akamai adaptive video streaming. In *Symposium of the Austrian HCI and Usability Engineering Group*, pages 447–464. Springer, 2010.

[28] How Xposed works. https://github.com/rovo89/XposedBridge/wiki/Development-tutorial, 2017.

[29] Technical Note TN2224. https://developer.apple.com/library/content/technotes/tn2224/_index.html#//apple_ref/doc/uid/DTS40009745-CH1-BITRATERECOMMENDATIONS, 2017.

[30] CBR and VBR Encoding FAQ: What is The Difference? https://www.lifewire.com/difference-between-cbr-and-vbr-encoding-2438423, 2017.

[31] Curl: command line tool and library for transferring data with URLs. https://curl.haxx.se, 2017.

[32] Choosing the Optimal Segment Duration. https://streaminglearningcenter.com/blogs/choosing-the-optimal-segment-duration.html, 2016.

[33] Ana Nika, Yibo Zhu, Ning Ding, Abhilash Jindal, Y Charlie Hu, Xia Zhou, Ben Y Zhao, and Haitao Zheng. Energy and performance of smartphone radio bundling in outdoor environments. In *Proceedings of the 24th International Conference on World Wide Web*, pages 809–819. ACM, 2015.

[34] ExoPlayer 2 - Why, what and when? https://bit.ly/2Qcu4dd, 2016.

[35] ExoPlayer: Adaptive video streaming on Android - YouTube. https://www.youtube.com/watch?v=6VjF638VObA, 2014.

[36] ExoPlayer from the other side. https://bit.ly/2MDlgLd, 2016.

[37] WhatsApp For Android Devices. https://tech.blorge.com/2016/09/23/whatsapp-2-16-274-download-available/-android-devices-new-emojis/155538, 2016.

[38] Building Periscope for Android. http://nerds.airbnb.com/building-periscope-for-android/, 2017.

[39] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2012.

[40] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C Begen, and David Oran. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE Journal on Selected Areas in Communications*, 32(4):719–733, 2014.

[41] Xiufeng Xie, Xinyu Zhang, Swarun Kumar, and Li Erran Li. piStream: Physical layer informed adaptive video streaming over LTE. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 413–425. ACM, 2015.

[42] Hyunwoo Nam, Bong Ho Kim, Doru Calin, and HG Schulzrinne. Mobile video is inefficient: A traffic analysis. *Nexus*, 1:1–5, 2013.

[43] Ahmed Mansy, Mostafa Ammar, Jaideep Chandrashekar, and Anmol Sheth. Characterizing client behavior of commercial mobile video streaming services. In *Proceedings of Workshop on Mobile Video Delivery*, page 8. ACM, 2014.

[44] Christian Sieber, Poul Heegaard, Tobias Hoßfeld, and Wolfgang Kellerer. Sacrificing efficiency for quality of experience: Youtube's redundant traffic behavior. In *IFIP Networking Conference (IFIP Networking) and Workshops, 2016*, pages 503–511. IEEE, 2016.

[45] BBC DASH Testcard Stream. http://rdmedia.bbc.co.uk/dash/ondemand/testcard/, 2017.

[46] Yao Liu, Sujit Dey, Fatih Ulupinar, Michael Luby, and Yinian Mao. Deriving and validating user experience model for dash video streaming. *IEEE Transactions on Broadcasting*, 61(4):651–665, 2015.

[47] Chenghao Liu, Imed Bouazizi, and Moncef Gabbouj. Rate adaptation for adaptive http streaming. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 169–174. ACM, 2011.

[48] Tong Zhang, Fengyuan Ren, Wenxue Cheng, Xiaohui Luo, Ran Shu, and Xiaolan Liu. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in dash. In *Computer Communications, IEEE INFOCOM 2017-The 36th Annual IEEE International Conference on*. IEEE, 2017.

[49] Sintel - Open Movie by Blender Foundation. `https://durian.blender.org/download/`, 2017.

[50] YouTube's road to HTTPS. `https://youtube-eng.googleblog.com/2016/08/youtubes-road-to-https.html`, 2016.

[51] Irena Orsolic, Dario Pevec, Mirko Suznjevic, and Lea Skorin-Kapov. Youtube QoE estimation based on the analysis of encrypted network traffic using machine learning. In *Globecom Workshops (GC Wkshps), 2016 IEEE*, pages 1–6. IEEE, 2016.

[52] Wubin Pan, Gaung Cheng, Hua Wu, and Yongning Tang. Towards QoE assessment of encrypted YouTube adaptive video streaming in mobile networks. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)*, pages 1–6. IEEE, 2016.

[53] Irena Orsolic, Dario Pevec, Mirko Suznjevic, and Lea Skorin-Kapov. A machine learning approach to classifying YouTube QoE based on encrypted network traffic. *Multimedia tools and applications*, 76(21):22267–22301, 2017.

[54] M Hammad Mazhar and Zubair Shafiq. Real-time video quality of experience monitoring for HTTPS and QUIC. In *INFOCOM 2018-IEEE Conference on Computer Communications, IEEE*. IEEE, 2018.

[55] Paul Schmitt, Francesco Bronzino, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2020.

[56] Tarun Mangla, Emir Halepovic, Mostafa Ammar, and Ellen Zegura. emimic: Estimating http-based video qoe metrics from encrypted network traffic. In *IEEE/IFIP Conference on Traffic Measurement and Analysis 2018*, 2018.

[57] Qixiang Sun, Daniel R Simon, Yi-Min Wang, Wilf Russell, Venkata N Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 19–30. IEEE, 2002.

[58] Andrew Reed and Michael Kranch. Identifying HTTPS-protected Netflix videos in real-time. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 361–368. ACM, 2017.

[59] Andrew Reed and Benjamin Klimkowski. Leaky streams: Identifying variable bitrate DASH videos streamed over encrypted 802.11 n connections. In *Consumer*

Communications & Networking Conference (CCNC), 2016 13th IEEE Annual, pages 1107–1112. IEEE, 2016.

[60] Jiaxi Gu, Jiliang Wang, Zhiwen Yu, and Kele Shen. Walls have ears: Traffic-based side-channel attack in video streaming. In *INFOCOM 2018-IEEE Conference on Computer Communications, IEEE*. IEEE, 2018.

[61] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 605–616. ACM, 2012.

[62] Scott E Coull and Kevin P Dyer. Traffic analysis of encrypted messaging services: Apple imessage and beyond. *ACM SIGCOMM Computer Communication Review*, 44(5):5–11, 2014.

[63] Alfonso Iacovazzi, Andrea Baiocchi, and Ludovico Bettini. What are you Googling?-Inferring search type information through a statistical classifier. In *Global Communications Conference (GLOBECOM), 2013 IEEE*, pages 747–753. IEEE, 2013.

[64] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of tor hidden services. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[65] Andrew M White, Austin R Matthews, Kevin Z Snow, and Fabian Monrose. Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 3–18. IEEE, 2011.

[66] Quick Start Guide to Using Cronet. `https://chromium.googlesource.com/chromium/src/+/master/components/cronet`.

[67] Brad Miller, Ling Huang, Anthony D Joseph, and J Doug Tygar. I know why you went to the clinic: Risks and realization of HTTPS traffic analysis. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 143–163. Springer, 2014.

[68] RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. `https://tools.ietf.org/html/rfc5246`.

[69] RFC 8446 - The Transport Layer Security (TLS) Protocol Version 1.3. `https://tools.ietf.org/html/rfc8446#section-5.4`.

[70] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 332–346. IEEE, 2012.

[71] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review*, 44(4):187–198, 2015.

[72] Yanyuan Qin, Shuai Hao, Krishna R Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. Abr streaming of vbr-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 366–378. ACM, 2018.

[73] TV Lakshman, Antonio Ortega, and Amy R Reibman. VBR video: Tradeoffs and potentials. *Proceedings of the IEEE*, 1998.

[74] Big Buck Bunny. `https://peach.blender.org/`.

[75] Per-Title Encode Optimization. `https://bit.ly/2MHZaHr`.

[76] Jan De Cock, Aditya Mavlankar, Anush Moorthy, and Anne Aaron. A large-scale video codec comparison of x264, x265 and libvpx for practical vod applications. In *Applications of Digital Image Processing XXXIX*, volume 9971, page 997116. International Society for Optics and Photonics, 2016.

[77] MP4Box — GPAC. `https://gpac.wp.imt.fr/mp4box/`.

[78] Chrome Devtools. `https://developers.google.com/web/tools/chrome-devtools/`.

[79] Firefox Developer Tools. `https://developer.mozilla.org/en-US/docs/Tools`.

[80] Mobile App versus Mobile Website Statistics. `https://jmango360.com/wiki/mobile-app-vs-mobile-website-statistics/`, 2018.

[81] App share of total mobile minutes in leading online markets. `https://www.statista.com/statistics/692752/app-share-of-mobile-minutes-countries/`, 2017.

[82] The 2017 U.S. Mobile App Report. `https://www.comscore.com/Insights/Presentations-and-Whitepapers/2017/The-2017-US-Mobile-App-Report`, 2017.

[83] google/shaka-player: JavaScript player library / DASH client / MSE-EME player. `https://github.com/google/shaka-player`.

[84] Youtube brings us stats for nerds. `http://tubularinsights.com/youtube-stats-for-nerds/`.

[85] UI Automator. `https://developer.android.com/training/testing/ui-automator`.

[86] Test patterns — Netflix. https://www.netflix.com/title/80018499.

[87] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[88] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking (TON)*, 22(1):326–340, 2014.

[89] Youtube Data API. https://developers.google.com/youtube/v3/.

[90] Optimized shot-based encodes: Now Streaming! https://medium.com/netflix-techblog/optimized-shot-based-encodes-now-streaming-4b9464204830.

[91] youtube-dl: Download videos from YouTube. https://rg3.github.io/youtube-dl/.

[92] Arash Molavi Kakhki, Fangfan Li, David Choffnes, Ethan Katz-Bassett, and Alan Mislove. Bingeon under the microscope: Understanding t-mobiles zero-rating implementation. In *Proceedings of the 2016 workshop on QoE-based Analysis and Management of Data Communication Networks*, pages 43–48. ACM, 2016.

[93] Stream More Video, Use Less Data with Stream Saver - AT&T. https://www.att.com/offers/streamsaver.html.

[94] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. An internet-wide analysis of traffic policing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 468–482. ACM, 2016.

[95] UI Automator. https://linux.die.net/man/8/tc-tbf.

[96] Big buck bunny. https://peach.blender.org/.

[97] FBDOWN.net. https://www.fbdown.net/.

[98] Clipr - Twitch clip download. https://clipr.xyz/.

[99] How to block QUIC protocol. https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000ClarCAC.

[100] Yang Xu, Chenguang Yu, Jingjiang Li, and Yong Liu. Video telephony for end-consumers: measurement study of google+, ichat, and skype. In *Proceedings of the 2012 Internet Measurement Conference*, pages 371–384. ACM, 2012.

[101] DeckLink capture and playback cards. https://www.blackmagicdesign.com/products/decklink.

[102] Media — Android Open Source Project. https://source.android.com/devices/media.

[103] Mallesham Dasari, Santiago Vargas, Arani Bhattacharya, Aruna Balasubramanian, Samir R Das, and Michael Ferdman. Impact of device performance on mobile internet QoE. In *Proceedings of the Internet Measurement Conference 2018*, pages 1–7. ACM, 2018.

[104] Toward A Practical Perceptual Video Quality Metric. https://bit.ly/36bJPXr, 2016.

[105] How to Choose and Use Objective Video Quality Benchmarks. http://www.streamingmedia.com/Articles/ReadArticle.aspx?ArticleID=122050&PageNum=2.

[106] FFmpeg H.264 Video Encoding Guide. https://trac.ffmpeg.org/wiki/Encode/H.264.

[107] Samsung Video Encoder. http://androidxref.com/4.4.2_r2/xref/hardware/samsung_slsi/exynos5/exynos_omx/openmax/exynos_omx/component/video/enc/Exynos_OMX_Venc.c#318.

[108] Qualcomm's codec implementation. http://androidxref.com/4.4.2_r2/xref/hardware/qcom/media/mm-video-v4l2/vidc/venc/src/omx_video_base.cpp#4290.

[109] Kai Zeng, Tiesong Zhao, Abdul Rehman, and Zhou Wang. Characterizing perceptual artifacts in compressed video streams. In *Human Vision and Electronic Imaging XIX*, volume 9014, page 90140Q. International Society for Optics and Photonics, 2014.

[110] YUV. https://en.wikipedia.org/wiki/YUV.

[111] Longest increasing subsequence. https://en.wikipedia.org/wiki/Longest_increasing_subsequence.

[112] Why the Meteoric Rise of Google QUIC is Worrying Mobile Operators. https://owmobility.com/meteoric-rise-google-quic-worrying-mobile-operators/, 2018.

[113] Employing QUIC Protocol to Optimize Uber's App Performance. https://eng.uber.com/employing-quic-protocol/, 2019.

[114] Divyashri Bhat, Amr Rizk, and Michael Zink. Not so quic: A performance study of dash over quic. In *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 13–18. ACM, 2017.

[115] Sevket Arisu and Ali C Begen. Quickly starting media streams using quic. In *Proceedings of the 23rd Packet Video Workshop*, pages 1–6. ACM, 2018.

[116] goproxy README. https://github.com/elazarl/goproxy/blob/master/README.md.

[117] A QUIC implementation in pure Go. https://github.com/lucas-clemente/quic-go/blob/master/README.md.

[118] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. A first look at quic in the wild. In *International Conference on Passive and Active Network Measurement*, pages 255–268. Springer, 2018.

[119] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 160–166. ACM, 2017.

[120] Caddy - The HTTP/2 Web Server with Automatic HTTPS. https://caddyserver.com/.

[121] Shadowsocks - A secure socks5 proxy. http://www.shadowsocks.org/en/index.html.

[122] NetEm - Network Emulator. http://man7.org/linux/man-pages/man8/tc-netem.8.html.

[123] google/ExoPlayer: An extensible media player for Android. https://github.com/google/exoplayer.

[124] Ricky KP Mok, Weichao Li, and Rocky KC Chang. IRate: Initial video bitrate selection system for HTTP streaming. *IEEE Journal on Selected Areas in Communications*, 34(6):1914–1928, 2016.

[125] RKP Mok, EWW Chan, and RKC Chang. Improving TCP video streaming QoE by network QoS management. 2011.

[126] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. Shedding light on the structure of internet video quality problems in the wild. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 357–368. ACM, 2013.

[127] Ricky KP Mok, Edmond WW Chan, Xiapu Luo, and Rocky KC Chang. Inferring the qoe of http video streaming from user-viewing activities. In *Proceedings of the first ACM SIGCOMM workshop on Measurements up the stack*, pages 31–36. ACM, 2011.

[128] Gonca Gürsun, Mark Crovella, and Ibrahim Matta. Describing and forecasting video access patterns. In *INFOCOM, 2011 Proceedings IEEE*, pages 16–20. IEEE, 2011.

[129] S Shunmuga Krishnan and Ramesh K Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking*, 21(6):2001–2014, 2013.

[130] Mojgan Ghasemi, Partha Kanuparthy, Ahmed Mansy, Theophilus Benson, and Jennifer Rexford. Performance characterization of a commercial video streaming service. In *Proceedings of the 2016 ACM on Internet Measurement Conference*. ACM, 2016.

[131] Ashkan Nikravesh, Hongyi Yao, Shichang Xu, David Choffnes, and Z Morley Mao. Mobilyzer: An open platform for controllable mobile network measurements. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 389–404. ACM, 2015.

[132] Ashkan Nikravesh, David Ke Hong, Qi Alfred Chen, Harsha V Madhyastha, and Zhuoqing Morley Mao. QoE Inference Without Application Control. In *Internet-QoE@ SIGCOMM*, pages 19–24, 2016.

[133] Yihua Guo, Feng Qian, Qi Alfred Chen, Zhuoqing Morley Mao, and Subhabrata Sen. Understanding on-device bufferbloat for cellular upload. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, pages 303–317. ACM, 2016.

[134] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z Morley Mao, and Subhabrata Sen. An in-depth understanding of multipath TCP on mobile devices: Measurement and system design. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 189–201. ACM, 2016.

[135] Yihua Guo, Ashkan Nikravesh, Z Morley Mao, Feng Qian, and Subhabrata Sen. Accelerating multipath transport through balanced subflow completion. In *Proceedings of the 23th annual international conference on Mobile computing and networking*. ACM, 2017.

[136] Giorgos Dimopoulos, Ilias Leontiadis, Pere Barlet-Ros, Konstantina Papagiannaki, and Peter Steenkiste. Identifying the root cause of video streaming issues on mobile devices. In *Proceedings of the 2015 ACM International Conference on Emerging Networking Experiments and Technologies: 1-4 December, 2015: Heidelberg, Germany*. Association for Computing Machinery (ACM), 2015.

[137] Ibrahim Ayad, Youngbin Im, Eric Keller, and Sangtae Ha. A practical evaluation of rate adaptation algorithms in http-based adaptive streaming. *Computer Networks*, 133:90–103, 2018.

[138] Thomas Zinner, Stefan Geissler, Fabian Helmschrott, and Valentin Burger. Comparison of the initial delay for video playout start for different http-based transport protocols. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1027–1030. IEEE, 2017.

[139] Christian Timmerer and Alan Bertoni. Advanced transport options for the dynamic adaptive streaming over http. *arXiv preprint arXiv:1606.00264*, 2016.

[140] Mirko Palmer, Thorben Krüger, Balakrishnan Chandrasekaran, and Anja Feldmann. The quic fix for optimal video streaming. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. ACM, 2018.

[141] Géza Szabó, Sándor Rácz, Daniel Bezzera, Igor Nogueira, and Djamel Sadok. Media QoE enhancement with QUIC. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 219–220. IEEE, 2016.