QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis

Qi Alfred Chen, Haokun Luo, Sanae Rosen, Z. Morley Mao,
Karthik Iyer†, Jie Hui†, Kranthi Sontineni†, Kevin Lau†
University of Michigan, †T-Mobile USA Inc.[1]
{alfchen,haokun,sanae,zmao}@umich.edu,
†{karthik.iyer,jie.hui,kranthi.sontineni1,kevin.lau}@t-mobile.com

ABSTRACT

Smartphones have become increasingly prevalent and important in our daily lives. To meet users' expectations about the Quality of Experience (QoE) of mobile applications (apps), it is essential to obtain a comprehensive understanding of app QoE and identify the critical factors that affect it. However, effectively and systematically studying the QoE of popular mobile apps such as Facebook and YouTube still remains a challenging task, largely due to a lack of a controlled and reproducible measurement methodology, and limited insight into the complex multi-layer dynamics of the system and network stacks.

In this paper, we propose *QoE Doctor*, a tool that supports accurate, systematic, and repeatable measurements and analysis of mobile app QoE. QoE Doctor uses UI automation techniques to replay QoE-related user behavior, and measures the user-perceived latency directly from UI changes. To better understand and analyze QoE problems involving complex multi-layer interactions, QoE Doctor supports analysis across the application, transport, network, and cellular radio link layers to help identify the root causes. We implement QoE Doctor on Android, and systematically quantify various factors that impact app QoE, including the cellular radio link layer technology, carrier rate-limiting mechanisms, app design choices and user-side configuration options.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Measurement techniques]; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*

General Terms

Design, Measurement, Performance

Keywords

Quality of Experience (QoE); UI Automation; Cross-layer Analysis; Mobile Applications; Cellular Network

[1]The views presented in this paper are as individuals and do not necessarily reflect any position of T-Mobile.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IMC'14, November 5–7, 2014, Vancouver, BC, Canada.
Copyright 2014 ACM 978-1-4503-3213-2/14/11 ...$15.00.
http://dx.doi.org/10.1145/2663716.2663726.

1. INTRODUCTION

As smartphones become more prevalent, mobile applications (apps) become increasingly important to our daily lives, providing access to information, communication, and entertainment. Users would like apps to respond quickly to their requests, consume less mobile data to reduce their monthly bill, and consume less energy to ensure sufficient battery life. The degree to which apps meet these user expectations is referred to as QoE (Quality of Experience). Ensuring good QoE is crucial for app developers, carriers, and phone manufacturers to sustain their revenue models; thus it is essential to obtain a comprehensive understanding of app QoE and the critical factors that affect QoE.

However, it remains a challenging task to effectively and systematically study the QoE of popular mobile apps, such as Facebook and YouTube. Prior work were relied on user studies or app logs to evaluate QoE through *subjective* metrics such as user experience scores and user engagement [20, 24, 23, 18, 17], but these experiments are either costly in human effort or less able to control user behavior variations. To overcome these limitations, Prometheus [15] measures *objective* QoE metrics, such as the video rebuffering ratio, to eliminate the dependence on user behavior, but it requires the application source code to log UI events, limiting its applicability. Besides the methodology, another challenge is that mobile app QoE is affected by factors at many layers of the system and the network. For example, on cellular networks, the radio link layer state machine transition delay can lead to longer round-trip times, and thus increase user-perceived latency [35, 34]. These multi-layer dynamics and their inter-dependencies further complicate QoE analysis.

To address these challenges, we design a tool called QoE Doctor to support more accurate, systematic, and repeatable measurements and analysis of mobile app QoE. QoE Doctor uses UI automation techniques to replay user behavior such as posting a status on Facebook, and at the same time measures the application-layer user-perceived latency directly through UI changes on the screen. Our tool does not require access to the application source code, or modifications to the app logic or the underlying system, making it applicable to QoE measurements of popular apps. In addition to QoE measurements, QoE Doctor supports cross-layer analysis covering the application layer, transport layer, network layer, and radio link layer, in order to understand the root causes of poor QoE caused by network activities and device-specific operations.

We implement QoE Doctor on the Android platform, and systematically measure and analyze various QoE metrics in popular Android apps, including Facebook's post upload time and pull-to-update time, the initial loading time and rebuffering ratio in YouTube videos, and the web page loading time in popular Android browsers. We quantitatively evaluate the important factors

impacting these QoE metrics, for example network conditions, application and carrier. Some of our key findings are:

- Network latency is not always on the critical path of the end-to-end user-perceived latency, such as when posting a status on Facebook.
- Changing one Facebook default configuration can reduce over 20% of mobile data and energy consumption.
- Carrier rate limiting policies can increase video loading time by more than 30 seconds (15×) and increase the rebuffering ratio from almost 0% to 50%.
- YouTube ads reduce the initial loading time of the main video, but on cellular networks the total loading time is doubled.
- The ListView version Facebook reduces device latency by more than 67% (compared to the WebView version), network latency by more than 30%, and downlink data consumption by more than 77%.
- Simplifying the 3G RRC state machine can reduce web page loading time by 22.8% for web browsing apps.

Our contributions in this paper are summarized as follows:

• To enable automated and repeated QoE data collection for mobile apps, we design a QoE-aware UI controller, which is able to replay QoE-related user interaction sequences on popular Android apps and directly measure user-perceived latency through UI changes.

• We design a multi-layer QoE analyzer, which provides visibility across the application layer, transport layer, network layer, and radio link layer, helping us systematically diagnose QoE problems and identify the root causes.

• We use QoE Doctor to measure QoE metrics in popular Android apps, and quantify how various important factors impact these QoE metrics.

For the rest of the paper, we first provide background information in §2 and a overview of QoE Doctor in §3. In §4 and §5, we describe two major parts of QoE Doctor: a QoE-aware UI controller and a multi-layer QoE analyzer respectively, and §6 summarizes the current tool limitations. In §7 we use QoE Doctor to systematically study various factors impacting mobile app QoE. We summarize related work in §8, and conclude the paper in §9.

2. BACKGROUND

Subjective and objective QoE metrics. QoE (Quality of Experience) refers to the metrics end users use to judge the quality of services they receive, such as web browsing, phone calls or TV broadcasts. There is a strong incentive for these services to maintain and improve QoE, as it is essential to their continued financial success. To assess the QoE perceived by the end users, one approach is to ask users to score the service experience, which we call a *subjective QoE metric*. Another approach is to directly measure service performance metrics that are related to user satisfaction, such as the number of stalls when watching a video, which we call *objective QoE metrics*. Much of the previous work in this area [20, 42, 17, 24, 18] has focused on subjective metrics. However, subjective evaluations usually require user studies that are hard to repeat and automate, and may be hard to reproduce due to varying user behavior. Thus, in this paper we focus on objective QoE metrics.

RRC/RLC. In order to understand the root causes of QoE problems on mobile devices, it is important to understand how various performance problems in the network stack can affect app QoE. Of particular interest are the RRC (Radio Resource Control) radio link layer control plane messages used by the base station[1] to coordinate with the device. RRC state behavior has a significant impact on app performance and power consumption [22, 33, 40, 41].

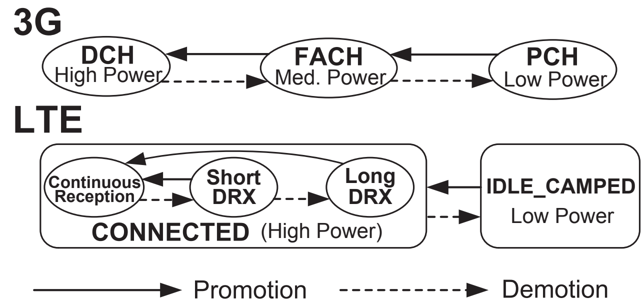

Figure 1: 3G and LTE RRC state machine overview.

Typically, 3G has three main RRC states: DCH, FACH and PCH; and LTE has CONNECTED and IDLE_CAMPED as shown in Fig. 1. DCH and CONNECTED are high-power, high-bandwidth states with dedicated communication channels, and PCH and IDLE_CAMPED are low-power states with no data-plane radio communication. FACH is an intermediate state with a lower-bandwidth shared communication channel. The device promotes from a low-power state to a high-power state if there is a data transfer, and demotes from high-power state to low-power state when a demotion timer expires.

We also examine the layer 2 data plane protocol, RLC (Radio Link Control) [14] . The smallest data transmission unit in RLC is called a PDU (Protocol Data Unit). For 3G uplink traffic, the PDU payload size is fixed at 40 bytes, while for 3G downlink traffic and all LTE traffic the size is flexible and usually greater than 40 bytes. As shown in Fig. 2, an ARQ (automatic repeat request) mechanism is used for reliable data transmission, which is similar to the TCP group acknowledgement mechanism but triggered by a polling request piggybacked in the PDU header.

3. OVERVIEW

In this paper, we develop a tool named *QoE Doctor* to support automated and repeated measurements of objective QoE metrics for popular mobile apps directly from the user's perspective, as well as systematically study various factors influencing these QoE metrics across multiple mobile system and network layers. In this section, we first introduce the target QoE metrics, and then provide an overview of the tool design.

3.1 QoE Metrics

In this paper, we study three important objective mobile app QoE metrics that directly influence user experience:

• **User-perceived latency.** This application-level QoE metric is defined as the time that users spend waiting for a UI response from the app. This includes the web page loading time in web browsers, the post upload time in social apps, and the stall time in video streaming apps.

• **Mobile Data consumption**. On mobile platforms, cellular network data can be expensive if a data limit is exceeded. Thus, for end users mobile data consumption is an important component of mobile app QoE [23].

• **Energy consumption**. Smartphones are energy-constrained devices, thus energy efficiency is a desired feature in mobile

[1]known as the *Node B* for 3G and the *eNodeB* for LTE

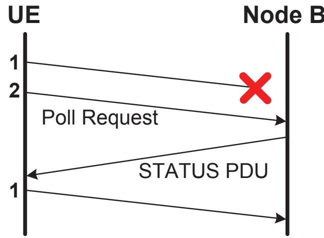

Figure 2: RLC PDU transmission with ARQ-based group acknowledgment mechanism

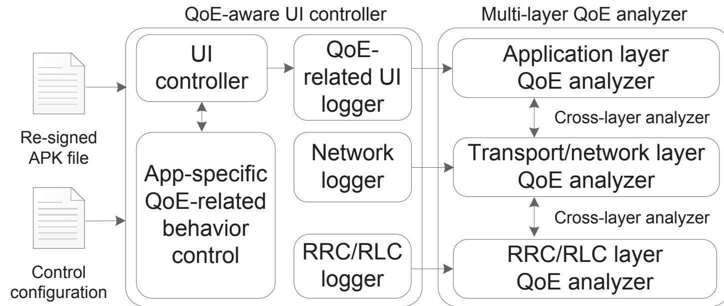

Figure 3: QoE Doctor design overview

apps [23]. In particular, we focus on the network energy consumption of mobile apps since it consumes a large share of the total device energy [19] and it is strongly influenced by app design choices [35, 32].

Among these 3 metrics, user-perceived latency is the most direct way for mobile end users to judge app performance. Thus, it is the main focus of this paper. Unlike previous work [39, 15, 47], our measurement approach (described in §4) directly calculates the latency from user's perspective — the UI layer, without requiring application source code or any OS/application logic instrumentation, which enables us to study this QoE metric broadly on any popular mobile apps of interest.

The other two QoE metrics, mobile data and energy consumption, are more mobile platform specific. Unlike previous work [43, 30, 35, 34], our analysis is driven by automatically replaying user behavior from the application layer. This enables us to study these QoE metrics from the user's perspective, and repeat experiments in a controlled manner.

3.2 Tool Design Overview

Fig. 3 shows the design of QoE Doctor. It includes 2 major components: a QoE-aware UI controller and a multi-layer QoE analyzer.

QoE-aware UI controller. This component runs online on the mobile device, and uses UI control techniques to drive Android apps to automatically replay user behavior traces, while collecting the corresponding QoE data at the application layer, the transport layer, the network layer, and the cellular radio link layer. This allows us to efficiently collect QoE data, and enables controlled QoE measurements without depending on varying user behavior. Unlike previous work [15], our UI control technique does not require access to application source code. Thus, QoE Doctor is able to support QoE analysis for popular Android apps such as Facebook and YouTube. At the UI layer, to accurately collect user-perceived latency data, our UI controller supports direct access to the UI layout tree. UI layout tree describes the app UI on the screen in real time and thus can be used to accurately record the time a UI change is made. We use `tcpdump` to collect network data, and a cellular radio link layer diagnosing tool from Qualcomm to collect radio link layer control plane (RRC) and data plane (RLC) data.

Multi-layer QoE analyzer. The collected QoE data are processed and analyzed offline in this component with multi-layer visibility. At the UI layer, user-perceived latency is calculated using the timestamps of each QoE-related UI event. At the transport and network layers, TCP flow analysis is used to separate network behaviors from different apps based on DSN requests and TCP flow data content. TCP flow analysis is also used to compute mobile data consumption corresponding to the QoE-related user behavior in the application layer. Mobile energy consumption is estimated based on the cellular network behavior according to the RRC state recorded in the radio link layer tool log. To more deeply analyze the QoE measurement results, our analyzer supports cross-layer mapping between the UI layer and the transport/network layer, and between the transport/network layer and the RRC/RLC layer. This allows us to better understand how user actions in the UI layer are delayed by the network traffic, and helps us identify the potential bottleneck in the cellular radio link layer that limits the TCP/IP data transmission speed.

4. QOE-AWARE UI CONTROLLER

In this section, we describe how QoE-related user behavior is automatically replayed and how the corresponding data is collected in QoE Doctor's QoE-aware UI controller.

4.1 Application control

As shown in Fig. 3, the QoE data collection in QoE Doctor is driven by a UI controller. This component initiates UI interactions such as button clicks, and thus controls the app automatically to perform user behaviors of interest. It eliminates human effort, and allows the same set of standardized user interactions to be replayed each time. In our implementation on Android, we control the app UI through the InstrumentationTestCase API [2] provided by the Android system for UI testing during app development. It allows UI interaction events to be sent to the app during testing, and the only requirement is to re-sign the binary APK file by our debugging key. Our work is the first to use the InstrumentationTestCase API for the purpose of automated QoE analysis.

UI control paradigm. The UI control in QoE Doctor follows a *see-interact-wait* paradigm. After launching the app, the *see* component first parses the app UI data shown on the screen, then the *interact* component chooses a UI element to interact with (*e.g.,* by clicking a button or scrolling a page). After the interaction, the *wait* component waits for the corresponding UI response. This paradigm follows natural user-app interaction behavior, allowing us to replay real user behavior. Using the InstrumentationTestCase API, the controller is launched in the same process as the controlled app, allowing direct access to the UI data as needed for the *see* and *wait* components. Unlike prior work which require system instrumentation or Android UI dump tools [21, 26, 29, 36], direct UI data sharing enables convenient and accurate latency measurements (described next).

The *Wait* component and accurate user-perceived latency measurement. In our *see-interact-wait* paradigm, the *wait*

component measures the user-perceived latency: the time between triggering a UI interaction event and receiving the corresponding UI response. Thus, in our controller we log the start and end timestamps of the waiting process to measure user-perceived latency. The waiting process can either be triggered (1) by the user (*e.g.,* uploading a post in Facebook), or (2) by the app (*e.g.,* a video stall). To log the start timestamp, for (1) we log the time when the controller triggers the user action, and for (2) we log the time when the waiting process indicator (*e.g.,* a progress bar) shows up. For the end timestamp, we log the time when the wait-ending UI indicator occurs (*e.g.,* the progress bar's disappearance). As the controller shares the same process as the app, these UI element events can be monitored directly.

User behavior replay. We select several popular Android apps and identify critical QoE-related user behavior, along with the corresponding user interaction sequence. Based on the interaction sequences, control specifications are written for the UI control logic implementation. To write the specification, only some familiarity with Android UI View classes is required, so the average Android app developer should be able to do so. In our design, we support replaying the user interaction sequences both with and without replaying the timing between each action. In §7 we use both according to the experiment requirements.

To ensure the user interactions are sent to the right UI elements during replay, we design a View signature describing the View characteristics in the UI layout tree. This signature includes the UI element class name, View ID, and a description added by the developer. To support different Android devices, the Vew element coordinates are not included in this signature.

4.2 App-specific Control Design and User-perceived Latency Collection

According to a recent report [6], social networking app Facebook and video app Youtube are the top 2 mobile applications used in North America during peak periods, and web browsing is ranked the third in the amount of aggregate mobile traffic globally. Thus, we focus on these applications. Table 1 summarizes the QoE-related user behavior that QoE Doctor replays and the associated user-perceived latency metrics.

4.2.1 Facebook

For Facebook, we measure the following two user actions:

Upload post. One of the most common user actions on Facebook is to post an update: posting a status, check-in, or uploading a photo. For these actions, the user-perceived latency is the time from when the "post" button is clicked to the time when the posted item is shown on the news feed list. To measure this ending time, we put a timestamp string in the post, and after the "post" button is clicked, the *wait* component repeatedly parses the UI layout tree and logs the end timestamp as the time when the item with the timestamp string appears in the news feed.

Pull-to-update. Another common user action on Facebook is to pull the news feed list down to update it. This can be generated either by: (1) a pulling gesture, or (2) passively waiting for Facebook to update the news feed list by itself. To replay the former, our controller generates a scrolling down gesture. For the latter, the controller just waits on the Facebook news feed list, and uses the *wait* component to log the appearance and disappearance time of the loading progress bar for the news feed list.

4.2.2 YouTube

For YouTube, we replay searching for a video by name and then watching it.

Watch video. To replay this user behavior, the controller takes as input a list of video names. It searches for the video and plays it until it finishes. There are two user-perceived latency metrics the *wait* component monitors: the initial loading time, and the rebuffering ratio. The rebuffering ratio is the ratio of time spent stalling to the sum of total play and stall time after the initial loading. For the initial loading time, we start measuring when the controller clicks on a video entry in the search results, and finish measuring when the loading progress bar disappears. For the rebuffering ratio, the controller parses the UI layout tree after the video starts playing, and logs when the progress bar appears and disappears as the video rebuffering start and end timestamps.

When an advertisement (ad) is shown before the video, we measure the initial loading time and rebuffering ratio for the ad and the actual video respectively. We configure the controller to skip any ads whenever users are given that option, as a recent study shows that 94% of users skip these ads [1].

4.2.3 Web Browsing

For web browsing apps, we choose Google Chrome and Mozilla Firefox, which both have more than 50 million downloads in Google Play, along with the default Android browser (named "Internet"). For these browsers, we replay loading a web page.

Load web page. For web browsing, the most important performance metric is the web page loading time. To replay page loading, our controller takes a file with a list of URL strings as input, and enters each URL into the URL bar of the browser app line by line before sending an ENTER key. The *wait* component logs the ENTER key sending time as the start time, and monitors the progress bar in the UI layout tree to determine when the loading completes. A more accurate way of determining the loading complete time would be to log the time when the visible parts on the page are loaded, for example by capturing a video of the screen and then analyzing the video frames as implemented in Speed Index metric for WebPagetest [8]. We plan to support this in our future work by adding a screen video capturing into the controller, and supporting video frame analysis in the application layer analyzer.

4.3 Data Collection

While replaying user behavior to measure QoE, the UI controller collects data at multiple layers.

4.3.1 Application Layer Data Collection

Application layer QoE data is collected by the *wait* component during the user behavior replay. The controller generates a log file, called `AppBehaviorLog`, which records each user interaction event sent to the controlled app, in particular the start and end timestamps to calculate the user-perceived latency. The user interaction for each app and the corresponding UI elements for user-perceived latency measurements are described in §4.2, and summarized in Table 1.

4.3.2 Transport/Network Layer Data Collection

To measure mobile data consumption and help identify the root causes of QoE problems for other layers, our controller collects traffic logs at the transport and network layers using `tcpdump` [9] during the user behavior replay.

4.3.3 RRC/RLC Layer Data Collection

To collect RRC/RLC layer data, we use QxDM (Qualcomm eXtensible Diagnostic Monitor), a cellular network diagnosis tool from Qualcomm [12]. This tool provides real-time access to both RRC and RLC layer information for all commercial handsets with a

Application	User behavior to replay	User-perceived latency to measure	UI events to monitor for latency measurement	
			Measurement start time	Measurement end time
Facebook	Upload post	Post uploading time	Press "post" button	Posted content shown in ListView
	Pull-to-update	News feed list updating time	Progress bar appears	Progress bar disappears
YouTube	Watch video	Initial loading time	Click on the video entry	Progress bar disappears
		Rebuffering time	Progress bar appears	Progress bar disappears
Web browsing	Load web page	Web page loading time	Press ENTER in URL bar	Progress bar disappears

Table 1: Replayed user behavior and user-perceived latency metrics for Facebook, YouTube and web browsers

Qualcomm chipset, including popular Android phone models such as Samsung Galaxy S3/S4. Recent work [44] exposes RRC state on Intel/Infineon XGold chipsets by instrumenting the system, but it cannot access RLC data, which is critical in our cross-layer analysis in §5.4.2. To collect this data, we need to connect the mobile devices to a Windows PC, and configure the QxDM software for the corresponding network technology, either 3G or LTE. QxDM saves all radio link layer information to a file on the PC. By parsing this information, we isolate the RRC state transitions and RLC PDUs with corresponding timestamps.

QxDM limitations. There are two major limitations of the QxDM tool. First, as mentioned earlier, it requires a PC connection in order to collect real-time user data outside the laboratory. QoE Doctor helps reduce the impact of this limitation as we can replay real user behavior using the UI controller. Second, the QxDM tool is not designed to provide complete RLC PDU payload information. Perhaps to reduce logging overhead, the RLC PDUs only contain 2 payload bytes, which makes the cross-layer mapping between RLC PDUs and network packets non-trivial. We explain in §5.4.2 how we use a technique we call *long-jump mapping* to overcome this limitation.

5. MULTI-LAYER QOE ANALYZER

In this section, we describe how QoE metrics are calculated at each layer using the data collected by the UI controller, and what multi-layer analysis is supported to help study these QoE metrics.

5.1 Application Layer Analyzer

At the application layer, we can simply calculate the user-perceived latency metrics based on the start and end timestamps logged in the `AppBehaviorLog` by the UI controller. Unlike previous work [22, 31], we calculate UI latency directly rather than inferring it from network traces, which can capture the ground truth, and also enables us to analyze apps having encrypted traffic such as Facebook.

User-perceived latency calibration. We measure user-perceived latency by observing app-generated UI events by periodically parsing the UI layout tree. Fig. 4 shows the process for measuring a Facebook post upload, where the start timestamp is measured from a UI controller action and the end timestamp is measured from the UI layout tree. In QoE Doctor we want to measure t_{ui}, but due to overhead in parsing the UI tree, the actual measured latency is $t_m = t_{ui} + t_{offset} + t_{parsing}$. To accurately calculate t_{ui}, we need to subtract t_{offset} and $t_{parsing}$ from t_m. Assuming the end time of the UI data update falls uniformly in the parsing time interval, the expected value of offset time is $\overline{t}_{offset} = \frac{1}{2} t_{parsing}$. We calibrate the user-perceived latency by subtracting $\frac{3}{2} t_{parsing}$ from t_m. This calibration is used to correct the post uploading time, the web page loading time, and the initial loading time in Table 1. For the other two latency metrics, the start timestamp is measured by parsing the UI tree, which is the same as the end timestamp measurement, so the average offset time is $\overline{t}_{offset} = 0$. For these, we just subtract $t_{parsing}$ from t_m in the calibration.

5.2 Transport/Network Layer Analyzer

In the transport and network layers, we calculate the mobile data consumption QoE metric from data collected by `tcpdump`. Our analyzer parses the raw packet trace and extracts TCP flows, defined by the tuple {srcIP, srcPort, dstIP, dstPort}, and then associates each TCP flow with the server's URL by parsing the DNS lookups in the trace. Over repeated experiments, we identify the TCP flows with common server URLs to associate the replayed user behavior, and then calculate its network data consumption. We also calculate the number of data retransmissions, the RTT (Round-Trip Time), and the throughput for the TCP flows.

5.3 RRC/RLC Layer Analyzer

We obtain RRC state change information from QxDM logs. Using the Monsoon Power Monitor [7], we obtain the power level for each RRC state, and thus can calculate the network energy consumption for the entire mobile device using a technique from previous work [22]. To get the network energy consumed by the controlled app only, we remove all other apps on the device and log out all system app accounts to ensure that the network energy calculation is not affected by unrelated traffic. We also calculate tail energy as defined in previous work [34], and count all other network energy as non-tail energy.

First-hop OTA RTT. QxDM logs both uplink and downlink RLC PDUs, which include polling PDUs and the corresponding STATUS PDUs, as mentioned in §2. Based on this feedback loop, we calculate the first-hop OTA (Over The Air) RTT, which is the time from when the device transmits a polling request until when it receives the STATUS PDU. However, because of the group acknowledgement mechanism, we may not find a corresponding polling PDU for each STATUS PDU. Thus, we estimate the first-hop OTA RTT for a PDU by finding the nearest polling PDU to a STATUS PDU.

5.4 Cross-layer Analyzer

Besides analyzing data at individual layers, QoE Doctor also supports cross-layer analysis across the application, transport, network, and RRC/RLC layers to help perform root cause analysis.

5.4.1 Cross Application, Transport/Network Layers

To help identify root causes of QoE problems in the application layer, we examine the transport and network layer behavior to identify the critical path of the application layer delay and pinpoint the bottlenecks. We first identify the start and end time of a user-perceived latency problem logged in the `AppBehaviorLog`, which forms a *QoE window*. Then we focus our analysis on the network traffic which falls into this QoE window, and use flow analysis to identify the TCP flows responsible for the application layer delay. Through this cross-layer analysis, we can study fine-

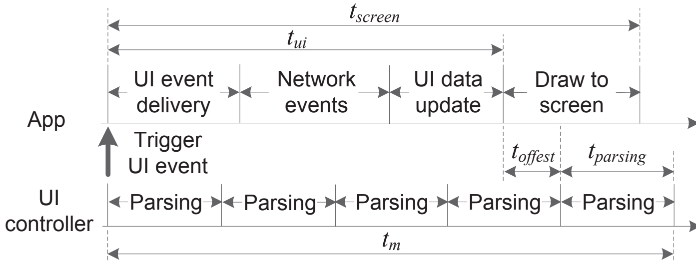

Figure 4: User-perceived latency measurement for uploading a post on Facebook

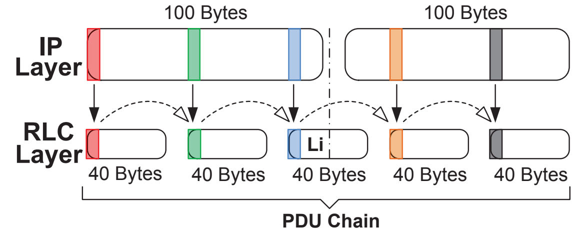

Figure 5: *Long-jump mapping* algorithm for cross-layer mapping from IP packets to a RLC PDU chain. The payload of the third PDU is a combination of the tail of the first IP packet and the head of the second IP packet.

grained QoE metrics such as the initial loading time for Youtube, which is otherwise indistinguishable from rebuffering events if only analyzing network data, as done in previous work [18, 42].

5.4.2 Cross Transport/Network, RRC/RLC Layers

To understand the impact of the interaction between the transport/network and the cellular radio link layers, QoE Doctor supports cross-layer analysis between them.

Transport/network layer and RRC layer. From the RRC state information logged by QxDM, we obtain the RRC state change information. By finding an overlap between the QoE window (defined in §5.4.1) and the RRC state transition window, we can pinpoint cases where RRC state transitions occur during the period of the user-perceived latency, which may help reveal the impact of RRC state transitions on the user-perceived latency.

Transport/network layer and RLC layer. To understand how network packets are transmitted in the lower layer, our analyzer supports the mapping from IP packets to RLC PDUs using the fine-grained RLC transmission information provided by QxDM. More specifically, we map complete IP packets to the corresponding fragmented RLC payload data bytes spreading among several PDUs. Due to the limitation of QxDM mentioned in §4.3.3, for each PDU only the first 2 payload bytes are logged, which provides us with limited information to identify the corresponding IP packet.

To ensure an accurate mapping, we design an algorithm which handles these limitations we have mentioned. As only 2 payload bytes are captured, after matching these 2 bytes we skip over the rest of the PDU, and try to match the first 2 payload bytes in the next PDU as shown in Fig. 5, which we call *long-jump mapping*. Since some PDUs may contain the payload data belonging to two consecutive IP packets, according to the 3G specification [14] we use the LI (Length Indicator) to map the end of an IP packet. If the cumulative mapped index equals the size of the IP packet, we have found a mapping successfully; otherwise no mapping is discovered.

We evaluate this mapping, and find that the percentage of mapped IP packets is 99.52% for uplink and 88.83% for downlink. The reason that we cannot achieve 100% accuracy is that occasionally a small fraction of RLC PDUs are not captured by QxDM, causing missing mappings for the corresponding IP packets. In our cross-layer analysis, we only consider the IP packets with successfully mapped RLC PDUs.

6. TOOL LIMITATIONS

The limitations of QoE Doctor are summarized as follows.

Manual efforts involved in the replay implementation. In order to replay user behavior, QoE Doctor currently requires manual identification of critical QoE-related user behavior, and some familiarity with Android UI View classes for writing control specifications. These manual efforts are necessary, as measuring the QoE metric of interest requires identifying the natural way a user interacts with the app. In future work, we will consider using learning algorithms to automatically generate common user behavior from user study logs.

Latency measurement imprecision. As shown in Fig. 4, although we are directly measuring the UI data changes (t_{ui}), the result may differ from the UI changes on the screen (t_{screen}), mostly due to the UI drawing delay. In §7.1 we find the measurement error is less than 4%.

Lack of fine-grained application layer latency breakdown. In our latency measurements, only the end-to-end latency is reflected in the UI layout data changes, and we cannot break down the latency into more fine-grained operations such as the transaction task delay and the inter-process communication delay. Without the ability to track detailed UI operation sequences, it is hard to confirm whether the network activity is asynchronous, which may mislead the cross-layer analysis. In our design, we do not support it because providing this information requires system or application logic instrumentation [39, 47], which may limit the tool's applicability.

Limitation related to RRC/RLC layer logger. Our analysis about layer 2 data plane (RLC) information may not be 100% accurate due to QxDM limitations as described in §4.3.3 and §5.4.2.

7. EXPERIMENTAL EVALUATION

In this section, we first evaluate the QoE measurement accuracy and overhead of QoE Doctor, and then use QoE Doctor to systematically and quantitatively study various factors which may have impact on our 3 QoE metrics. In this paper, we consider three factors potentially impacting QoE: (1) the network type and quality, for WiFi and 3G and LTE cellular networks; (2) the app's design and configuration; and (3) the carrier, in particular carrier rate limiting mechanisms for throttling. These factors are identified based on either our experiences with the apps or practical problems from real user experience identified by the authors at T-Mobile. We summarize our experiment goals in Table 2. In this section, 2 carriers are involved in our experiments, which we denote as C1 and C2.

7.1 Tool Accuracy and Overhead

In this section, we report our evaluation results of the accuracy and overhead of QoE Doctor. Table 3 summarizes the results in the section along with the IP packet to RLC PDU mapping performance reported in §5.4.2.

Section	Experiment goal	Relevant factor	Application
§7.2	Device and network delay on the critical path for user-perceived latency	Network condition, app	Facebook
§7.3	Data and energy consumption during application idle time	Network condition, app	Facebook
§7.4	Impact of app design choices on user-perceived latency	Network condition, app	Facebook
§7.5	Impact of carrier throttling mechanisms on user-perceived latency	Network condition, carrier	YouTube
§7.6	Impact of video ads on user-perceived latency	Network condition, app	YouTube
§7.7	Impact of the RRC state machine design on user-perceived latency	Network condition, carrier	Web browsers

Table 2: Experiment goals and mobile applications studied in §7.

Item	Value
User-perceived latency measurement error	$\leq$40 ms (t_d)
	$\leq$4% (t_d/t_{screen})
Transport/network to RLC data mapping ratio	99.52% (uplink)
	88.83% (downlink)
CPU overhead	6.18%

Table 3: Tool accuracy and overhead summary of QoE Doctor.

QoE measurement accuracy. The mobile data consumption metric is a precise value, calculated directly from data in the transport/network layer. The network energy consumption is calculated directly from RRC/RLC layer information using a well-established model [22, 48]. For the user-perceived latency metric, however, the UI data changes in the UI layout tree may not precisely correspond to screen changes due to UI drawing delays (Fig. 4). We evaluate the measurement accuracy by recording a video of the screen at 60 frames per second for each of the user-perceived latency tests we perform. Each experiment is repeated 30 times. The result shows that for all actions, the average time difference t_d between t_{screen} and the measurement result from QoE Doctor is under 40 milliseconds. We determine how the user-perceived latency measurements are affected by calculating the ratio of t_d to t_{screen} for each metric, as shown in Fig. 6. As t_d is not proportional to t_{screen}, the ratio differences between the 5 metrics are due to t_{screen}. To calculate an upper bound on this ratio, for each metric we use the shortest t_{screen} among all the experiments in this section in Fig. 6. As shown, for all experiment results the latency measurement error is less than 4%.

QoE measurement overhead. We use DDMS [3] to compare the CPU time when manually inputting the target user behavior with the CPU time when using QoE Doctor. We find the upper bound of this overhead by running the most compute-intensive operation, parsing the UI tree (Fig. 4), on the most computation-intensive app operation: uploading a Facebook post. We run this test 30 times, and find the average worst-case CPU computation overhead introduced by QoE Doctor is 6.18%.

7.2 Facebook: Post Uploading Time Breakdown Analysis

In this section, we focus on the action of uploading a post to Facebook, leveraging our multi-layer analysis to break down the roles the device and the network play in the user-perceived latency.

Experiment setup. We run the experiments on Facebook version 5.0.0.26.31 on a Samsung Galaxy S3 device with Android 4.3. We use QoE Doctor to post status, check-in, and 2 photos every 2 seconds for C1 3G and C1 LTE network. Each action is repeated 50 times.

Finding 1: The network delay is not always on the critical path. To understand the role of the device and the network in the end-to-end delay, we break down the device and network latency according to the steps of uploading a post shown in Fig. 4. To separate out the network latency portion, we first identify the TCP flows which are responsible for the post uploading using techniques described in §5.4.1. Even though the trace is encrypted, it is not hard to identify the flow with high confidence since in most cases only one flow has traffic during the QoE window (defined in §5.4.1). We then calculate the network latency as the timestamp difference between the earliest and latest packet of this TCP flow, and calculate the device latency by subtracting the network latency from the user-perceived latency.

Fig. 7 shows the breakdown results for posting 2 photos, status and check-in for C1 3G and C1 LTE network. In the figure, the standard deviation values of the latencies for posting 2 photos are all less than 0.7 seconds, and those for posting a check-in and a status are all less than 0.25 seconds. Surprisingly, we find that the network delay contributes little to the check-in and status uploading latency. We double-checked the network trace, and found that the corresponding TCP ACK packets for both actions are actually outside the QoE window. This indicates that showing these posts on the news feed list doesn't depend on the response from Facebook server. In other words, the *Facebook app pushes a local copy of status and check-in posts directly onto the news feed list to remove the network delay from the critical path, which ensures low user-perceived latency*. Note that this only happens to posting a status and a check-in; for posting 2 photos, the network latency always falls inside the QoE window, suggesting that it is very likely to be on the critical path, which is the case described in Fig. 4.

Finding 2: 3G RLC transmission delay contributes more than expected in the end-to-end photo posting time. Unlike status and check-in posting, Fig. 7 shows that for 2 photo uploading action the network latency has more than 65% share in the end-to-end latency. Using our cross-layer analysis between the application and transport/network layers, we always see a clear pattern of uploading then downloading two large chunk of data in the TCP flow inside QoE Window. Interestingly, for this action 3G has around 50% more network latency than LTE, while their device latencies are similar. To find out the reason, we further break down the network latency using our cross-layer analyzer between the transport/network layer and the RRC/RLC layer as described in §5.4.1.

In this more fine-grained network latency breakdown, we target four metrics: *IP-to-RLC delay* (t_1), *RLC transmission delay* (t_2), the *first-hop OTA delay* (t_3), and *other delay* (t_4) as shown in Fig. 9. *The IP-to-RLC delay* is the time difference between an IP packet and its first mapped RLC PDU when no other RLC PDUs are transmitted. For the *RLC transmission delay*, we first identify the periods where the device is "busy" transmitting RLC PDUs using a burst analysis for RLC PDUs. We implement this analysis by checking whether the inter-PDU time is less than the estimated first-hop OTA RTT defined in §5.3. Then, the *RLC transmission delay* is calculated by summing up all the inter-PDU time within

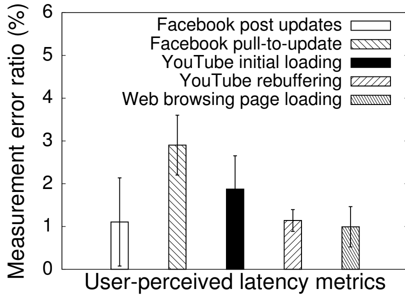

Figure 6: Error ratio of user-perceived latency measurements for each action.

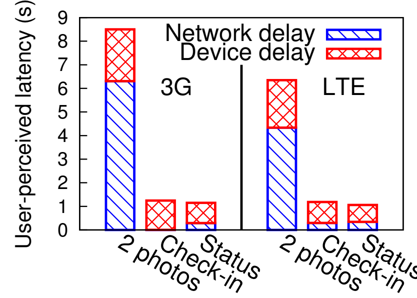

Figure 7: Device and network delay breakdown for different post uploads.

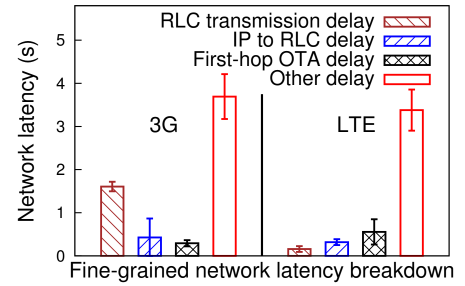

Figure 8: Fine-grained network latency breakdown for 2 photo uploading.

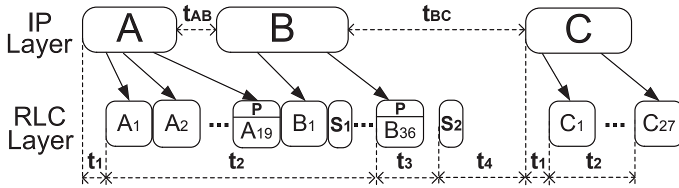

Figure 9: Example cross-layer network latency breakdown. t_{AB} and t_{BC} are the inter-IP packet times. Although $t_{AB} > 0$, A and B are transmitted back-to-back within one PDU burst in the RLC layer. PDUs with letter P are polling PDUs. S_1 and S_2 are STATUS PDUs.

each RLC burst. For the *first-hop OTA delay*, we notice that not all the first-hop OTA RTTs are on the critical path. For example, the RTT from A_{19} to S_1 is not on the critical path but the RTT from B_{36} to S_2 is, because for the former the device is busy transmitting B_1 but not explicitly waiting for S_1. Therefore, we define the *first-hop OTA delay* as the summation of the first-hop OTA RTTs which the device explicitly waits for. *Other delay* is calculated by subtracting the end-to-end time by the *IP-to-RLC delay*, the *RLC transmission delay*, and the *first-hop OTA delay*. The *IP-to-RLC delay*, the *RLC transmission delay*, and the *first-hop OTA delay* are all within one hop range from the local devices, so *other delay* consists of the latencies outside of the one-hop range, for example the latency in the switch/router, server processing delay, *etc.*

The breakdown results are shown in Fig. 8. In the figure, the RLC transmission delay in C1 3G is significantly greater than that in C1 LTE. We manually inspect the traces of posting 2 photos, and find that there are on average 270 IP packets transmitted within the QoE window, corresponding to 10553 RLC PDUs for C1 3G and 4132 RLC PDUs for C1 LTE. Such 2.55× additional number of RLC PDUs implies significant RLC PDU header processing overhead, which could be the potential reason of the higher RLC transmission delay in C1 3G.

7.3 Facebook: Background Traffic Data Consumption and Energy Analysis

To ensure that users can get interesting content from their social network at any time, Facebook app keeps communicating with the server even when it is not in the foreground. How much mobile data and energy are consumed by these background network events? How can we as users reduce cost and battery power usage while still getting timely updates? In this section, we use QoE Doctor to explore the answers.

Experiment setup. We use two devices, a Samsung Galaxy S3 device with Android 4.3 (referred as device A) and a Samsung Galaxy S4 device with Android 4.3 (referred as device B), and configure the accounts on device A and B to be mutual and exclusive friends with each other. Then, we use QoE Doctor's controller on device A to post statuses with texts of identical lengths, causing device B to receive these status updates. We change the settings of the account in device B so that it receives a notification for every news feed post from device A. Thus, device A simulates Facebook friends or public pages from which the user with device B wants to get updated with no delay. We consider these content to be *time-sensitive*. On device B, we only use the data collection functionality of QoE Doctor controller. Since our target is Facebook background traffic, we use flow analysis and only analyze TCP flows which talk to Facebook DNS domain names. For energy consumption, we use QxDM RRC state machine logs as described in §5.3.

Finding 3. Facebook's non-time-sensitive background traffic adds non-negligible overhead to users' daily mobile data and energy consumption. To see how uploading frequency impacts mobile data and energy consumption, we set the uploading frequency on device B to be every 10 minutes, 30 minutes, 1 hour, and no uploading. We run the experiment for 16 hours, and the results are shown in Fig. 10 and Fig. 11. For uploading every 10 minutes, 30 minutes and 1 hour, the results are expected: data and energy consumptions are strictly proportional to the upload frequency of device A. These content is time-sensitive for device B, so these overhead is acceptable. However, to our surprise, when device A's only friend, device B, posts nothing, device A still has around 200 Kilobytes mobile data consumption and around 300 J mobile network energy every day! We repeat the experiment and check the news feed content on device B, and find out that this traffic mainly comes from Facebook friends and page recommendations in the news feed list. Compared to the posts of device B's friends or public pages (simulated by device A), for which device B wants to get updated with no delay, we consider this traffic to be *non time sensitive*. For these content, even if it is not updated in the background, only a few seconds of waiting time is needed to update the list after the app launches. From our experiment results, if device B has time-sensitive updates every 1 hour, around half of the data and energy is spent on non-time-sensitive traffic, doubling the mobile data and energy overhead.

Finding 4. Changing one Facebook configuration can reduce mobile data and energy consumption caused by non-time-sensitive background traffic by 20%. In Facebook app's settings, an item called "refresh interval" determines how frequently the news feed list is refreshed in the background, which controls the

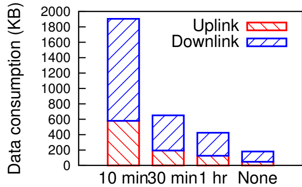

Figure 10: Per-flow mobile data consumption breakdown by post upload frequency.

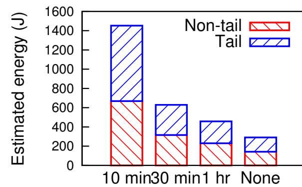

Figure 11: Estimated energy consumption breakdown by post upload frequency.

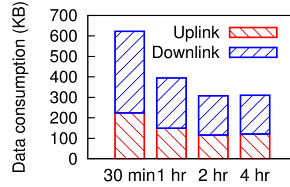

Figure 12: Per-flow mobile data consumption by refresh interval configuration.

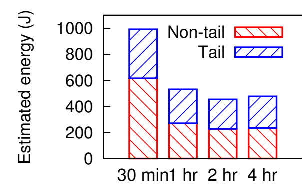

Figure 13: Estimated energy consumption by refresh interval configuration.

refresh frequency of the non-time-sensitive background traffic. In Fig. 10 and Fig. 11, the refresh interval we use is the default value, 1 hour. To explore how to configure the refresh interval, we fix the device A uploading frequency to 30 minutes to simulate the activity of its friend on device B, and change device B's refresh interval to 30 minutes, 1 hour, 2 hours and 4 hours. We collect data for 16 hours for each configuration. The results are shown in Fig. 10 and Fig. 11. As shown, the 2-hour configuration reduces mobile data consumption by 25% and mobile network energy consumption by 20% compared to the default 1-hour configuration. Another observation is that the data and energy consumptions are similar between the 2-hour and 4-hour configurations. After inspecting the traces closely, we find that the network traffic is mainly generated by status upload notifications from device A every 30 minutes, which are time-sensitive for device B. Thus, for users who think that delaying non-time-sensitive information for a while is acceptable, changing the refresh interval from 1 hour to 2 hours is likely to be a good balance between content timeliness and data, and energy consumption: it reduces mobile data and energy consumption by more than 20%, while only delaying non-time-sensitive content by 1 hour.

7.4 Facebook: Application Design Impact on News Feed Update Latency

In this section, we leverage QoE Doctor to study the impact of app design choices on user-perceived latency. We compare Facebook app version 1.8.3 and version 5.0.0.26.31. The major difference between them is that Facebook app changed the way of showing news feed list from an Android WebView to a ListView. The goal of our comparison is to identify and quantify impact of this change on QoE. We choose to relay pull-to-update action using QoE Doctor, which is an updating process related only to the news feed list.

Experiment setup. All experiments are launched on the same Samsung Galaxy S4 device with Android 4.2.2. Like §7.3, we use 2 devices, denoted by A and B. Their Facebook accounts are mutual and exclusive friends. Using QoE Doctor we have device A posting a status every 2 minutes for 6 hours. Device B passively waits for the news feed list to update by itself, which is also every 2 minutes, and measures the update latency. Facebook app version 1.8.3 does not self-update every 2 minutes, so we generate a scrolling gesture every 2 minutes to trigger the updating. We launch the experiment under both C1 LTE and WiFi. For all the experiments, we choose the same time period of a day to avoid time-of-day effect. We also choose the same place to run the experiments to ensure that the cellular and WiFi signal strengths in the comparison are the same.

Finding 5. The ListView design reduces device latency by more than 67% , network latency by 30%, and download data consumption by more than 77% compared to the WebView design. Fig. 14 shows the news feed list updating time distribution for both the WebView design and the ListView design under C1 LTE and WiFi. Under both network conditions, the user-perceived latency is greatly affected by the design – the average latency of the WebView design is more than 100% longer than that of the ListView. At the same time, the latency of the ListView has less variance. To understand the root cause, we break down the device and network delay using the same technique as in §7.2. As for posting photos in §7.2, in this experiment the network latency for news feed list updating is always inside the QoE window. As shown in Fig. 15, both network and device latency are improved by at least 67% and 30% respectively after changing the WebView design to the ListView design. We hypothesize that the reason for the device latency improvement is that WebView updating is quite complex compared to the ListView since it involves iterated content fetching and HTML parsing, which leads to a less responsive UI. For the network latency improvement, we further calculate the network uplink and downlink data consumption for the TCP flow responsible for the news feed list updating, which is shown in Fig. 16. For both C1 LTE and WiFi, the only difference is that the amount of downlink TCP data in the WebView design is more than 77% more than that in the ListView design. Thus, the network latency improvement in the ListView design is caused by much less network data to download. We think the reason is that WebView needs to display HTML content, thus compared to ListView it requires extra data to specify layout, structure, CSS, *etc.* Note that these results just suggest that using the ListView may lead to lower user-perceived latency compared to using the WebView, but the actual amount of improvement also depends on other factors such as app traffic characteristics.

7.5 YouTube: Carrier Throttling Mechanism Analysis

Most mobile users have limited monthly data plan contracted by different carriers. Normally users will be charged for over-limit data usage, while C1 uses another policy: users are still provided free data services even after exceeding the data limit [13], and the penalty is that carrier will throttle the network bandwidth on the base station. In this section, we use QoE Doctor to study how this policy may impact app QoE.

Experiment setup. To study the throttling mechanism's impact on QoE, we use QoE Doctor to play videos in YouTube app version 5.2.27 on a Samsung Galaxy S4 device with Android 4.3, with a throttled and an unthrottled SIM card for both C1 3G and C1 LTE.

Video dataset. We use "a" to "z" as keywords to search for videos in YouTube app, and choose the top 10 videos for each keyword to form our video dataset of 260 videos. This dataset is quite diverse in both the video length (1 minute to half an hour) and video popularity (several thousand views to more than 10 billion views). In this dataset the total video length is about 34.6 hours.

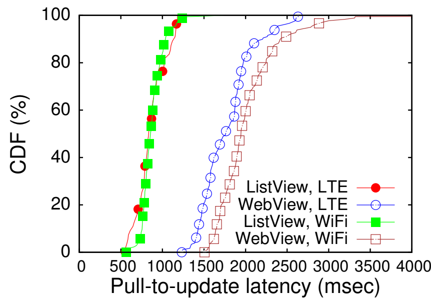

Figure 14: The news feed list updating time for WebView and ListView under C1 LTE and WiFi.

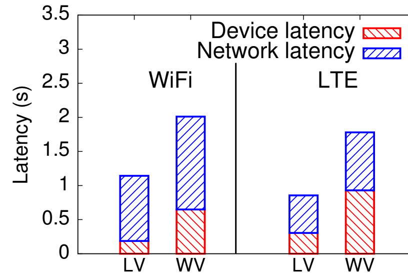

Figure 15: Breakdown of the news feed updating time for the WebView (WV) and ListView (LV) Facebook versions

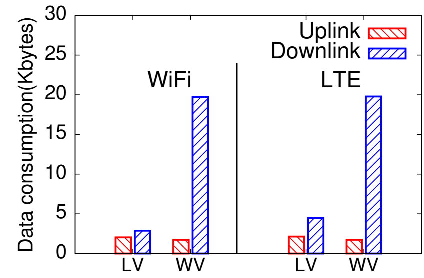

Figure 16: Network data consumption for the news feed updating for WebView (WV) and ListView (LV) Facebook versions

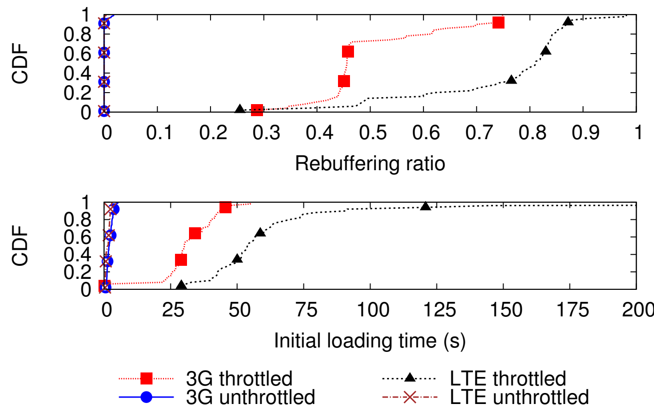

Figure 17: The video rebuffering ratio and the initial loading time distribution under throttled and unthrottled conditions for both C1 3G and C1 LTE.

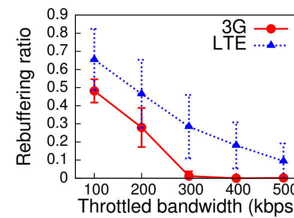

Figure 19: Rebuffering ratio for C1 LTE is consistently greater than that for C1 3G

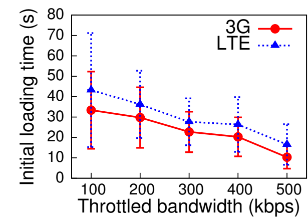

Figure 20: Initial loading time for C1 LTE is consistently greater than that for C1 3G

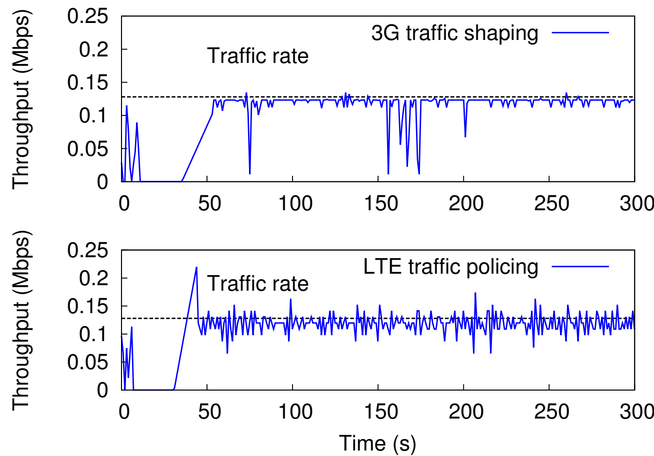

Figure 18: Throughput comparison between C1 3G traffic shaping and C1 LTE traffic policing.

Finding 6. Network bandwidth throttling causes more than 30 seconds (15×) more initial loading time and increases the rebuffering ratio from around 0% to 50% on average. We randomly play 100 videos from our video dataset under both throttled and unthrottled conditions in C1 3G and C1 LTE, and the results are shown in Fig. 17. For C1 3G, the initial loading time increases by 30 seconds after the bandwidth throttling, which is 15× more compared to the unthrottled case. For C1 LTE, it is even worse: the increased initial loading time is more than 1 minute, which is 48× more! For the rebuffering ratio, without throttling there are nearly no rebuffering events, but with throttling, more than 50% and 75% of the total playback time is spent in rebuffering for C1 3G and C1 LTE respectively, which makes the user experience highly negative.

Besides the effect of bandwidth throttling, from Fig. 17 we have another interesting observation: both the value and the variance of the initial loading time and the rebuffering ratio for throttling in C1 3G are much smaller than those for throttling in C1 LTE. Next, we use QoE Doctor to further investigate the root cause.

Finding 7. The throttling mechanism choice causes more variance in the initial loading time and the rebuffering ratio in C1 LTE. By contacting carrier C1, we find out that C1 3G and C1 LTE actually adopt different throttling mechanisms: C1 3G uses traffic shaping, and C1 LTE uses traffic policing. Both throttling mechanisms use the token bucket algorithm for rate limiting, but when the traffic rate reaches the configured maximum rate, traffic policing drops excess traffic while traffic shaping retains excess packets in a queue and then schedules the excess for later transmission [11]. Using our application layer and transport/network layer analyzer, we compare the C1 3G and C1 LTE throttling impact in Fig. 18. In the network trace, compared with C1 LTE, in C1 3G there are relatively fewer TCP retransmissions, which implies less TCP packet drops on 3G base station. Thus, the average throughput variance for C1 3G is smaller than that for C1 LTE. These are consistent with the more bursty traffic pattern expected in traffic policing [11], which is very likely the reason for more variance in the initial loading time and the rebuffering ratio in C1 LTE.

Finding 8. A simple video resolution adaptation in YouTube increases the initial loading time by 50% and doubles the rebuffering ratio in C1 LTE compared to C1 3G. Throttling mechanism choice can explain the QoE variance differences between C1 3G and C1 LTE, but it does not explain the value differences. To understand the relationship between throttling bandwidth and the video QoE with rebuffering events, we utilize a Linux traffic control tool `tc` [10]. We select a small video set (26 videos) randomly from the videos with less than 90 seconds lengths in our video dataset, and automatically play them using QoE Doctor. We repeat the bandwidth limits of 100 kbps, 200 kbps, 300 kbps, 400 kbps, and 500 kbps for both C1 3G and C1 LTE network, and the results are shown in Fig. 19 and Fig. 20. In these figures, for all bandwidth limits, the rebuffering time ratio and initial loading time for C1 LTE is consistently much higher than that of C1 3G. Using our cross-layer analyzer, we find that the root cause lies in the total downloaded video data: in C1 LTE around 96.6% ($\sim$ 40 MB) more data is downloaded than that in C1 3G. From the MediaPlayer log [5] in Android logcat [4], we find out the reason: the default video resolution is 320$\times$180 px for 3G and 640$\times$360 px for C1 LTE. Due to this default adaptation, in Fig. 17 YouTube videos on throttled C1 LTE have 50% higher initial loading time and double the rebuffering ratio than on throttled C1 3G. LTE has better maximum throughput than 3G, but it does not imply that the throughput in LTE is always better in any network conditions. We suggest that *to improve its video QoE, YouTube should adapt the video resolution based on more fine-grained real-time network performance instead of simply based on network type information*.

7.6 YouTube: Advertisement Impact on Initial Loading Time

Ads that play in the video stream before the actual video (known as "pre-roll ads") are a popular monetization approach for major video providers. Previous work [25] has studied the effectiveness of video ads as measured by their completion and abandonment rates. We instead focus on the impact of pre-roll ads on the initial loading time for YouTube.

Experiment setup. We use the same experimental setup and video data set as §7.5. We use QoE Doctor to play 100 random videos from the dataset (in total 13.5 hours) under C1 3G, C1 LTE and WiFi. In our experiments, the cellular signal strength for C1 3G and C1 LTE is around -95 to -105 dbm. Note that the ad loading latency in the results in this section might be shorter in areas with better network condition.

Finding 9. Advertisements reduce the initial loading time of the actual video, but double the total initial loading time. Fig. 21 shows the distribution of initial loading time under C1 3G, C1 LTE and WiFi. We measure four values: "video after ad" refers to the time to load the video after a pre-roll ad; "video, no ad" refers to the time to load a video in the absence of a pre-roll ad; "ad" refers to the ad loading time, and "ad + video" refers to the combined loading time for both ad and video in the presence of a pre-roll ad. Interestingly, *the ad loading time is longer than the actual video loading time for every network*. We use the cross-layer analyzer to examine the network traffic inside the QoE Window, and find that the video appears to be pre-loading while the ad is loading as well. Despite this pre-loading, *the total initial loading time is roughly doubled*. Most interestingly, on WiFi the video loading time is largely masked by the ad loading time, but on cellular networks there is still a substantial loading delay for the actual video. By examining network traffic, the root cause is that the ad loading process often has to compete with traffic to analytics

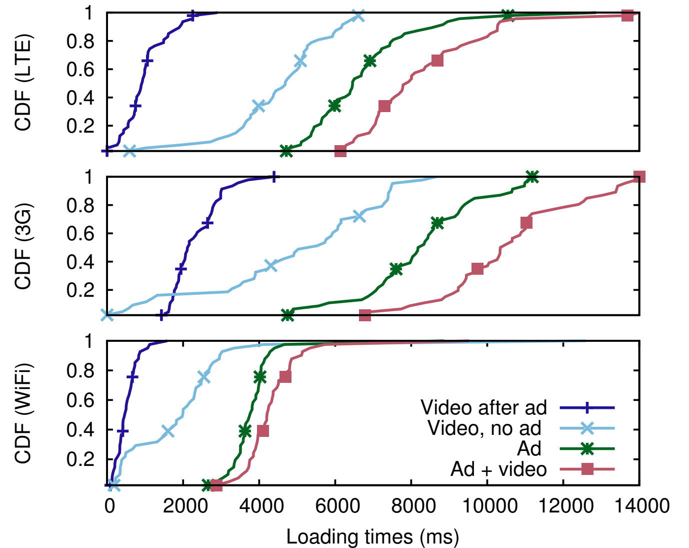

Figure 21: CDFs of video loading times for different network types, in the presence and absence of ads

services. Overall, cellular network has an additional delay of 4-6 seconds of loading time when an ad is played, on top of the 5 seconds the user must spend in watching an ad before skipping it.

7.7 Web Browsing: RRC State Machine Impact on Web Page Loading Time

As described by a previous study [33], the RRC state machine in 3G network generally consists of 3 states shown in Fig. 1. Surprisingly, during our experiment we find from QxDM that C2 has simplified its 3G RRC state machine into a 2-state model, which is shown in Fig. 22. This finding makes us curious about the possible influence this RRC state machine model change may have on mobile app QoE. In this section, we use QoE Doctor to study the impact of different RRC state machine models on the web page loading time in Android Google Chrome web browser version 18.0.1025469.

Experiment setup. In order to evaluate over real user experiences, we conducted a user study with 20 students from University of Michigan for 9 months. We installed `tcpdump` on 20 Samsung Galaxy S3 devices to collect network traces, and we anonymized device identities to protect user privacy. Using TCP and HTTP analysis, we separate the traffic generated by the Chrome browser app, and then filter out the the URLs visited by the real users along with the inter-request timings. In this filtering process, we first parse the traces to extract the links inside the HTTP requests, and then clean up the links manually to make sure that they are delivering web pages with meaningful content instead of downloading objects such as images, icons and scripts. QoE Doctor takes the filtered URL list in and replays the web browsing behavior in Google Chrome app on the C1 3G network and C2 3G network. C1 uses the 3-state model and C2 uses the simplified 2-state model in Fig. 22. As RRC state machine behavior is sensitive to the timing between network-related actions, in the experiment we not only replay the sequence of user actions, but use the URL visiting time intervals as well to ensure that the timings between user actions are also replayed.

In the experiments, the generated URL list for replay is 4 hours in length and has 597 URLs belonging to 71 different sites with a

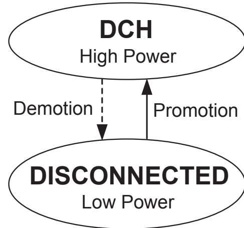

Figure 22: C2's RRC state machine has only two states, omitting FACH.

RRC transition	C1	C2
DCH	76.6 %	93.5 %
DCH→FACH→DCH	3.2 %	—
FACH→DCH	16.0 %	—
PCH→FACH→DCH	4.2 %	—
DCH→Disconnected→DCH	—	0.2 %
Disconnected→DCH	—	6.3 %

Table 4: RRC state transition distribution in the user study trace. C1 experiences a RRC state transition 23.4% of the time, while C2 only experiences 6.5%.

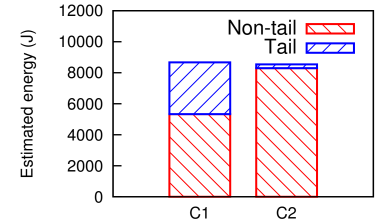

Figure 23: Tail energy contributes 35.68% of the total energy for C1, but only 2.88% for C2.

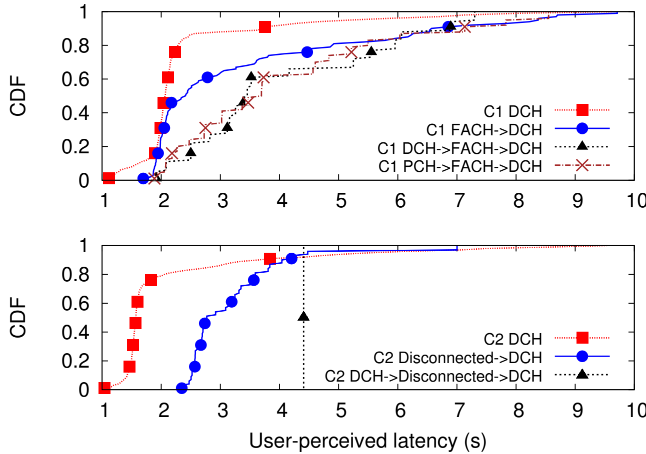

Figure 24: RRC state transitions introduce additional delay for the user-perceived web page loading time. There is only one data point for C2 DCH→DISCONNECTED→DCH.

wide range of content types, including news, movies, sports, social networking, google search results, and travel guides. Between the experiment for C1 and the experiment for C2, the average relative download data size difference is only 1.94%, which is mainly caused by changes in the HTTP metadata such as time, encoding mechanism, *etc.* instead of the web page content.

Finding 10. The 2-state RRC state machine model has a 16.9% lower chance to experience RRC state transitions, which causes 22.8% lower web page loading time than the 3-state RRC state machine model. To understand the impact of different types of RRC state machine transitions, we classify the loading time based on the RRC promotion and demotion events that happen inside the QoE Window. Fig. 24 shows the web page loading time distribution for different types of transitions for C1 and C2's 3G networks, and Table 4 shows the percentage of different types of transitions. Consistent with previous work [35, 34], the loading time in Fig. 24 for C1 and C2's 3G networks becomes longer when it involves RRC state machine transitions. The situation is the worst if the loading time involves both the state promotion and demotion – the page loading time at least doubles. In Table 4, between the 2-state model of C2 and 3-state model of C1, the web page loading in the 2-state model has a 16.9% lower chance to involve RRC state transitions. Based on QxDM logs, for C1 the DCH→FACH and FACH→PCH demotion timers are 3 seconds and 6 seconds, while for C2 the DCH→Disconnected timer is 10 seconds, which is roughly equal to the sum of the demotion timers in the 3-state model. Thus, the main cause for fewer state transitions in C2 is the simplified state machine design which removes the middle state: it stays in DCH most of the time, and once it promotes, it is very unlikely to demote. This results in 22.8% lower loading time in the 2-state model of C2 compared to the model of C1. We also calculate the energy consumption using the technique described in §5.3. As shown in Fig. 23, although the overall energy is similar, due to the lower chance of demotion, C2 has 92.5% lower tail energy compared to C1 according to the definition of tail energy in previous work [34]. For these results, we believe that they are generalizable to other apps besides web browsing apps, since the root cause lies in the RRC state machine design.

8. RELATED WORK

UI automation. UI automation tools are a common approach for dynamically analyzing applications for various purposes. **Application bug detection** is the most common application. In this category, Dynodroid [29], A3E [16], VanarSena [37], and ContextualFuzzing [27] are designed to uncover application bugs and crashes by automatically exploring all possible internal states, and exposing them to various external contexts. **Accessibility policy checking** is also a popular target. AMC [26] automatically explores app UI states to check for violations of UI requirements for vehicular apps. DECAF [28] uses UI automation to detect ad fraud. UI automation is also used for **security and privacy** purposes. For example, AppsPlayground [36] provides a framework using fuzz testing for malware analysis. Finally, PUMA [21] generalizes the common procedures for all the systems above, and provides a generic programmable framework. Unlike PUMA, QoE Doctor in this paper does not aim to expose abnormal behavior in mobile apps, but instead aims to analyze the QoE for normal mobile application usage. For example, we have a specially designed *wait* component in our UI automation (detailed in §4) for the accurate measurements of user-perceived latency, which in previous work was usually implemented using heuristic waiting timers.

QoE measurement. Previous work have measured application QoE using subjective evaluations from users such as evaluation scores. D. Joumblatt et. al. [24], Chen et. al. [20], and Ickin et. al. [23] define a target QoE metric based on user satisfactions and dissatisfactions. Schatz et. al. [42] and Balachandran et. al. [17, 18] use user engagement as the target QoE to predict. Unlike them, we focus on objective QoE metrics, which are reproducible and can be measured repeatedly and automatically. Like us, Prometheus [15], AppInsight [38], Timecard [39], and Panappticon [47] measure objective QoE metrics, but they either require access to app source code, or require instrumentation of the app logic or the underlying system, which QoE Doctor does not.

Rather than directly measuring subjective and objective QoE, QoE estimation from network traffic is also a popular approach. D. Joumblatt et. al. [24] predicts the QoE of network applications from network metrics. Schatz et. al. [42] and Balachandran et. al. [17, 18] build a predictive framework to find the relationships between measurable user engagement metrics and actionable video delivery mechanisms in the network (*e.g.,* the bit rate, initial loading time, and buffering ratio). Prometheus [15] predicts objective QoE metrics such as the buffering time with passive network measurements. Unlike them, QoE Doctor does not predict or estimate QoE metrics, but rather directly measures the ground truth values of QoE metrics.

QoE improvement. There has also been work on improving the QoE of mobile apps. Timecard [39] instruments a mobile OS to ensure that user-perceived delays can meet deadline requirements. Proteus [46] predicts future network performances over cellular networks in real time, and increases QoE for RTC applications. Sprout [45] builds a UDP-based end-to-end protocol for mobile apps such as video conferencing which requires both low latency and high throughput. Our work is complementary: QoE Doctor can be used to automatically and repeatedly collect and analyze QoE data for validating these systems, and can potentially uncover root causes of new QoE problems, shedding light on future areas for QoE improvement.

Cross-layer analysis. Cross-layer analysis has been less extensively explored. RILAnalyzer [44] uses cross-layer analysis approach to uncover how RRC states affect app performances. Compared to it, QoE Doctor supports automatically collecting objective QoE values from UI changes directly, instead of relying on user studies and studying network layer performance metrics such as TCP operations which are less directly related to user-perceived latency. ARO [35] analyzes `tcpdump` traces to uncover app performance issues. However, compared to QoE Doctor, their work mainly focuses on radio resource efficiency problems rather than app QoE.

9. CONCLUSION

In this paper, we built a tool, QoE Doctor, which automatically replays user interaction sequences of interest to measure mobile app QoE, record relevant QoE metrics, and allow the root causes of QoE problems to be analyzed across multiple layers, covering both the system and network stacks. Using this tool, we systematically study various QoE metrics for popular apps, and quantitatively evaluate the impact of various QoE-related factors on these QoE metrics. With QoE Doctor, we uncover several significant QoE problems along with the potential root causes of them for major applications and carriers.

Acknowledgments

We would like to thank Yihua Ethan Guo, Ashkan Nikravesh, the anonymous reviewers, and our shepherd, Anmol Sheth, for providing valuable suggestions and feedback on our work. We would also like to thank Warren McNeel, John Murphy, Pete Myron, Isaac Robinson, Samson Kwong, Jeffery Smith, the Device Development and QoE Lab team at T-Mobile for their assistance. This research was supported in part by the National Science Foundation under grants CNS-1059372, CNS-1039657, CNS-1345226, and CNS-0964545.

10. REFERENCES

[1] 94% Users Skip Pre-roll Ads. https://econsultancy.com/blog/63277-pre-roll-video-ads-is-it-any-wonder-why-we-hate-them#i.9gohovetnegyqcc.
[2] Android Activity Testing. http://developer.android.com/tools/testing/activity_testing.html.
[3] Android DDMS. http://developer.android.com/tools/debugging/ddms.html.
[4] Android Logcat. http://developer.android.com/tools/help/logcat.html.
[5] Android MediaPlayer. http://developer.android.com/reference/android/media/MediaPlayer.html.
[6] Global Internet Phenomena. https://www.sandvine.com/trends/global-internet-phenomena/.
[7] Monsoon Power Monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.
[8] Speed Index metric for WebPagetest. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.
[9] TCPDUMP. http://www.tcpdump.org/.
[10] Traffic Control HOWTO. http://tldp.org/HOWTO/Traffic-Control-HOWTO/.
[11] Comparing Traffic Policing and Traffic Shaping for Bandwidth Limiting. http://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-policing/19645-policevsshape.html, 2005.
[12] QxDM Professional Proven Diagnostic Tool for Evaluating Handset and Network Performance. http://www.qualcomm.com/media/documents/files/qxdm-professional-qualcomm-extensible-diagnostic-monitor.pdf, 2012.
[13] T-Mobile's next move: Shame AT&T and Verizon into ditching data overage fees. http://bgr.com/2014/04/09/t-mobile-vs-att-vs-verizon-data-overage-fees, 2014.
[14] T. 3rd Generation Partnership Project. 3GPP TS 25.322: Radio Link Control (RLC) - UMTS, 2013.
[15] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan. Prometheus: Toward Quality-of-Experience Estimation for Mobile Apps from Passive Network Measurements. In *HotMobile*, 2014.
[16] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *Proc. OOPSLA*, 2013.
[17] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. A quest for an internet video quality-of-experience metric. In *ACM HotNet*, 2012.
[18] A. Balachandran, V. Sekar, A. Akella, S. Seshan, I. Stoica, and H. Zhang. Developing a predictive model of quality of experience for internet video. In *ACM SIGCOMM*, 2013.
[19] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *USENIX ATC*, 2010.
[20] K. Chen, C. Huang, P. Huang, , and C. Lei. Quantifying Skype user satisfaction. In *Proc. ACM SIGCOMM*, 2006.
[21] S. Hao, B. Liu, S. Nathy, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps. In *Proc. ACM MobiSys*, 2014.
[22] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and

power characteristics of 4G LTE networks. In *Proc. ACM MobiSys*, 2012.
[23] S. Ickin, K. Wac, M. Fiedler, L. Janowski, J.-H. Hong, and A. K. Dey. Factors influencing quality of experience of commonly used mobile applications. *Communications Magazine, IEEE*, 50(4):48–56, 2012.
[24] D. Joumblatt, J. Chandrashekar, B. Kveton, N. Taft, and R. Teixeira. Predicting user dissatisfaction with internet application performance at end-hosts. In *INFOCOM, 2013 Proceedings IEEE*, 2013.
[25] S. S. Krishnan and R. K. Sitaraman. Understanding the effectiveness of video ads: a measurement study. In *Proc. ACM IMC*, 2013.
[26] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin. AMC: Verifying User Interface Properties for Vehicular Applications. In *Proc. ACM MobiSys*, 2013.
[27] C.-J. M. Liang, N. Lane, N. Brouwers, L. Zhang, B. Karlsson, R. Chandra, and F. Zhao. Contextual Fuzzing: Automated Mobile App Testing Under Dynamic Device and Environment Conditions. Technical report, Microsoft Research, 2013.
[28] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proc. NSDI*, 2014.
[29] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *Proc. ACM FSE*, 2012.
[30] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *EuroSys*, 2012.
[31] F. Qian, S. Sen, and O. Spatscheck. Characterizing Resource Usage for Mobile Web Browsing. In *MobiSys*, 2014.
[32] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Periodic transfers in mobile applications: network-wide origin, impact, and optimization. In *WWW*, 2012.
[33] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing radio resource allocation for 3G networks. In *Proc. ACM IMC*, 2010.
[34] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Top: Tail optimization protocol for cellular radio resource allocation. In *ICNP*, 2010.
[35] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *MobiSys*, 2011.
[36] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: automatic security analysis of smartphone applications. In *CODASPY*, 2013.
[37] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. Technical report, Microsoft Research, 2013.
[38] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proc. Operating Systems Design and Implementation*, 2012.
[39] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: controlling user-perceived delays in server-based mobile applications. In *Proc. ACM SOSP*, 2013.
[40] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Discovering Fine-grained RRC State Dynamics and Performance Impacts in Cellular Networks. In *Proc. ACM MobiCom*, 2014.
[41] S. Rosen, H. Luo, Q. A. Chen, Z. M. Mao, J. Hui, A. Drake, and K. Lau. Understanding RRC State Dynamics through Client Measurements with Mobilyzer. In *ACM MobiCom S3 Workshop*, 2014.
[42] R. Schatz, T. Hossfeld, and P. Casas. Passive YouTube QoE Monitoring for ISPs. In *Proc. IEEE IMIS*, 2012.
[43] G.-H. Tu, C. Peng, C.-Y. Li, X. Ma, H. Wang, T. Wang, and S. Lu. Accounting for roaming users on mobile data access: Issues and root causes. In *Proc. ACM MobiSys*, 2013.
[44] N. Vallina-Rodriguez, A. Auçinas, M. Almeida, Y. Grunenberger, K. Papagiannaki, and J. Crowcroft. RILAnalyzer: a comprehensive 3G monitor on your phone. In *Proc. ACM IMC*, 2013.
[45] K. Winstein, A. Sivaraman, H. Balakrishnan, et al. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proc. NSDI*, volume 13, 2013.
[46] Q. Xu, S. Mehrotra, Z. Mao, and J. Li. PROTEUS: network performance forecast for real-time, interactive mobile applications. In *Proc. ACM MobiSys*, 2013.
[47] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: event-based tracing to measure mobile application and platform performance. In *CODES+ ISSS*, 2013.
[48] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. M. Mao, and L. Yang. Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones. In *CODES+ ISSS*, 2010.