Design and Implementation of Web-based Speed Test Analysis Tool Kit

Rui Yang[1], Ricky K. P. Mok[2], Shuohan Wu[3], Xiapu Luo[3], Hongyu Zou[4], and Weichao Li[5]

[1] ETH Zürich, Switzerland
[2] CAIDA/UC San Diego, USA
[3] The Hong Kong Polytechnic University, Hong Kong
[4] UC San Diego, USA
[5] Peng Cheng Laboratory, China

Abstract. Web-based speed tests are popular among end-users for measuring their network performance. Thousands of measurement servers have been deployed in diverse geographical and network locations to serve users worldwide. However, most speed tests have opaque methodologies, which makes it difficult for researchers to interpret their highly aggregated test results, let alone leverage them for various studies.
In this paper, we propose WebTestKit, a unified and configurable framework for facilitating automatic test execution and cross-layer analysis of test results for five major web-based speed test platforms. Capturing only packet headers of traffic traces, WebTestKit performs in-depth analysis by carefully extracting HTTP and timing information from test runs. Our testbed experiments showed WebTestKit is lightweight and accurate in interpreting encrypted measurement traffic. We applied WebTestKit to compare the use of HTTP requests across speed tests and investigate the root causes for impeding the accuracy of latency measurements, which play a vital role in test server selection and throughput estimation.

1 Introduction

Internet surfers often use web-based speed tests to measure their access bandwidth, diagnose slow residential broadband connections [27], and validate the ISP-advertised speed [12,38]. A few such testing platforms make the collected data and source code publicly available (e.g., M-Lab NDT [20]), which researchers have leveraged for various studies including evaluating video streaming and cloud platform performance [8,21], measuring Internet latency [13], and inferring network congestion [35,42,1]. On the contrary, commercial speed tests (e.g., `fast.com`), serving millions of users across the world, have much more diverse server deployment than open-source ones. As of October 2021, there have been over 38 billion tests conducted by Ookla [23], a popular speed test which deploys tens of thousands of test servers worldwide. Meanwhile, video content providers including Netflix and Hulu advise users to run speed tests using their custom platforms [22,16] which host their servers in video delivery networks.

Despite their popularity and resources, unfortunately, it is still very hard for the research community to effectively utilize these proprietary tests. First, most speed tests offer only a web interface that selects a default test server for users, which makes automatic and configurable test execution burdensome. Second, despite the overhead introduced by different layers (e.g., the browser itself) and environmental dynamics (e.g., network congestion), these platforms report only few simple metrics (e.g., download/upload throughput). Without data providing the necessary context, diagnosing the root causes of performance degradation is very difficult. Finally, even though the measurement traffic is dummy data, speed tests run over HTTPS, preventing observation of HTTP transactions directly from packet captures. This opacity presents challenges for understanding speed tests since correlating the HTTP transactions with timings of the corresponding packets is essential for comprehensively analyzing speed test results (e.g., mapping TCP congestion behavior to underlying test methodologies).

To bridge this gap, we propose WebTestKit, a lightweight framework which enables automatic execution and in-depth analysis of web-based speed tests. WebTestKit has three design goals: *(i)* provides a unified and configurable interface for executing reproducible tests across multiple platforms, *(ii)* captures test data extensively from multiple layers in a lightweight manner, and *(iii)* conducts cross-layer analysis to provide a comprehensive view of the speed test results.

We implemented a prototype of WebTestKit to achieve our goals. The key idea of WebTestKit is to keep our framework lightweight and still allow an in-depth analysis of test results with high accuracy. To achieve this, we *(i)* minimized overhead by capturing only packet headers of measurement traffic and *(ii)* developed a packet matching algorithm to locate HTTP messages in encrypted traffic without decrypting them. We used a headless Chromium browser to automate the execution of five major web-based speed tests: Ookla Speedtest [24], Comcast Xfinity test [7], Netflix Fast.com [9], CloudFlare speed test [6], and speedof.me [2]. We crawled the full server lists by exploiting RESTful APIs used on Ookla and Xfinity test websites. When conducting measurements, WebTestKit allows users to select specific servers from these server lists.

Our testbed experiments showed that WebTestKit has a significantly lower impact on measurements compared to capturing full-size packets. WebTestKit is also accurate in inferring the locations of HTTP messages in packet traces, allowing us to extract many timing information from the encrypted traffic.

We demonstrated the capability and usability of WebTestKit with three use cases. We studied the behavior of different speed tests (§6.1) and found that the number and size of HTTP requests/responses were largely different between tests. Some tests sent thousands of small requests, increasing the load of the client. We discovered that preflighted requests were often unintentionally triggered, generating additional network overhead. We also used WebTestKit in the wild to run Xfinity speed test from Google Cloud (GCP) for two weeks (§6.2). We found high round-trip time (RTT) variances reported by the test due to the plausible glitches in the web interface. We compared the accuracy of two sets of JavaScript APIs that speed tests commonly used for RTT measurements. We

found that the measurements using XMLHttpRequest API suffered from at least 2.3ms error, compared to the RTTs captured by `tcpdump`.

2 Related Work

In this section, we first survey work on speed test tools and then, explore the studies on evaluating their performance and accuracy.

Speed Test Tools Most commercial web-based speed tests [24,9,2,20,7] are flooding-based which use one or multiple parallel TCP connections to saturate the access link. However, these tools could incur high costs including excessive data transfer. Probe-optimized tools like Spruce [33] and IGI/PTR [15] employ the Probe Gap Model which sends back-to-back packet pairs to estimate the available bandwidth with the packet pair dispersion. There are also some tools using the Path Rate Model (e.g., Pathload [17] and Pathchirp [28]) which sends packet trains at different sending rates to self-induce congestion at the bottleneck. Unfortunately, these tools are highly sensitive to different network dynamics (e.g., packet loss), often leading to non-negligible inaccuracies especially in high-speed networks. Recently, FastBTS [43] used a statistical sampling framework to probe elastic bandwidth for high-speed wide-area networks, with significantly reduced data usage and test duration. CLASP [21] leveraged speed tests to perform throughput measurements from the cloud. Murakami [19] supports running automated speed test measurements and collecting test results. It also provides a configurable interface for recurring jobs. Instead of building new speed test tools, WebTestKit focuses on analyzing the existing web-based speed tests that are most popular among end-users.

Speed Test Evaluation Goga and Teixeira [11] compared the accuracy of flooding-based methods [33,28] and probe-optimized tools [33,15,17] for measuring residential broadband performance from home gateways. Sundaresan *et al.* [34] conducted experiments to determine the number of parallel TCP flows required to accurately perform throughput measurements using the BISMark platform. Li *et al.* [18] evaluated three commonly used browser-based delay measurement methods, and found that the socket-based approach incurred smaller overhead than the HTTP-based one. Feamster and Livingood [10] and Bauer *et al.* [4] identified potential issues in various speed test platforms for measuring Gigabit broadband networks (e.g., the selected off-net measurement servers). Yang *et al.* [43] evaluated the accuracy of eight representative speed tests. However, different from WebTestKit, they did not perform any further analysis of test data to infer causes of inaccuracies.

3 Web-based Speed Test Platforms

Web-based speed test platforms conduct bulk data transfers over HTTP(S)/ TCP [25] to measure the bandwidth of the bottleneck link by saturating it with TCP flows. The bottleneck link is commonly the "last mile" — the access link between the client and the Internet. In this scenario, the ideal location of

the server is as close as possible to the client to minimize the latency. TCP throughput has a well-understood inverse relationship with latency [26] — the longer the latency across a path, the lower the throughput, all other factors being equal. As broadband access speeds increase, low latencies from test servers to clients ensure that measurement flows can saturate the bottleneck link [3].

Table 1: Comparison of HTTP-based speed test platforms.

Platform	# of Servers	Network(s)	Server Selection	# of TCP flows[‡]
Ookla [24]	>12,000	Various ISPs	Latency, IP Geolocation	6
Xfinity [7]	78[†]	Comcast	IP Geolocation	18
Fast.com [9]	Unknown	Netflix	Latency, IP Geolocation	11
SpeedOf.Me [2]	88	Verizon Edgecast	Anycast	1
Cloudflare [6]	Unknown	Cloudflare CDN	Anycast	1

[‡]: Speed test platforms may adapt the number of connections used depending on the type and speed of connections. We evaluated the tests in a wired Gigabit network.
[†]: These Xfinity servers were distributed in 29 locations in the United States.

Table 1 summarizes the properties of HTTP-based speed test platforms that WebTestKit supports. The scale of deployment and network coverage largely varies across platforms. Ookla speedtest has deployed the largest number (>12 000) of measurement servers around the world. These servers are hosted by ISPs, web-hosting companies, and cloud services. Other speed test platforms host servers only within their own networks or CDNs. Speed test platforms employ different methods to select test servers for users. Ookla first selects 10 servers nearest to the user based on IP geolocation and uses the one with the lowest round-trip time from the user. CDN-based speed tests share the same catchment functions as their host CDNs to divert users. All the platforms, except `speedof.me`, use HTTP/1.1 over TLS to perform throughput measurements. Three of the platforms establish multiple concurrent TCP connections to saturate the bottleneck. Although `speedof.me` adopts HTTP/2, it sequentially downloads/uploads web objects without invoking the multiplexing mechanism.

Some of these speed tests do offer a command-line interface (CLI) [31,30,29]. Though convenient for automating tests, these CLIs cannot capture multiple-layer information (e.g., browser-layer), which is essential for correctly interpreting measurements and performing further analysis.

4 Design of WebTestKit

In this section, we discuss the design objectives of WebTestKit (§4.1), then its components (§4.2) and implementation (§4.3).

4.1 Design Objectives

The main challenge of building a tool to leverage the many web-based speed tests for various research studies, lies in designing a framework that extensively

collects and provides in-depth analysis of test data, while keeping it lightweight and easy to use. Specifically, we have the three following design objectives.

A Unified and Configurable Interface Our first design objective is to provide users with a unified and configurable interface for automatically executing all web-based speed tests we support. This objective is two-fold: *(i)* a unified interface to improve usability and repeatability of these platforms which allows fair comparison among them, and *(ii)* a configurable interface to support measurement server selection which effectively exploits the resources of speed tests.

Lightweight and Extensive Data Collection Our second design objective is to enrich metrics we collect to allow correct interpretation of measurements. Due to the run-time dynamics of speed tests, collecting information to monitor the local environment, browser, and network traffic is essential for in-depth analysis of network quality and possible sources of measurements inaccuracies. Lightweight data collection also necessary mitigates potential interference with actual speed test measurements.

In-depth Cross-layer Analysis Our third design objective, closely related to the second, is to perform cross-layer analysis of collected data. We aim at providing a comprehensive and in-depth view across different layers in the system.

4.2 Components of WebTestKit

With the three design objectives in mind, we propose WebTestKit, a lightweight framework for automating speed tests with different configurations and providing in-depth analysis of test results. WebTestKit consists of three modules:

1. Measurement server exploration module discovers available measurement servers in speed test infrastructures. Of the five platforms listed in Table 1, Xfinity and Ookla speed tests support manual selection of measurement servers. However, their full lists of servers are not publicly available like M-Lab. To allow adjustably configuring the targeted servers for different needs, WebTestKit first identifies RESTful APIs that the web interface uses to query measurement servers by observing the associated HTTP transactions, and then uses these RESTful APIs to retrieve measurement server information including hostnames, IP addresses, and physical locations. As of October 2021, WebTestKit found 78 and 12 149 test servers in Xfinity and Ookla platforms, respectively.

2. Test execution module integrates a suite of tools to automate the execution of speed tests and capture data from different layers. The main challenge is to minimize the interference of data collection with throughput measurements. Our key insight is to capture only headers of measurement packets to minimize overhead. We show (§5.1) that this module is lightweight and has minimal impact on the throughput measurements.

For all speed test platforms we support, WebTestKit provides a command-line interface for programmably visiting and executing tests. To guarantee that our results are identical to a user manually running the test with a Chromium browser, we used an actual browser to render and interact with the speed test platforms. Given a speed test platform and the measurement configurations (e.g., target server), the interface will interact with the web page by clicking the 'Start'

button, selecting measurement options, detecting the completion of the test, and capturing the results displayed on the page.

In the browser, WebTestKit uses the performance trace function [37] to reveal the resource timing API information [39] and event information received by JavaScript `XMLHttpRequest` API [41]. We could have obtained such information by capturing two types of internal messages (`devtools .timeline` and `blink.user_timing`). However, a recent update in Chrome (and Chromium) [14] removed visibility of pre-flight HTTP `OPTIONS` requests from performance trace. Having only partial visibility of HTTP transactions could easily cause inaccuracies in matching the corresponding packets. Therefore, we employed Chrome's NetLog [36], which provides network layer information including TCP source ports and the data transmission progress of *all* HTTP flows at the socket level.

At the packet level, WebTestKit captures the first 100 bytes of packets which include the headers. It also collects CPU and memory usage of the end host every second during the execution of tests using `SoMeta` [32].

3. Analysis module performs analysis of data collected in *Test execution module* to generate an in-depth view of all HTTP transactions from application level to packet level. WebTestKit conducts its analysis in a three-step fashion: *(i)* identifies URLs for measurement flows, *(ii)* extracts information for HTTP transactions, and *(iii)* locates HTTP messages in encrypted packet traces.

The module first identifies URLs of RESTful APIs or web objects corresponding to download/upload tests by response/request sizes and crafts platform-specific regular expressions to match URLs. Then, it uses the NetLog trace to filter HTTP transactions associated with measurement flows.

The second step is to extract events from NetLog and performance traces to obtain timing information observed by the browser and JavaScript, respectively. Since both traces have limited documentation, to correctly interpret the traces, we used visualization tools (e.g., NetLog viewer [5] and `about:tracing` tool [37]). For each HTTP transaction in the measurement flows, WebTestKit extracts the send/arrival times of HTTP requests/responses, and the sending/receiving progress events of downloading/uploading large objects.

Finally, we developed a packet matching algorithm to locate HTTP messages in encrypted traffic. This step is the key for our analysis because lacking visibility of HTTP messages at the packet level will make an in-depth analysis almost impossible. For example, without such visibility, we cannot quantify the overhead posed by different layers from timing information of measurement traffic. Although we could export SSL keys from the browser to decrypt the traffic, this approach requires us to capture full-size packets, posing significant overhead.

To make WebTestKit lightweight but capable of performing an in-depth analysis, we only capture packet headers and then locate packets containing HTTP messages by referring to the information extracted from NetLog (e.g., sequence number, payload size). This task is challenging due to the use of HTTP persistent connections and TLS encryption. An HTTP persistent connection reuses a single TCP connection for multiple HTTP transactions. We used HTTP request/response sizes to separate consecutive HTTP transactions in the same

TCP flow. However, we could not directly apply the HTTP request/response sizes obtained from NetLog to infer the total packet payload sizes, because TLS encryption induces an overhead of 20-40 bytes to TLS records to include the TLS Record header and padding bytes. The size of overhead depends on the cipher suite negotiated in the TLS Handshake process.

Our packet-matching algorithm is designed to tackle these two challenges. The algorithm first calculates the TLS overhead by identifying the first HTTP request packet in the first measurement flow. Because all test platforms initially perform latency or download tests using HTTP GET, the HTTP request is small and should fit in one packet. Therefore, the difference between the size of the HTTP header and the TCP payload size is the TLS overhead, S_{tls}. We assume the overhead is constant across different flows in the same experiment since the same cipher suite is used.

After the algorithm identifies the TLS overhead, it locates the end of the POST requests/GET responses, by estimating the size of the messages after TLS encryption using $S_{body} + S_{tls} \times N_{prog}$, where S_{body} is the message body size of the HTTP request/response, and N_{prog} is the number of progress update events of HTTP transactions in NetLog. For HTTP POST requests, N_{prog} is equal to the number of TLS records, which allows us to accurately calculate the size of requests after TLS encryption. However, as the network socket aggregates incoming response packets spanning multiple TLS records before passing data to the browser, the number of update events is fewer than of TLS records, implying an underestimation of the post-encrypted data size. To this end, the algorithm determines to have found the end of GET responses when it observes any outgoing data packet after receiving the expected amount of data (indicating the next HTTP request) or the end of TCP connection.

4.3 Implementation

Our implementation of WebTestKit[1] consists of $\approx$*2k* lines of JavaScript code and $\approx$*10k* lines of Golang code. For test execution and data collection, it uses `puppeteer`, a node.js library, to control a headless Chromium browser to programmably execute speed tests. It uses `tcpdump` to capture measurement packets, and `SoMeta` [32] to collect CPU and memory usage information on the end host.

5 Testbed Evaluation

We set up a semi-controlled testbed to examine the resource consumption of WebTestKit in four configurations and its impact on throughput measurements (§5.1), and the accuracy of the analysis module in matching packets to HTTP transactions in measurement flows (§5.2).

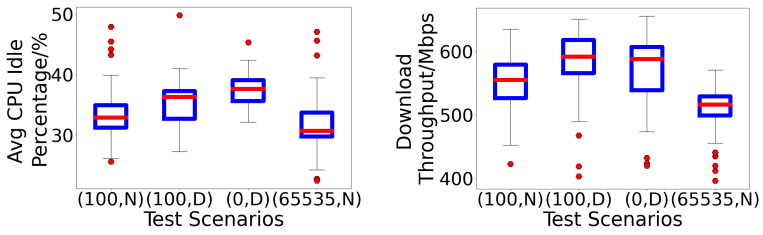
We performed two sets of experiments using *(i)* a server (Intel E3-Xeon 1270, 32GB RAM, 1Gbps Ethernet, Ubuntu 20.04), and *(ii)* a virtual machine (VM)

[1] Available at `https://github.com/CAIDA/webtestkit`

allocated with 2 vCPU and 8GB RAM set up on this server to simulate a client with low computational power. We performed speed tests with the five platforms listed in Table 1 using WebTestKit. For consistency, Ookla tests used a server hosted by 13D.net in Hong Kong, and Xfinity speed tests used servers in Seattle, WA. We configured four scenarios in WebTestKit, denoted with (*snaplen*,D/N), where *snaplen* is the snapshot length used in `tcpdump` (disabled when *snaplen* = 0), and D/N represents disabling/enabling NetLog. We ran each speed test in each scenario 50 times in the VM.

5.1 Resource Overhead of WebTestKit

We studied resource usage and its impact on measurements results under different configurations of WebTestKit. Fig. 1a and 1b show box-and-whisker plots of CPU idle rate and reported download throughput of Ookla speed tests in the VM, respectively. As expected, the control case (0,D) consumed the least CPU resource. By default, WebTestKit adopted the leftmost scenario (100,N) in the figures, which consumed 4.2% more CPU time in median than that of (0,D), but 1.5% less than (65535,N), which captures full-size packets. Comparing scenarios (100,N) and (100,D), we found that enabling NetLog slightly increased the median CPU usage by 2.4%. The download throughput followed a similar pattern to the CPU idle rate. Capturing full-size packets had the lowest median throughput (507Mbps), 41Mbps lower than the (100,N) scenario. Our results showed that compared to full-size packet capture, the default configuration of WebTestKit had a significantly lower impact on the throughput measurement.



(a) CPU idle rate. (b) Download Throughput.

Fig. 1: Box-and-whisker plots for Ookla test results under four configurations.

5.2 Accuracy of Analysis Module

We evaluated the accuracy of the analysis module in identifying HTTP transactions in encrypted packets. We used data collected in test scenario (65535,N) in our testbed experiments from the server, so that we could decrypt the traffic to reveal HTTP messages as ground truth. We compared the packet sequence numbers of HTTP request and response headers inferred by our analysis module and the actual ones observed in the decrypted traffic. We define *Request/Response*

matching accuracy ($= \frac{\#\ of\ correctly\ located\ HTTP\ requests/responses}{Total\ \#\ of\ HTTP\ requests/responses}$) to quantify the accuracy of our packet matching algorithm.

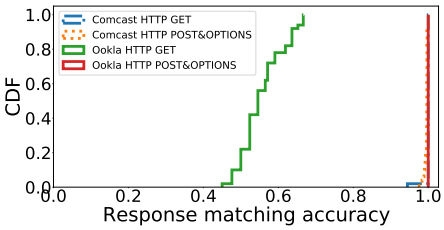


Fig. 2: Accuracy of the packet matching algorithm for HTTP responses

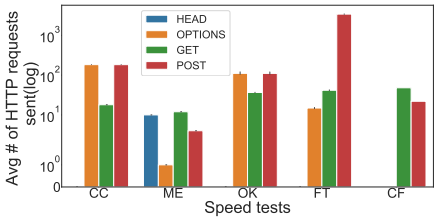


Fig. 3: The number of different HTTP requests for five speed tests

We examined our results in Ookla and Xfinity speed tests, because we could select the same servers in repeated trials. We found that WebTestKit achieved 100% matching accuracy in locating all HTTP requests. Fig. 2 shows CDFs of the response matching accuracy over 50 runs. We obtained very high accuracy here as well, except the HTTP GET responses in Ookla tests, where the median accuracy was 54.5%. The reason is that the Ookla server sent a TLS new session ticket packet to the client right after receiving the probing GET requests. We cannot easily distinguish this type of control packet from HTTP headers based on the information in the TLS record protocol header. We found that these mismatches had minimal impact on analyzing measurements. First, all these incorrectly inferred HTTP responses were for latency measurements before the actual download throughput tests, which had only a header indicating `200 OK` status. Second, the send times of the TLS packets were close to those of the send HTTP response headers. Specifically, these TLS packets were only 3 or 4 packets ahead of the actual HTTP responses.

6 Use Cases

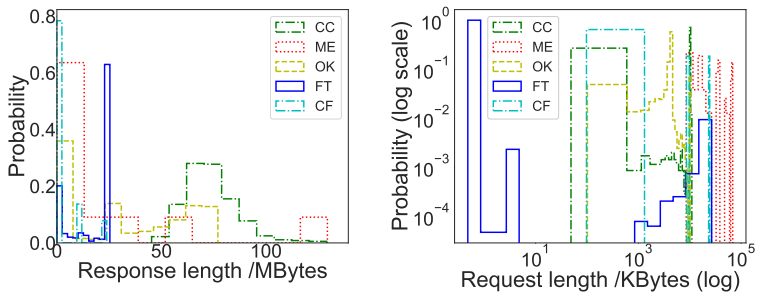
We present three use cases of WebTestKit: *(i)* characterizing the types and sizes of HTTP transactions of 5 speed test platforms (§6.1), *(ii)* diagnosing high variances in latency measurements (§6.2), *(iii)* and evaluating the inaccuracy in latency measurement using HTTP request-response time (§6.3).

6.1 Characterizing Speed Tests with HTTP Transactions.

Without access to the source code, speed tests' methodologies remain opaque. We used WebTestKit to characterize their implementations. We ran five speed tests – Xfinity (CC), SpeedOf.Me (ME), Ookla (OK), Fast.com (FT), and Cloudflare (CF) – 20 times with default settings in a workstation connected to a 1Gbps campus network. Fig. 3 shows the average number of each type of HTTP request elicited by the tests. All tests sent 10-50 HTTP GET requests to measure

downlink throughput. Meanwhile, Xfinity, Ookla, and Fast.com sent hundreds to thousands of HTTP POST requests, mainly for uplink throughput tests. Except for CloudFlare, all tests sent many HTTP OPTIONS requests. These requests were the preflighted requests to enforce the cross-origin resource sharing policy (CORS) in browsers [40]. The test servers in these four tests were in different domains from the web interface, which triggered a preflighted request for every new URIs to the test servers. Although small, these requests still consumed network bandwidth and delayed the sending of POST requests.

Fig. 4 shows two histograms of response sizes of HTTP GET and request sizes of HTTP POST for the five platforms. Over 80% of web objects downloaded by Fast.com were either 2KB (20.15%) or 24MB (63%). Xfinity's download objects sizes were between 53MB and 78MB and the download test often did not finish due to time limit expiration. SpeedOf.Me used the largest web object (128MB) among all tests. 35% of GET requests from Ookla tests had the response size of only 300B and 79% of GET requests from Cloudflare tests had the response size <1MB. All uplink tests had more consistent choices in POST request sizes than GET response sizes. We found that the POST requests sent by Fast.com were either very small (98% were 480Bytes) or very large (0.9% were >26MB). High variances between numbers and sizes of HTTP transactions for different platforms might lead to inaccuracies in various network environments [10].



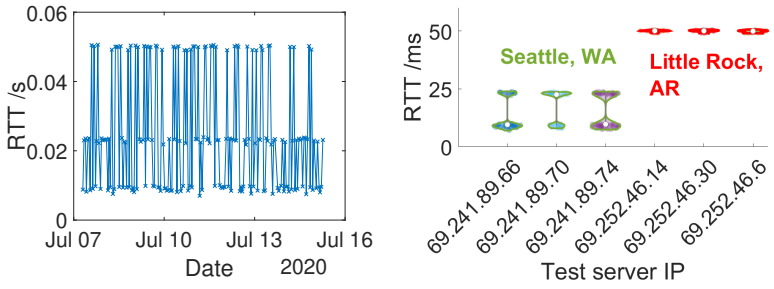
(a) HTTP GET Response (b) HTTP POST Request

Fig. 4: The distribution of HTTP object sizes for the five speed tests.

6.2 Variances in RTT Measurements

To observe RTT variances, we conducted analysis on the data collected using `CLASP` [21], which performed hourly tests from a VM in Google Cloud us-west1 to the nearest Xfinity speed test servers in Seattle, WA for two weeks (Jul 7-15, 2020). Because Xfinity by default selected servers in Little Rock, AR, we had WebTestKit instead configure the server location to Seattle, WA. Xfinity speed test sent ten persistent HTTP GET requests consecutively and reported the minimum HTTP request-response time as the RTT.

RTTs largely varied between 10-55ms throughout the time period (Fig. 5a). The screen capture WebTestKit recorded after each test confirmed that the



(a) (b)

Fig. 5: Hourly Xfinity measurements from GCP us-west1 to test servers in Seattle, WA. (a) RTTs fluctuated throughout the measurement period. (b) 25.7% of tests reported high RTTs (>50ms) still used the default servers in Little Rock, AR, ignoring our selection.

script correctly selected the server location. However, WebTestKit revealed that the measurements reporting high RTTs did not use the correct servers. Fig. 5b shows the distributions of the RTTs reported by each test according to the test servers' IP. We used the server information crawled from the test platform to locate the servers. Each location had three test servers in the same /24 subnet. The left-/right-three servers were located in Seattle/Little Rock, respectively. The RTTs to servers in Little Rock were above 50ms, while the RTTs to Seattle servers showed a bimodal distribution with peaks at 9ms and 23ms, possibly due to an asymmetric reverse path (Comcast → GCP).

The high RTT variations were partially due to Xfinity speed test failing to switch to the selected servers even when specified in the webpage. We reported this issue to Comcast for further investigation. This example illustrates WebTestKit's capability to diagnose problems with speed test implementations.

6.3 Accuracy of RTT Measurements.

Web-based speed tests often use HTTP request-response times (HRTs) to estimate RTTs, an important factor for server selection and throughput estimation. The packet matching algorithm allows WebTestKit to estimate HRTs from packet traces. Meanwhile, JavaScript-based test clients could use two APIs to measure HRTs: `XMLHttpRequest` (XHR) [41] and Resource Timing APIs (RET) [39], which both could introduce overhead to measurement results from browser rendering and system function calls. To evaluate the accuracies of these two APIs, WebTestKit records timing information from both in execution and compares them with the derived HRTs.

We analyzed the HRTs in our Xfinity speed test measurements (§6.2). We obtained three HRTs for the i^{th} HTTP transactions used for latency measurements with XHR (T_X^i), RET (T_R^i), and packet traces (T_P^i). We then computed the differences between two of the HRTs, $\Delta_{B-A}^i (= T_B^i - T_A^i . \forall A, B \subset \{X, R, P\})$. Blue, cyan, and purple bars in Fig. 6 represent the probability of different values of Δ_{X-R}, Δ_{R-P}, and Δ_{X-P}, respectively. We found XHR performed much

worse than RET. The minimum value of Δ_{R-P} and Δ_{X-P} were 0.59ms and 2.3ms, respectively, indicating the unavoidable inflation in HRTs. 66.4%/83.1% of $\Delta_{R-P}/\Delta_{X-P}$ was less than 1ms/10ms, respectively. Even though RET was much more accurate than XHR, we found 10% of Δ_{R-P} were higher than 28ms.

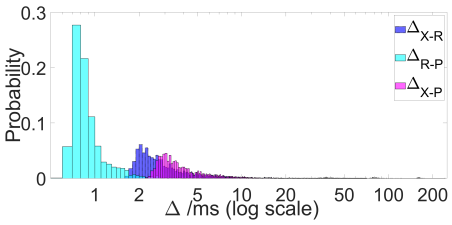


Fig. 6: Normalized histograms of HRT differences, Δ, between T_X, T_R, and T_P.

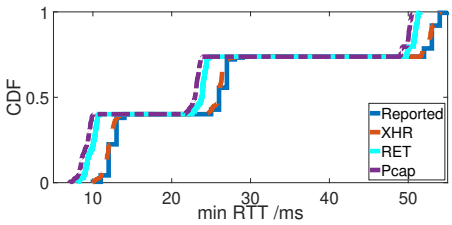


Fig. 7: CDFs of reported latency and minimum RTTs obtained from different layers.

We studied the impact of the HRT inaccuracy on the final measurement results. We selected the minimum HRTs measured with XHR, RET, and packet trace (Pcap) in each test. Fig. 7 shows the CDFs of the minimum RTTs and the reported latency. Xfinity speed test used the XHR method to measure RTTs. Therefore, the reported values were almost identical to the XHR values, except for the rounding errors. The RTTs derived using RET and packet trace were consistently lower than the XHR RTT values by around 2 and 2.9 ms, respectively, consistent with our results in Fig. 6. As the RTT between the VM and test servers in Seattle was low (The lowest RTT was 7.01ms/7.93ms/10.0ms measured by packet trace/RET/XHR), the error rate in RET/XHR was over 13%/30%, respectively. We concluded that using XHR to measure RTTs resulted in inflated values. Applying a minimum filter to measurements did not mitigate this error.

7 Conclusion

We presented WebTestKit, a unified and configurable framework for automating speed tests and performing cross-layer analysis of test results. Our evaluation showed WebTestKit was lightweight and accurate in interpreting encrypted traffic. We used WebTestKit to characterize the behavior of five major speed tests and identify a large number of preflighted requests, generating additional network overhead. We discovered high variances in RTT measurements of Xfinity speed test, caused by inconsistency between web interface and test servers.

Acknowledgment

We thank anonymous reviewers for their valuable comments. This work was supported by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164001), NSF CNS-2028506, NSF OAC-1724853, Comcast Innovation Fund, and Google Cloud credit grant.

References

1. Ookla open datasets. https://registry.opendata.aws/speedtest-global-performance/.
2. Speedof.me. https://speedof.me.
3. S. Bauer, D. Clark, and W. Lehr. Understanding broadband speed measurements. In *Proc. TPRC*, 2010.
4. S. Bauer, W. Lehr, and M. Mou. Improving the measurement and analysis of gigabit broadband networks. Technical report, Massachusetts Institute of Technology, 2016.
5. Chromium. Netlog viewer. https://netlog-viewer.appspot.com/.
6. CloudFlare. Cloudflare speed test. https://speed.cloudflare.com.
7. Comcast. Xfinity speed test. http://speedtest.xfinity.com.
8. T. V. Doan, V. Bajpai, and S. Crawford. A longitudinal view of Netflix: Content delivery over IPv6 and content cache deployments. In *Proc. IEEE INFOCOM*, 2020.
9. Fast.com. Internet speed test. https://fast.com.
10. N. Feamster and J. Livingood. Measuring internet speed. *Communications of the ACM*, 63(12):72–80, nov 2020.
11. O. Goga and R. Teixeira. Speed measurements of residential Internet access. In *Proc. PAM*, 2012.
12. T. Haselton. CNBC tech guide: How to make sure you're getting the internet speeds you pay for. https://www.cnbc.com/2018/08/17/how-to-check-internet-speed.html, 2018.
13. T. Høiland-Jørgensen, B. Ahlgren, P. Hurtig, and A. Brunstrom. Measuring latency variation in the internet. In *Proc. ACM CoNEXT*, 2016.
14. HTTP Toolkit. Chrome 79+ no longer shows preflight CORS requests. https://httptoolkit.tech/blog/chrome-79-doesnt-show-cors-preflight/.
15. N. Hu and P. Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE J.Sel. A. Commun.*, 21(6):879–894, Sept. 2006.
16. Hulu. Hulu help center: Test your internet connection. https://help.hulu.com/s/article/speed-test?language=en_US.
17. M. Jain and C. Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. *IEEE/ACM Trans. Netw.*, 11(4):537–549, Aug. 2003.
18. W. Li, R. Mok, R. Chang, and W. Fok. Appraising the delay accuracy in browser-based network measurement. In *Proc. ACM/USENIX IMC*, 2013.
19. m lab. murakami. https://www.measurementlab.net/blog/murakami/. Accessed on Junly 15, 2021.
20. M-Lab. NDT (network diagnostic tool). https://www.measurementlab.net/tests/ndt/.
21. R. K. Mok, H. Zou, R. Yang, T. Koch, E. Katz-Bassett, and K. Claffy. Measuring the network performance of google cloud platform. In *ACM IMC*, Virtual Event, 2021.
22. Netflix. Netflix help center: Internet connection speed recommendations. https://help.netflix.com/en/node/306.
23. Ookla. About ookla. http://www.speedtest.net/en/about.
24. Ookla. Speedtest. http://www.speedtest.net.

25. Ookla. How does the test itself work? How is the result calculated? https://support.speedtest.net/hc/en-us/articles/203845400-How-does-the-test-itself-work-How-is-the-result-calculated-, 2012.
26. J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. Modeling TCP Reno Performance: A Simple Model and its Empirical Validation. *IEEE/ACM Transactions on Networking*, 2000.
27. A. Philip. Slow internet? how to figure out if it's your problem or your service provider's. https://www.azcentral.com/story/news/local/arizona-investigations/2018/09/06/your-internet-slow-heres-how-figure-out-whos-fault/1058007002/.
28. V. J. Ribeiro, R. H. Riedi, R. G. Baraniuk, J. Navratil, and L. Cottrell. pathchirp: Efficient available bandwidth estimation for network paths, 2003.
29. sivel. cloudflare-cli. https://github.com/KNawm/speed-cloudflare-cli.
30. sivel. fast-cli. https://github.com/sindresorhus/fast-cli.
31. sivel. speedtest-cli. https://github.com/sivel/speedtest-cli.
32. J. Sommers, R. Durairajan, and P. Barford. Automatic metadata generation for active measurement. In *Proc. ACM IMC*, 2017.
33. J. Strauss, D. Katabi, and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *Proc. ACM IMC*, 2013.
34. S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapé. Broadband Internet performance: a view from the gateway. In *Proc. ACM SIGCOMM*, 2011.
35. S. Sundaresan, D. Lee, X. Deng, Y. Feng, and A. Dhamdhere. Challenges in inferring internet congestion using throughput measurements. In *Proc. ACM IMC*, 2017.
36. The Chromium Projects. NetLog: Chrome's network logging system. https://www.chromium.org/developers/design-documents/network-stack/netlog.
37. The Chromium Projects. The trace event profiling tool. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool.
38. The Office of the New York State Attorney General. Are you getting the internet speeds you are paying for? https://ag.ny.gov/SpeedTest.
39. W3C. Resource Timing Level 2. https://www.w3.org/TR/resource-timing-2/. Accessed on June 26, 2021.
40. M. web docs. Cross-origin resource sharing (cors). https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS\#Preflighted_requests. Accessed on Feb 23, 2019.
41. WHATWG. XMLHttpRequest Living Standard. https://xhr.spec.whatwg.org.
42. D. Xu, A. Zhou, X. Zhang, G. Wang, X. Liu, C. An, Y. Shi, L. Liu, and H. Ma. Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption. In *ACM SIGCOMM*, Virtual Event, NY, USA, 2020.
43. X. Yang, X. Wang, Z. Li, Y. Liu, F. Qian, L. Gong, R. Miao, and T. Xu. Fast and light bandwidth testing for internet users. In *USENIX NSDI*, Virtual Event, 2021.