CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier

Ranysha Ware, Adithya Abraham Philip, Nicholas Hungria,
Yash Kothari, Justine Sherry, Srinivasan Seshan
Carnegie Mellon University

Abstract

We present CCAnalyzer, a novel classifier for deployed Internet congestion control algorithms (CCAs) which is more accurate, more generalizable, and more human-interpretable than prior classifiers. CCAnalyzer requires no knowledge of the underlying CCA algorithms, and it can identify when a CCA is *novel* – *i.e.* not in the training set. Furthermore, CCAnalyzer can cluster together servers it believes use the same novel/unknown algorithm. CCAnalyzer correctly identifies all 15 of the default Internet CCAs deployed with Linux, including BBRv1, which no existing classifier can do. Finally, CCAnalyzer can classify server CCAs while being as efficient or better than prior approaches in terms of bytes transferred and runtime. We conduct a measurement study using CCAnalyzer measuring the CCA for 5000+ websites. We find widespread deployment of BBRv1 at large CDNs, and demonstrate how our clustering technique can detect deployments of new algorithms as it discovers BBRv3 although BBRv3 is not in its training set.

CCS Concepts

• **Networks** → *Transport protocols*; **Network measurement**; • **Information systems** → Clustering and classification.

Keywords

Congestion control, network measurement

ACM Reference Format:
Ranysha Ware, Adithya Abraham Philip, Nicholas Hungria,, Yash Kothari, Justine Sherry, Srinivasan Seshan. 2024. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3651890.3672255

1 Introduction

There has been a growing shift in the Internet's transport layer including an explosion of novel congestion control algorithm (CCA) proposals [10, 18, 20, 21, 53–55], many of which are already deployed or being considered and tested for deployment in the Internet by content providers. Examples include novel versions of BBR deployed by Google [17], Copa deployment by Facebook [26], and FastTCP deployment by Akamai [9, 41].[1]

[1]Although our measurement study at the conclusion of this paper suggests that Akamai has largely dropped FastTCP in favor of BBR and `www.facebook.com` uses Cubic.

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672255

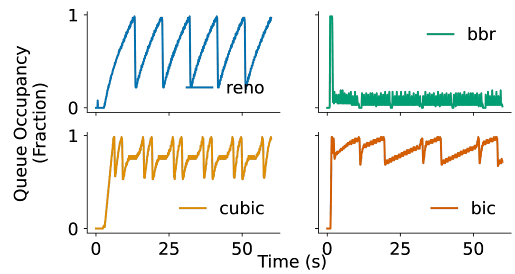


Figure 1: Time series of queue occupancy for four CCAs (from top, left to right: New Reno, BBR, Cubic, and BIC). Each CCA has a visually distinct queue occupancy behavior.

With the growing diversity in CCA proposals and potential deployments, we have an ever-growing need to understand what CCAs are currently deployed in the Internet today. Assumptions about what CCAs are widely deployed underlie decisions about how to size buffers in routers [27] (proportional to $\frac{1}{\sqrt{n}}$, *if* everyone is deploying NewReno [32]); whether or not routers need multiple queues [15] (to protect low-latency traffic from buffer filling traffic, *if* both classes of CCAs are deployed); and how to test new Internet services to ensure that they do not starve legacy traffic [30, 51, 52] (*if* Reno is no longer widely used, perhaps we do not need to test new CCAs for Reno-friendliness).

The desire to understand CCA deployment motivated the development of *CCA classifiers* starting with TBIT in 2001 [29, 41, 43, 47, 57]. Most of these tools focus on estimating the CCA's congestion window (`CWND`) by requesting a bulk data transfer from the server and then observing the transfer's reaction to dropping and delaying packet acknowledgments or to modulating the available bandwidth. Unfortunately, state-of-the-art CCA classifiers using these techniques, e.g., Gordon [41] and Inspector Gadget [29], have several limitations that prevent them from providing a truly comprehensive picture of CCA deployments. We discuss prior approaches and their limitations in detail in §2.

We seek to develop a CCA classifier with several desirable properties: **Support for all well-known CCAs**: A CCA classifier should be able to identify known CCAs with minimal errors. Supporting identification of the 15 built-in wide area CCAs in Linux[2] is especially desirable.

Efficient and nearly-passive: Network measurements should aim to be as lightweight and minimally burdensome as possible

[2]In fairness, we exclude the lp and dctcp algorithms because these algorithms require in-network support which is not available in the wide area. All other prior work also excludes these algorithms.

on non-cooperating parties. Heavyweight techniques make it difficult to perform large-scale measurement studies and can lead to measurement tools being 'blocklisted' by services.

Discover new CCAs: Open-set classification is the ability for a classifier to classify that a testing sample is not in the training set [40]. In the current period of significant experimentation in the congestion control space, a CCA classifier should be able to identify if a website is using a known or unknown CCA. Furthermore, to identify truly novel CCAs, the classifier should be able to determine which servers using unknown CCAs all appear to be using the *same* algorithm.

Interpretable results: A CCA classifier should be 'interpretable' [37]. That is, as human experts, we should be able to understand why our algorithm classifies two web servers as using the same CCA. This allows for evaluation and validation of results as well as aiding in the discovery of new CCAs.

In this paper, we present CCAnalyzer, a new CCA classifier. CCAnalyzer can correctly classify *all* built-in Linux CCAs. It is 40x faster than Gordon, and unlike Inspector Gadget, CCAnalyzer can efficiently identify if a group of servers are all using the same unknown algorithm. CCAnalyzer achieves this by taking a radically different approach to classification than prior work. Both Gordon and Inspector Gadget use decision trees hand-crafted or trained on observed CWND values or gradients; they inflate round-trip-times (RTTs) and/or introduce timeouts to precisely measure the CWND at each point in time. In contrast, CCAnalyzer starts from a simple observation: if we visually observe the occupancy of packets in a bottleneck queue over time, even a human expert can identify the connection's CCA. In Figure 1, we present the queue occupancy of the bottleneck link from real TCP connections; the familiar Reno 'sawtooth' is visible for Reno while other CCAs have their own patterns of rising and falling queue size. Because CCAnalyzer does not interfere with a connection's normal behavior (beyond introducing a low-capacity link to force a bottleneck) we describe the approach as *nearly-passive* and argue that it is minimally intrusive for operators.

Rather than trying to collect CWND traces, CCAnalyzer works by measuring a connection's queue occupancy over time and uses this time series data as input to a classic algorithm for measuring the distance between two time series called Dynamic Time Warping (DTW) [13]. DTW is used in a variety of applications requiring signal comparison, such as voice recognition and shape detection. DTW compares two signals for similarities in shape and magnitude while accounting for distortions such as stretching or noise – this latter accounting is especially valuable since we expect to see such distortions in network traces due to variances in RTT, jitter, random packet loss, *etc.* CCAnalyzer uses a 1-Nearest Neighbor(1NN) classifier with DTW as the distance measure and labeled time-series as the training set. A testing trace is given the label as the closest training sample. CCAnalyzer collects 4 queue occupancy traces for each website, and votes across the labels of those traces to give a website a final label. We describe the our methodology in more detail in §3.

We find that, in addition to being more *efficient* and *broadly applicable* than prior approaches, CCAnalyzer offers additional advantages. Collecting queue occupancy traces as well as the ability to compare these traces to one another using the 'distance' measure provided by DTW allows us to visualize and validate results. By looking at the website traces and their closest training sample we can see when and why the classification may have been incorrect for identifying possible errors. In addition, using a matrix of all the pairwise distances between a set of traces, we can cluster traces and identify the deployment of new CCAs outside of our training set. We demonstrate these additional advantages in §4 and §5.

We use CCAnalyzer to conduct a measurement study of Top 10K websites ranked by Google Chrome's UX Report (CrUX) [58] and find the following:

1. Inspector Gadget can only classify 1% of these 10K websites.
2. We find several major CDNs have deployed BBRv1 (Cloudflare, Akamai), while others still use Cubic (Fastly).
3. Clustering queue occupancy traces makes our results interpretable and straightforward to validate. It allows us to fix when a website's traces are marked as unknown when they are actually known and using a CCA in the training set.
4. CCAnalyzer was able to discover Google's deployment of BBRv3, even though we do not have a BBRv3 implementation in our testbed and did not train CCAnalyzer on BBRv3 traffic.
5. We see some deployment of other unknowns CCAs.

The rest of this paper is organized as follows. In §2 we discuss prior work in classifying CCAs. In §3, we present the CCAnalyzer methodology. In §4 we evaluate CCAnalyzer's accuracy, speed, and resource utilization. In §5 we provide a brief measurement study focusing on (a) a 2023 update on CCAs used by web servers and (b) the results of clustering unknown CCAs. In §6 we conclude and highlight future work.

2 Prior Work and Limitations

There have been several attempts at CCA classification over the past two decades beginning with TBIT [29, 41, 43, 47, 57]. Of recent classifiers, we focus on the two state-of-the-art algorithms: Gordon [41] (2019) and Inspector Gadget[29] (2020). Table 1 highlights the limitations of these classifiers.

Gordon: Gordon inspired a renaissance in CCA classification algorithms after two decades of relative dormancy. The authors insightfully noted the deployment of numerous novel algorithms (at the time, BBRv1 was beginning to 'take off' [41]) and the need to measure the changing CCA landscape due to the impact of CCAs on a wide range of Internet issues from infrastructure design to network fairness. In addition to developing the Gordon classification tool, the paper also provides the widest measurement study of CCA deployment in the post-BBR era; significantly, the authors noted the surprisingly rapid growth in the deployment of BBRv1, which 17.75% of servers they measured used at the time.

The Gordon classifier works by creating a bottleneck between the web server and the client, introducing various network events including packet losses and changes in bandwidth and delay in the hopes of exactly measuring the CWND. Generating these CWND traces comes at a high cost: Gordon requires incremental probing, RTT-by-RTT, starting and restarting connections with a web server many times—requiring up to 800MB of data transferred to successfully perform a classification. In addition, we observe in our own evaluation that more servers reject connections from the Gordon tool [8] than reported in 2019; conversations with one of the Gordon authors lead to the hypothesis that Gordon is being blocked or rate-limited due to these overheads. In 2023, Gordon authors were only able to classify 4% of Alexa Top 10K. As we will show in §4.3, CCAnalyzer trace collection transfers 85% fewer bytes, and is 40x faster than Gordon. CCAnalyzer's passivity avoids the pitfalls Gordon has with onerous active CWND estimation.

After collecting CWND traces, Gordon, uses a hard-coded decision tree to classify these traces. Because some algorithms are not distinguishable based on the parameters in this decision tree, Gordon cannot tell the difference between Compound TCP/Illinois, Vegas/Veno, and New Reno/Highspeed (HSTCP) and instead groups these into the same category although all of these algorithms are distinct.

Consequently, Gordon requires detailed knowledge about how each CCA works to support a new CCA. For example, it needed a special-cased test to support BBR. While Gordon can mark a CWND trace as 'unknown', Gordon cannot group web servers as using the same unknown CCA without running several additional handcrafted tests putting even more additional load on web servers. In addition, we will show in §4.2, although Gordon has good accuracy for supported CCAs, its lack of support for many CCAs, requirements for special tests for new CCAs, inefficiency and inability to natively discover new novel CCAs makes it challenging to use with a constantly evolving transport layer.

Inspector Gadget (IG): Published in 2020, IG's authors developed the tool to fingerprint a web server's networking stacks, including its CCA. In their results, they notably found that Cubic was the dominant CCA followed by BBR in North America, but also saw most servers from other regions were still using Reno. Similarly to Gordon, IG also tries to carefully inject network events including timeouts and changes in delay to generate CWND traces. To generate these traces, IG addresses issues with prior work's CWND estimations with some optimizations. Rather than classifying raw CWND traces, IG extracts a vector capturing the CWND as a series of offsets, using a decision tree classifier on these vectors.

IG's published code [5] includes a user-level TCP stack and modifications to a TLS library to manipulate packets in a HTTPS connection, which we find does not work in practice. We ultimately had to re-implement IG to the best of our ability. As we will show in §4.2 we obtain reasonably good accuracy with our re-implementation. We find this technique is more efficient than Gordon. However, we highlight three limitations of IG.

First, we find that IG does not make it straightforward to classify a CCA as unknown or discover new CCAs. Given the decision tree classifier, we can only mark a trace as a known label. Second, it takes considerable effort to re-implement; we find that we need to carefully account for TCP stack optimizations at the sender like F-RTO [48] that impact how a TCP flow will respond to losses that are independent of CCA behavior. These special cases are also challenges in prior work that try to collect CWND traces [57].

Lastly, when we try to use our re-implementation of IG to classify the 10K websites in our measurement study, we find that we can only successfully classify 1% of these websites because IG requires at least a 1.5MB file to classify a website, we could not find web pages large enough, and for most of the remaining that do have large enough files, we fail to generate a CWND trace. Appendix §C details these results. IG CWND estimation technique generally fails in practice when attempting to classify real websites.

Table 1: CCA classifier desirable properties

Classifier	Accurate	Unobtrusive	Open-set	Interpretable
Gordon [41]	✗	✗	✓	✗
IG [29]	✗	✗	✗	✗
CCAnalyzer	✓	✓	✓	✓

Furthermore, because of IG and Gordon's significant active manipulation of ACK timings and packet drops, their extensibility to other protocols with encryption (e.g. QUIC) or applications (e.g. video) is severely limited relative to a more passive measurement approach.

Other classifiers: The literature prior to Gordon and IG includes other influential classifiers such as TBIT [43] and CAAI [57], however, all of these approaches are superseded in both accuracy and coverage by Gordon and IG, therefore we focus our comparisons on these to prior approaches only. Other techniques that attempt to classify the CCA of a flow as it crosses a router (rather than classifying a server) such as DeePCCI [47] and DragonFly [19], are solving an orthogonal problem that is out of scope for this work.

Given the limitations of prior work our goal is the following: **We want to design a new CCA classifier with higher coverage of known CCAs, better efficiency, better passivity, and open set: able to discover new CCAs without considerable effort.** In the following sections we discuss how CCAnalyzer achieves these goals.

3 Methodology

We propose a new algorithm, CCAnalyzer, for identifying CCAs in an efficient and nearly-passive way. CCAnalyzer takes a radically different approach to prior CWND estimation techniques by relying on bottleneck queue occupancy traces. In this section, we describe how we can frame the CCA classification problem as a time series classification problem and how this enables CCAnalyzer to achieve the goals outlined in previous sections.

3.1 Observing Queue Occupancy

A key issue with prior techniques is that they require brittle and resource-intensive flow manipulation to *estimate* the CWND, which is not directly observable, and then perform classification. Our key insight is that we need not try to force network events *e.g.*timeouts to force a CCA to behave in some expected way, but rather we can observe CCAs in their natural habitat: at the bottleneck queue.

In order to observe the bottleneck queue occupancy when downloading data from a server, we insert our own switch with a deliberately slowed egress link between the server and the client using a testbed as shown in Fig. 2. Because the switch processes incoming packets at a speed much slower than upstream links, it becomes the

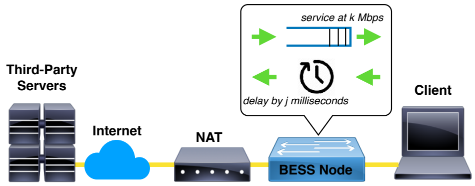


Figure 2: Testbed to issue requests to third-party servers and identify their CCAs.

connection bottleneck. The switch uses a queue of a chosen size and we configure it to record when packets are enqueued, dequeued, or dropped. We implement this switch using the BESS software switch [1], and the client issues pipelined HTTP requests to third party servers using h2load [4] to utilize the available bandwidth.

On page 1, Fig. 1 shows *real* example bottleneck queue occupancy traces collected from our testbed. A human observer can clearly see the classic 'TCP sawtooth' of Reno, x^3 curves of Cubic and even periodic bandwidth probes of BBR in these traces. CCAs will cycle through their behavior: increasing their sending rates to use the available bandwidth and react to losses (depending on their design) that occur naturally if they fill the bottleneck queue. We posit that if the patterns observed by two different flows in the bottleneck queue are equivalent, then the CCAs are equivalent.

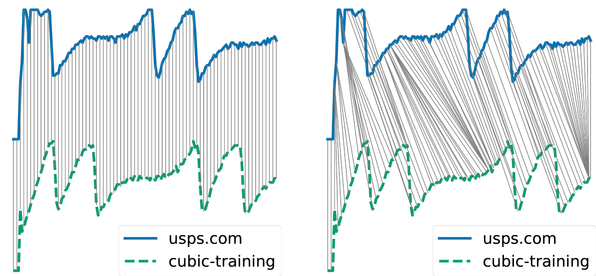
CCAnalyzer's simple inference from queue occupancy traces achieves the goals outlined in the previous section. CCAnalyzer has higher coverage of known CCAs and is more general than prior work. We can support classifying a CCA, if we can collect queue occupancy traces for that CCA. CCAs may be loss-based, may be latency sensitive, or have other characteristics and CCAnalyzer can still classify them without needing any special tests.

CCAnalyzer is nearly-passive: it does not need to force timeouts, radically modulate bandwidth, implement numerous serial connections, *etc.*. Although CCAnalyzer does normalize round-trip times and bottleneck bandwidth, to the server under test it appears as a normal TCP connection with no anomalous behaviors.

Lastly, CCAnalyzer is also open-set. Because we can compare queue occupancy traces, we can determine if a trace does not match anything in the training set. Further, we can cluster like traces and detect if multiple servers are deploying the same CCA that is not in the training set. No prior tool can *automatically* cluster servers using like, novel CCAs and we believe that this trait of CCAnalyzer is crucial to measuring and modeling a continuously-evolving Internet. While some prior work also creates a local bottleneck (*e.g.* Gordon [41]), or may try to *estimate* queue occupancy for a particular flow crossing a router (*e.g.* DragonFly [19]), our work is the first to *directly measure* bottleneck queue occupancy by creating a local bottleneck and recording every time a packet is enqueued, dequeued, and dropped from that bottleneck queue to use this trace to classify CCAs.

3.2 A Time Series Classification Approach

CCAnalyzer compares two queue occupancy traces to each other using a well-known algorithm called Dynamic Time Warping (DTW)[13], which takes in two time series traces and returns a 'distance' measurement quantifying how similar the two traces are. DTW is traditionally used in pattern matching tasks like automatic speech recognition and speaker identification; just as a speaker will have a signature pitch and cadence, congestion control algorithms each have a unique typical queue occupancy and rate of change. These types of problems are known as 'time series classification' problems, and despite 40 years of research since the invention of DTW, it remains a widely used general-purpose algorithm for this class of challenges [11].



(a) Euclidean distance = 2.47 **(b) DTW distance = 0.76**

Figure 3: Queue occupancy distance calculation for a sample from usps.com to a Cubic training sample. DTW allows a flexible one-to-many mapping between similar points, while euclidean is a one-to-one mapping to points at the exact same time.

To understand DTW, we first consider a naïve approach to compare two traces using Euclidean distance (ED). Consider two queue occupancy traces, $X = (x_1 ... x_n)$ and $Y = (y_1 ... y_n)$, where x_i is the queue occupancy at time i in trace X and where X and Y are n time steps long. We can compute ED between these two traces by computing the sum of the squared difference between each element x_i and y_i.

Fig. 3 shows why this one-to-one mapping approach fails for most network traces. In Fig. 3a, we compute the ED between a trace collected from `usps.com` to a Cubic training sample, while in Fig. 3b we take the same traces and compute the DTW distance. Traces can dilate and contract relative to time on the real Internet. For example: a host may stall during the trace, sending a packet a few ms later than expected; an in-network queue may fill up with background traffic, temporarily increasing the RTT; a long-running flow in the background may end, suddenly reducing the RTT. These effects can cause two traces from the same CCA to appear stretched and squeezed relative to one another.

DTW accounts for this stretching and squeezing by allowing a one-to-many mapping: a given index from each trace can map to one or more indices in the other trace. DTW finds the optimal point-to-point mapping between the two traces to minimize the sum of the distances between all their points with some constraints. Fig. 3b shows how this results in DTW measuring a smaller distance than ED for same-CCA traces. We describe the formal definition of DTW in Appendix §B. There are many more well-studied aspects and applications of DTW [11, 13, 33, 35, 44, 46] but we do not require their discussion here to understand CCAnalyzer.

CCAnalyzer uses a one-nearest-neighbor classifier with DTW as the distance measure (1NN-DTW), a commonly used time series classification methodology [11]. Given a website to classify, CCAnalyzer computes the DTW distances between the queue occupancy traces of all training samples and the queue occupancy trace of the website. The website is given the label of the closest training sample.

Given this approach, DTW allows us to classify if a time series matches one within the training set, but how will we determine if a CCA is not in the training set and should be classified as unknown? We explore using a well-known extension to our 1NN classifier called TNN where T is a distance threshold [40]. If the DTW distance between a website trace and it's closest label is higher than T, then the trace is marked as unknown.

3.3 Parameter Tuning

CCAnalyzer observes a TCP connection's natural behavior as its `CWND` rises and falls, probing for bandwidth. However, classifying CCAs based on this natural behavior requires that we observe TCP connections in sufficient conditions that they *act distinguishably* from one another. To be specific:

We must choose bottleneck bandwidth, RTT, and queue size such that same-family CCAs exhibit different behavior (§3.3.2): This is most important for Reno-family CCAs (Westwood, Highspeed, YeAH, etc.) which are all variants of each other. Many are designed to simply 'act like Reno' in low BDP environments and only exhibit their unique growth and backoff behaviors at higher BDP environments.

We must observe connections for long enough that each CCA goes through several 'cycles' of operation (§3.3.3): DTW matches similar traces to each other, but minor perturbations in the network environment (arrival/departure of background flows, external packet loss) can make traces appear dissimilarly. Having multiple iterations of the CCA's characteristic behavior allows DTW to self-correct for brief abberations as the characteristic connection behavior re-emerges after a few RTTs.

We need to identify when a trace is too far from its nearest neighbors in the training set (§3.3.4): We would expect servers using novel CCAs to produce a DTW distance which is 'far' from any training sample: but how far is far enough to declare that a server is indeed using a new algorithm?

Note that the above issues all somewhat depend on the set of CCAs that the system is meant to classify. We take an empirical approach to setting appropriate parameters to correctly distinguish CCAs which we describe in the following sections. However, it is not unlikely that if the CCA landscape were to evolve dramatically with the deployment of many new CCAs and the phasing out of many old ones, that we would need to re-tune these parameters for CCAnalyzer to remain effective in the future.

3.3.1 Experimental Setup The CCAnalyzer testbed is installed on Cloudlab servers in Wisconsin, USA [22] (see Fig 2). To generate ground truth data for evaluation, we collect traces to servers installed on Amazon Web Services (AWS) datacenters in Virginia and Microsoft Azure's 'East' US datacenter. We use the AWS-Virginia dataset as our *training* data for CCAnalyzer and our Azure-East datasets for *testing*. When measured using `iperf`[6], the total available bandwidth between the CCAnalyzer testbed client and the AWS machines is 500Mbps and between the testbed client and Azure machines is 920Mbps.

Each server is configured as follows:

- Training Set (**AWS-Virginia**): Ubuntu 22.04.2, Linux kernel version 5.19. RTT to testbed 22ms. 3 samples per CCA.
- Testing Set (**Azure-East**): Ubuntu 20.04.6, Linux kernel version 5.15. sRTT to testbed 24ms. 5 samples per CCA.

Training and Testing for CCAnalyzer: Using AWS-Virginia we generate training samples for 15 CCAs available in Linux.[3] We run `iperf` flows between a transmitting host located in AWS Virginia and a receiving host in our testbed for 120s (as we will discuss in §4.3 we need not use all 120s for accurate training and only need 20s). To generate testing data, we set up an Apache web server on Azure-East with a 100MB file. We use `wget` to download the file to the receiving host in our testbed for 60s.

3.3.2 Network Configuration Many CCAs, especially Reno-family CCAs, are designed to behave similarly in low-BDP environments. Therefore, we need to identify network settings in which these CCAs exhibit their distinguishing behavior. Our testbed enables us to emulate different network conditions by varying the bottleneck bandwidth, round-trip time, and the bottleneck queue size. Our main goal is to find a minimal set of network settings that we can confidently use to classify all 15 CCAs in Linux and identify unknown CCAs. We need to capture just enough cycles of CCA probing behavior that makes these algorithms distinguishable.

Bandwidth setting: We choose to use small bandwidth ranges because we want to ensure that our queue is the bottleneck for the connection; if queueing were to build up elsewhere in the network we would not observe useful behavior in the queue occupancy traces. We test setting the bandwidth to 5Mbps, 10Mbps, and 15Mbps.

RTT setting: We enforce an RTT in our testbed by adding additional delay to packets sent to the web server. Therefore, the RTT we choose for our network settings cannot be so small that the majority of websites will be too far away. In addition, if we set the RTT to be too large, then it can take the CCA a long time to fill the queue resulting in traces without enough cycles of CCA probing behavior to distinguish different algorithms. Fig. 4 shows the distribution from the 10K websites we will attempt to classify in §5. We test setting the RTT to 85ms, 130ms, and 275ms.

Queue size: Given the bandwidth and RTT of a setting, we need to chose a queue size that captures the right number of cycles of CCA probing to highlight distinguishable behavior. We find a queue size of 1BDP works well.[4] Fig. 5 and Fig. 6 show how queue occupancy traces change depending on the queue size for a 5Mbps and 275ms RTT network setting (128 packets is ~1 BPD in this setting). With queue sizes too large, queue occupancy traces degrade. In case of Cubic, it takes too long to fill the queue so the trace does not have enough cycles of Cubic probing behavior. In the case of BBR, it uses very little of the queue when the queue it is too large.

We run 1NN-DTW on our test dataset from Azure-East, classifying each 60s trace as its 'nearest' training trace. Fig. 7 shows the

[3]Note: We only include BBRv1 in our testing and training sets. In §5.2 we will show we can also classify websites using BBRv3.

[4]BESS requires the queue size to be a power of 2 so the actual queue size is set to be a power of 2 closest to 1BDP.

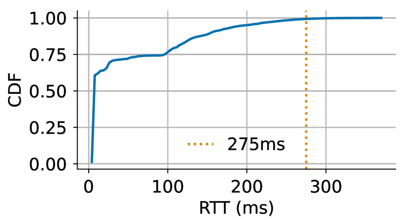


Figure 4: Distribution of ping time to 10K websites. Most websites are within a distance of 275ms.

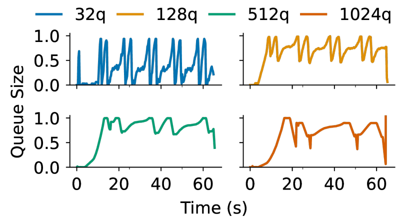


Figure 5: 5bw-275rtt: Example Cubic queue occupancy traces from Azure-East for varying queue sizes (pkts).

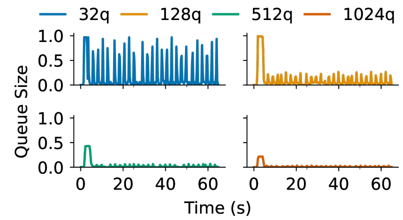


Figure 6: 5bw-275rtt: Example BBR queue occupancy traces from Azure-East for varying queue sizes (pkts).

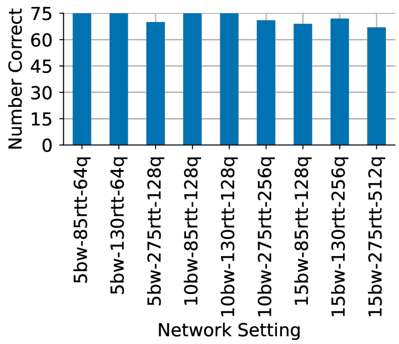


Figure 7: Accuracy mapping each testing sample to closest training sample per network setting.

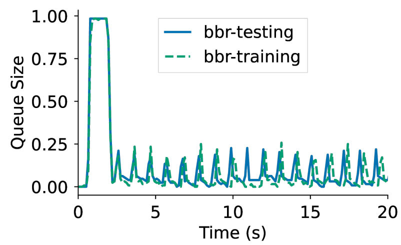


Figure 8: 10bw-130rtt setting: A BBR testing trace correctly labelled.

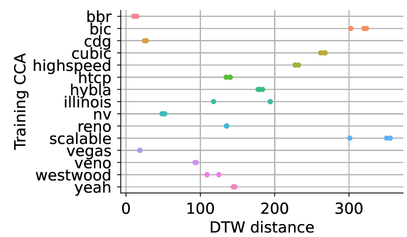


Figure 9: 10bw-130rtt: BBR trace correctly labelled. It is close to other low-latency CCAs, Vegas and CDG.

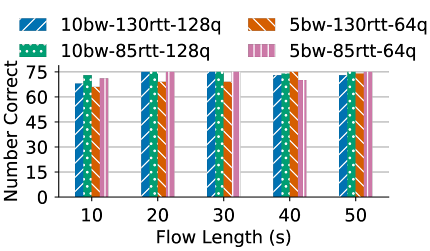


Figure 10: Results from classification for truncated traces in accurate settings. Near perfect accuracy is reached with as little as 20s flows.

accuracy of CCAnalyzer for 9 network settings for each testing set. Some settings work slightly better than others but overall accuracy across these settings is 96% (649 correct out of 675 samples). Misclassifications include Illinois samples misclassified as Westwood, both Reno variants. Similarly, BBR samples get misclassified as Vegas, both low-latency CCAs. The most accurate 4 settings are when the bandwidth is 5mbps or 10mbps, and when the RTT is 85ms or 130ms. Our ultimate design relies on voting across multiple settings in order to 'boost' our accuracy to 100%, but we want each voter to be as confident as possible: hence we restrict our measurements in CCAnalyzer to the four most accurate settings. In addition, we want to minimize the load on the web servers by finding a small number of network settings that can produce distinct traces.

Fig. 8 shows an example of why 1NN-DTW works well, with a BBR testing sample and its closest training sample which are nearly identical. More illuminating is how closely the testing sample relates to the *incorrect* CCAs. Fig. 9 shows all the distances between a BBR sample and all the training samples in the 10bw-130rtt setting. All the BBR training samples are closest to this testing sample, but other similar CCAs that are also not loss-based, such as CDG and Vegas, are the next closest. These algorithms all have relatively low magnitude in their queue occupancy compared to, *e.g.*, Reno and Cubic variants. This also highlights the interpretability of our results as the traces are visually distinct, with clear similarities between testing samples and their closest training sample and the DTW distance *quantifies* the similarity. Furthermore, unlike both Gordon and IG, CCAnalyzer *does not need a special test* to classify BBR or other algorithms that are not loss-based.

Given the accuracy we have for these 4 settings, we complete the rest of our analysis and measurement study using these settings. These work well and achieve our goals but these are not the *only settings* that will have high accuracy. There are many settings that could accurately distinguish CCAs using 1NN-DTW. We discuss further network setting options and their accuracy in §5.3.

3.3.3 Trace Length/Duration One of our key goals with CCAnalyzer is to reduce the overhead of probing relative to prior approaches. At the same time, we need to observe CCAs over a sufficient period of time such that they iterate through multiple 'cycles' of their bandwidth probing mechanisms. Consequently, we

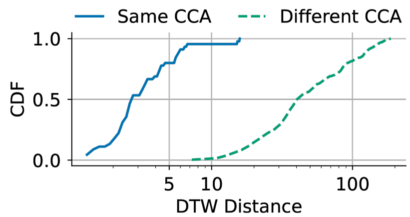


Figure 11: Distribution of distances between training samples for 5bw-85rtt-64q accurate settings. The seperation between the same CCA distribution and different CCA distribution suggests we can set a distance threshold to mark CCAs as unknown.

aim to identify the minimum duration we should measure a network trace while still ensuring strong accuracy. Fig 10 shows the accuracy from classifying flows individually (without voting) with durations ranging from 10 to 50s. We see a modest dip in accuracy when we drop as low as 10s. However, for traces from 20s-50s, we see relatively indistinguishable accuracy. Hence, we can use traces as short as 20s with minimal impact to classification accuracy and hence use this duration as our minimum trace length.

3.3.4 Classifying Unknowns Our final parameter tuning step enables us to identify *unknown* or *novel* CCAs. This is referred to as solving an 'open-set' classification problem (a problem in which some of the data to be classified may not match any of the labels in the training set) rather than a 'closed-set' problem. In prior work, only Gordon [41] provides an open-set algorithm – all other algorithms in the literature, including Inspector Gadget, are closed-set, meaning that they will always erroneously identify novel CCAs as some other existing algorithm in the training set.

CCAnalyzer's mechanism for identifying novel CCAs requires identifying some DTW distance threshold T such that if the nearest training sample to a trace is more than T distance away according to DTW, we should mark it as unknown. The algorithm for classifying with such a threshold, called TNN [40], is otherwise identical to the 1NN algorithm we described previously. Figure 11 provides intuition as to why such a threshold is useful. Here, we plot a CDF of all DTW distances between pairs of traces in our training data in which the pairs use the *same* CCA or in which they represent *different* CCAs. The distribution of distances between samples with the same CCA is tight – between roughly 1 and 15 – where pairs of different CCAs generally have a much higher DTW distance between them. The key is to choose the threshold T smartly: if we set T too high, we will mark true unknowns with a known CCA (*a false known*) and if we set T too low we will mark things that should have been labeled as a known CCA as unknown (*a false unknown*). Between the two classes of errors, we slightly prefer false unknowns because we believe that the vast majority of servers on the Internet do indeed use well-known CCAs. Consequently, we choose a low T that will have some false unknowns. In §5 we explore how we can further reduce false unknowns through clustering.

Our challenge in setting T is that we lack a way to rigorously evaluate our choice of T, since we lack ground-truth knowledge about the deployment of novel CCAs on the Internet, or even at what frequency novel CCAs are used. We can, however, emulate

Table 2: Distance thresholds per setting.

Setting	Quantile	Distance Threshold
10bw-130rtt-128q	0.90	4.41
10bw-85rtt-128q	0.94	6.45
5bw-130rtt-64q	0.90	9.73
5bw-85rtt-64q	0.95	6.68

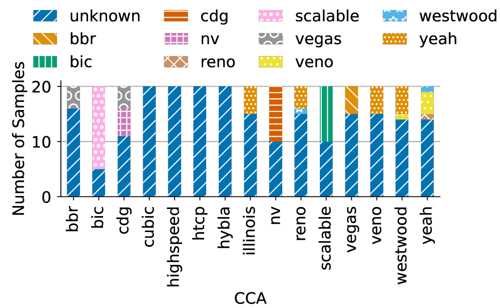


Figure 12: False positives when removing the training samples with the correct label from the testing set and seeing if we can correctly classify as unknown using a distance thresholds in Table 2 per CCA. After voting only CDG, BIC, and Scalable are misclassified as known labels.

the deployment of novel CCAs to guide our search for a good value of T.

We use our *existing* training data (AWS-Virgina) and run classification on a *new* testing set (we use a server hosted in the AWS-Ohio region) to simulate unknowns. To classify a testing sample, we remove that testing sample's CCA from the training set. For example, when we want to classify a Reno testing sample, we remove all Reno training samples from the training set, and see if the Reno testing sample will be correctly classified as unknown, or if it will be erroneously given a known label. We repeat this process for all 15 CCAs, and vary T to balance false knowns and false unknowns. Table 2 shows the results of these experiments with our choice of T for each setting. For example, in the 5bw-85rtt setting, we choose the value that is the 95th percentile of the "Same CCA" distribution in Fig. 11.

To evaluate how well these values of T work, we repeat this process with the Azure-East testing set. Figure 12 shows how each CCA is classified when we remove that CCA's training samples; ideally the CCA should be classified as unknown. Once we apply our voting scheme across all four settings (voting description in §3.4), only CDG, BIC, and Scalable are misclassified with known labels – and are mislabeled with similar CCAs (CDG is mapped to another low-latency CCA; BIC and Scalable are mapped to each other).

Now that we have a mechanism to classify unknowns, a new question arises: how do we tell which services are all using the *same* unknown? The short answer is that we can cluster unknown traces using pairwise DTW distance measures – groups of traces with small distances between them are likely to represent the same novel CCA. We return to this clustering procedure in §5.1.

3.4 CCAnalyzer End-to-End

In order to classify servers, CCAnalyzer is configured with a ground truth set of labeled queue occupancy traces for 15 CCAs for 4 network settings. Using TNN-DTW and the testbed in Fig. 2, CCAnalyzer does the following to classify a server:

1. Collect a queue occupancy trace for 20s (§3.3.3) for 4 network settings where the bandwidth is 5 or 10mbps and RTT is 85 or 130ms (§3.3.2).
2. Compute the DTW distance between each queue occupancy trace and all the training traces in the same network setting.
3. Each queue occupancy trace is given the label of the CCA that has the closest DTW distance.
4. If the distance is bigger than a distance threshold shown in Table 2 (§3.3.4) the trace is marked as unknown.
5. To assign the final label, for a website there is a vote between the 4 traces for the website. The final label for the website is the majority label across the 4 traces. If there is a tie between a known label and marking it as unknown, the CCA is marked as the known label. Lastly, if there is a tie between multiple CCAs, the final label is from the trace with the minimal distance to its closes training sample.

Finally, we use Agglomerative Clustering [42] to group unknown traces based on their DTW distances to each other. We use the distance threshold and manual inspection of these clusters to detect and identify proprietary, new, or unknown algorithms. CCAnalyzer is the only classifier which clusters unknown CCAs in any automated fashion. We explore the accuracy and efficiency of this approach in the next section.

4 Evaluation

In §4.2 and §4.3, we measure the accuracy and efficiency of CCAnalyzer and compare its performance with Gordon and IG. We were unable to obtain an executable version from the authors of IG, and ultimately had to re-implement it using the same techniques described in the paper (we describe them in §2) to the best of our ability. We find that CCAnalyzer is able to achieve 100% accuracy using its voting scheme for *all* 15 built-in CCA algorithms in Linux. During trace collection, on average, CCAnalyzer transmits 85% fewer bytes of the data that Gordon needs to classify a website, and completes 40x faster in terms of wall-clock time. CCAnalyzer achieves the same accuracy and coverage as IG and better efficiency than Gordon (§4.3), with the flexibility of open-set classification (§5.2) and more interpretable results (§4.2).

4.1 Experimental Setup

The CCAnalyzer testbed is installed on Cloudlab at Wisconsin, USA [22] (see Fig 2). To generate ground truth data for evaluation, we collect traces to servers installed on Amazon Web Services (AWS) datacenters in Virginia as well as to Microsoft Azure's 'East' US datacenter. We use the AWS-Virginia dataset as our *training* data for CCAnalyzer and our Azure-East datasets for *testing*. When measured using iPerf, the total available bandwidth between the CCAnalyzer testbed client and the AWS machines is 500Mbps and between the testbed client and Azure machines is 920Mbps.

Each server is configured as follows:

- Training Set (**AWS-Virginia**): Ubuntu 22.04.2, Linux kernel version 5.19. 3 samples per CCA. RTT to testbed 22ms
- Testing Set 2 (**Azure-East**): Ubuntu 20.04.6, Linux kernel version 5.15. 5 samples per CCA. RTT to testbed 24ms.

4.2 Accuracy

Experimental setup: We evaluate Gordon and IG using the same Azure-East web server we use to evaluate CCAnalyzer (§4.1). We point the Gordon client and IG client, installed on a server in the CloudLab Utah testbed, to download the same 100MB file from the Apache web server. We classify each CCA 5 times for Gordon and CCAnalyzer using their hand-crafted decision tree (as is done in the Gordon paper). We classify each CCA 20 times for IG (as is done in the IG paper) using traces collected from the same AWS-Virginia web server as the training set.

Both Gordon and CCAnalyzer use a voting scheme to determine their final result. In the case of CCAnalyzer, we generate measurements in four bandwidth/RTT/queue-size settings, measure DTW distances to our training data for each sample, and then vote across these four settings (§3.4). In the case of Gordon, they run 15 trials and take a vote across these 15 trials. To repeat classifying each CCA 5 times, CCAnalyzer classifies 20 queue occupancy samples per CCA and Gordon classifies 75 `CWND` trace samples per CCA.

In Fig. 15 we show the number of correct classifications for IG, Gordon and CCAnalyzer. For both Gordon and CCAnalyzer we report the results after applying their voting schemes. CCAnalyzer achieves 100% accuracy across all CCAs. The results for Gordon are more complex: CDG, Hybla, and New Vegas (nv) are not supported by Gordon and so we mark these as unsupported. Further, the published code does not support Westwood so we also mark that as unsupported. For the algorithms that Gordon does support, it misclassified all Highspeed samples, and is mostly accurate for the other CCAs

We illustrate the accuracy of these individual votes in Fig. 13 for Gordon and in Fig. 14 for CCAnalyzer. Stacked bars show how many 'votes' went to each CCA. For CCAnalyzer, its individual votes are accurate with the exception of marking known CCAs as unknown (we do favor false unknowns vs. false knowns §3.3.4) and mislabelled Illinois samples as YeAH (both are variants of Reno). For Gordon, the errors are more varied, with several loss-based protocols (BIC, Highspeed, and Illinois), identified unexpectedly as BBR. While the results in the Gordon evaluation include correct classifications for Westwood, the publicly released code [3] for Gordon does not classify traces as Westwood, and therefore does not support this algorithm. Notably, Gordon does correctly classify 3 algorithms it does not support (CDG, Hybla, and NV) as unknown, demonstrating its ability to classify some CCAs not in its known set as unknown.

To validate that our IG implementation is faithful, we attempt to replicate the results in [29] by using Azure servers to generate both testing and training samples. Under this setting, IG achieves 100% accuracy, likely due to over-fitting. IG's overall accuracy dips to 73% if we include all 15 CCAs in the training and testing set and use AWS training samples to classify Azure testing samples (like we do to evaluate CCAnalyzer); these are the results shown in Fig. 15. When restricting this set to just the 12 CCAs that IG classifies in their paper, the accuracy is 74%.

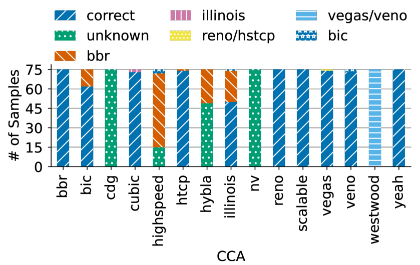


Figure 13: Gordon: Individual votes for each CCA trace. Note that CDG, NV, and Hybla are correctly marked as unknown.

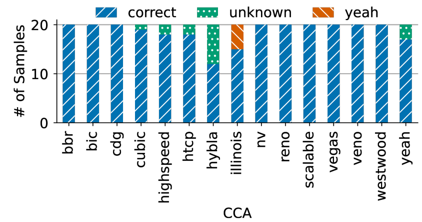


Figure 14: CCAnalyzer: Individual votes for each CCA trace.

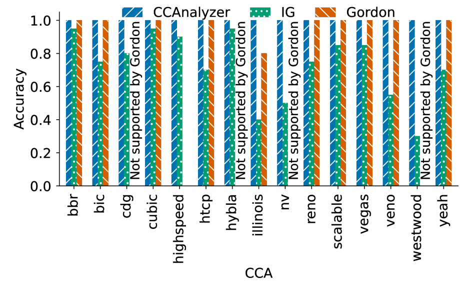


Figure 15: Comparison between CCAnalyzer, IG and Gordon classifying the same servers.

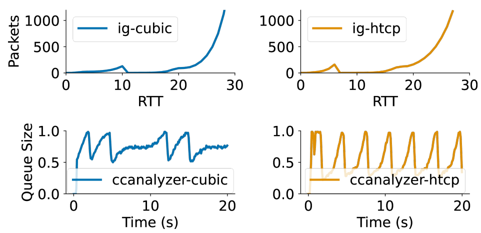
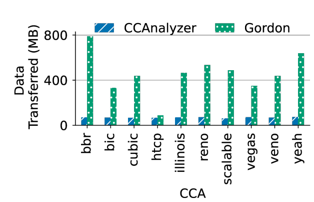
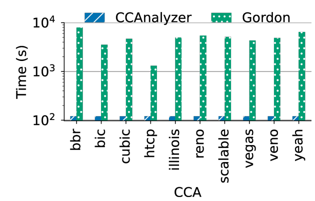


Figure 16: Traces from CCAnalyzer and IG.



(a) Data transfer comparison



(b) Time comparison

Figure 17: Efficiency comparison between CCAnalyzer and Gordon classifying the same web server. Gordon's CWND estimator depends on the CCA so both bytes transferred and time is CCA dependent.

Interpretability: There are many competing definitions for what makes a classifier "interpretable" [37]. Human experts want to be able to understand our classifier's output: why are these two traces labeled as the same CCA? Is this classification likely correct? This is one of the key advantages of CCAnalyzer over the prior work: capturing the inherent cyclical nature of CCAs, makes them more distinguishable. So much so, that not only can a classifier find these distinguishable patterns, but so too can a human observer. In Fig. 16, we compare CWND traces from IG to queue occupancy traces from CCAnalyzer. While the top graphs show traces for IG for Cubic and HTCP are nearly identical, the traces for the same CCAs from CCAnalyzer are easily distinguishable. Note CWND is not *directly measurable*, and techniques that *estimate* CWND by forcing timeouts have a shorter set of observations. This makes it more difficult to interpret these CWND estimations rather than queue occupancy measurements. CCAnalyzer is able to achieve better accuracy than prior work, with the important additional benefit of interpretable results.

4.3 Efficiency

We have two measures of efficiency: total bytes transferred and wall-clock time. Using our testbed experiments, we measured that for CCAs supported by Gordon, CCAnalyzer requires on average 15% fewer bytes to perform classification and completes probing 40x faster in terms of wall-clock time. IG is more efficient than Gordon and CCAnalyzer. For all of these classifiers, classification is inexpensive and done offline after collecting traces, so here we only consider the efficiency of collecting the traces before classification. We collect pcaps for all experiments and measure the average amount of bytes transferred between the web server and the client for classifying each CCA. In addition, we measure the time from the first packet sent from the client to the last received from the web server.

Bytes transferred. Fig. 17a compares the number of bytes transferred between CCAnalyzer and Gordon. Because Gordon waits to measure the reaction to packet loss, the time and amount of data transferred to classify a webpage is heavily dependent on the CCA. Because BBR does not respond to individual packet losses, it transmits more data during the measurement and requires a special test to classify. In contrast, CCAnalyzer's classification is not as dependent on the CCA, aside from CDG which doesn't always manage to maintain full throughput, data transferred is independent of the underlying algorithm. The mean number of bytes transferred for CCAnalyzer over the 13 CCAs supported by Gordon is 68MB (total for collecting 4 traces) while for Gordon it is 456 MB (with a large std dev. of 186) since it heavily depends on the CCA. IG only collects 1 trace for up to 50 RTTs without any repetitions or restarts and at most transfers 2MB. However, since IG only emulates a single timeout, this efficiency comes at the cost of failing to capture the cyclical nature of CCA behavior, leading to worse accuracy than CCAnalyzer and poor interpretability of generated traces (see Fig. 16).

Time comparison. Fig. 17b compares the amount of time it takes to collect traces for CCAnalyzer and Gordon. CCAnalyzer only needs 20s per trace. Including setup, CCAnalyzer overhead takes only a maximum of 30s per measurement and is not dependent on the CCA. Since we collect 4 traces for each CCA the total amount of time for trace collection for CCAnalyzer is about 2 minutes. In contrast, Gordon's runtime heavily depends on the CCA with a max of 130 minutes and a minimum of 2.6 minutes to complete all of its 15 trials. IG takes at most 90s to collect traces.

5 CCA Measurement Study of Top 10K Websites

We conduct a measurement study using our comprehensive tool and our testbed in Fig. 2. We have two goals here. The primary goal is to demonstrate the effectiveness and robustness of CCAnalyzer in classifying known CCAs and detecting novel CCAs. We show how we can detect a new CCA, BBRv3, with minimal effort. The second goal is to take steps towards answering important questions about the current state of CCA deployment in the Internet today, for example: Is Cubic still the most dominant CCA? How has the deployment of BBR evolved? Is Reno deprecated?

5.1 Methodology

The Google Chrome UX Report (CrUX) releases rank ordered lists of top websites, which is more accurate than alternatives [45, 49]. We pull the websites from the Top 10K bucket from the February 2023 dataset, which accounts for 70% of all Chrome page loads [45, 58]. The websites in the CrUX dataset are identified by *origin*, not domain. For example, this list includes `www.google.com`, `scholar.google.com`, `maps.google.com`, and so on as separate websites, so we try to classify each of these separately. While we believe that this measurement study covers a large fraction of popular websites, and we draw some important conclusions, we do not claim to be a comprehensive Internet measurement study. We leave a larger measurement study for future work (which is considerably more feasible with CCAnalyzer than prior work).

Both Gordon and Inspector Gadget had to search websites for a webpage large enough to download to generate `CWND` traces. We found we could only classify 1% of the 10K websites with IG primarily because we could not find large enough files (Appendix §C). Similarly, we need a web transfer between the client and server for at least 20s. To achieve this goal without requiring large files, we use the `h2load` [4] tool to send multiple parallel HTTP requests to the websites we want to classify to download enough data from the webpage to utilize the available bandwidth (5Mbps, 10Mbps). We use the `findcdn` [2] tool to identify if a website is hosted by a CDN. Occasionally, this tool returns more than one CDN for a given website. In those cases, we use the first result returned by this tool.

Unresponsive and invalid traces: Table 3 shows a summary of how many websites we were able to successfully classify and their classifications. 34% of these 10K websites are "unresponsive" because they did not respond to pings or the homepage did not respond with a `200 OK` response to `h2load`. 13% had RTTs that were larger than 85ms. In addition, we measure the bandwidth utilization for each trace. We set a bandwidth threshold of 80% because for all our training samples the CCA is able to use at least 90% of the available bandwidth; a threshold of 80% gives some headroom. A trace is marked invalid if it does not meet the bandwidth threshold. A website is marked as "All Invalid" if all of the traces collected for that website do not meet the bandwidth threshold. 9% of the websites have traces that are all invalid.

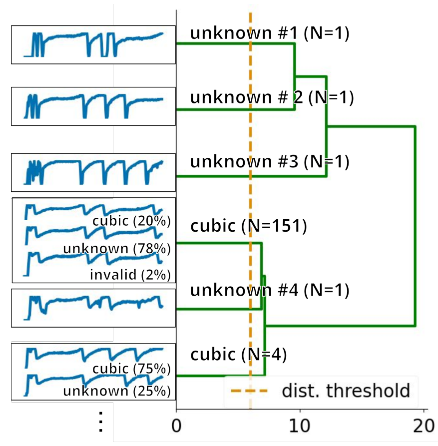


Figure 18: Example of a portion of a dendrogram from hierarchical clustering Fastly websites in 5bw-85rtt-128q setting. The vertical line is the distance threshold.

Validation and clustering within CDNs: We initially classify each web server using the methodology described in §4.1, and report those numbers in Table 3 (the numbers before slashes). We notice about 1600+ websites are marked as unknown which means most of the traces for these websites were not close enough to their closest training sample. Recall in §3.3.4, when we determined the distance threshold, we set a small T based on experiments to an AWS server. We favored *false unknowns* vs. *false knowns*. Because of the likely possibility of more noise in our measurements to third-party servers, the distance threshold may be too conservative. To further reduce *false unknowns*, we do an additional clustering step where we may re-classify websites. The values after the slashes are the counts per CDN, per CCA if there were changes after this additional clustering step.

In this additional step, we cluster all the CCAs that are in the same CDN using agglomerative clustering [42]. Agglomerative clustering works by putting each sample initially in its own cluster, and then merging samples into the same cluster based on the distances between the samples. We use the "average" metric based on the DTW distances between all samples, which links samples to minimize the average of the distances of each observation of the two sets. Once we compute the links between all the samples, samples can be put into clusters based on a distance threshold; we use the distance thresholds described in Table 2. If a resulting cluster contains 5% or more labeled traces, we re-classify unknowns or invalid traces as that label. For the cases where we re-classify, while the traces initially labelled unknown are not close enough to other *training samples*, they are close enough to other *testing samples* with a known label. We believe in this case, these are *false unknowns* and should be given a known label. After re-classifying traces, we re-do the voting across the 4 settings (§3.4) and give websites potentially new labels (we do not re-label 'All Invalid' websites).

We see two notable changes after this clustering of all samples from the same CDN. First, the majority of the unknown Fastly websites (105) are re-classified as Cubic. Second, there is a similar shift with Cloudflare websites: 362 websites are re-classified as BBR. In Fig. 18, we visually show the partial output of the clustering of

Table 3: Classification results for websites by CDN websites. The values after the slashes are after a clustering step on traces within each CDN.

CDN	BBR	BIC	CDG	CUBIC	Highspeed	HTCP	NV	Reno	Vegas	Westwood	Yeah	Unknown	All Invalid	RTT > 85ms	Unresponsive	Total
Akamai	470/491	0	3/4	4	0	0	0	0	0	0	0	115/91	189	36	233	1050
Cloudflare	1233/1595	0	6/7	5/6	0	0	0	1	0	0	0	824/460	394	55	989	3507
Cloudfront	530/545	0	9/10	7/10	0	0	3/2	0	0	0	0	74/56	78	10	121	832
Fastly	21/25	1/13	3	25/130	0	0	0/1	0	0	0	0	174/52	26	3	30	283
Google	29	0	1	2	0	0	2	0	0	0	1	230	37	18	66	386
Other CDN	28/32	2/0	1	53/92	0	0	1	0	0	0	0	72/31	54	41	226	478
No CDN	116/122	3/0	8/9	89/116	3/5	2	5	4/3	1/0	3	0	146/115	127	1205	1752	3464
Total	2427/2839	6/13	31/35	185/360	3/5	2	11	5/4	1/0	3	1	1635/1035	905	1368	3417	10000

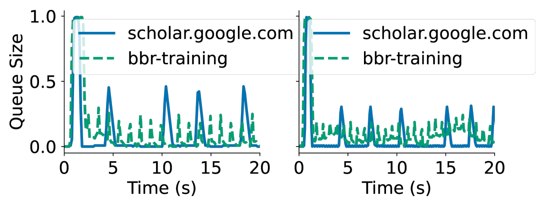


Figure 19: Example trace from a Google website that we believe is BBRv3 from 2 settings: 10bw-130rtt-128q (left), 10bw-85rtt-128q (right)

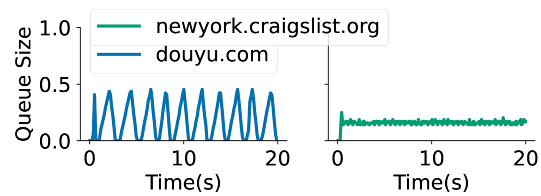


Figure 20: Example of unknown traces from websites not hosted by a CDN.

Fastly results. The yellow vertical dotted line shows the clustering distance threshold used. The labels on the green lines indicate the final label given to all traces in each cluster and the number of traces in the cluster. The boxes on the left show some example traces in each cluster, along with their initial label. For example, in the cluster labeled 'cubic (N=151)', 20% of these traces are Cubic traces so this cluster is labelled Cubic. It is encouraging that the these traces are highly similar and are clearly Cubic traces based on manual inspection. Similarly, the process does a good job keeping the 'unknown' label for unusual traces.

There are two benefits of this clustering step. First, we can validate the results of our classification. Looking at the dendrograms of the output we can visualize how close samples are to each other and can see at what distance threshold similar samples are clustered together (highlighting the interpretability of CCAnalyzer results). We expect like traces to end up close together, while dissimilar traces to be far apart. Second, which we demonstrate in Fig. 18, it can help classify false negative unknowns as actually known CCAs.

5.2 Clustering Unknowns

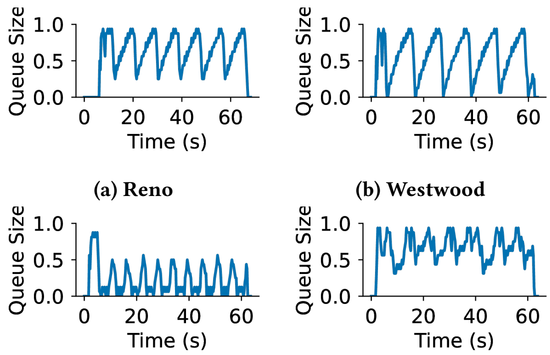
After the initial classification as well as the clustering within CDNs and validation, we now have websites that are still classified as unknown. We take the traces from all of these websites, across CDNs, and run agglomerative clustering on all of them. We manually view the dendrogram of these results and look for web servers that are likely using the same unknown CCA.

BBRv3: We notice that the majority of websites originating from Google CDN are classified as unknown when we expected to see BBRv1. We see these traces are closest to BBRv1, but the periods of bandwidth probing and RTT probing are more spaced out. Fig. 19 shows an example queue occupancy traces for 2 settings for `scholar.google.com`. We conclude these sites are using BBRv3 and confirm with Google's BBR team [7]. According to presentations from Google, BBRv3 was deployed on Google servers by Summer 2023 when our measurement study was conducted in Fall 2023. Based on clustering, we label 102 Google CDN websites originally classified as unknown instead as BBRv3. This example also highlights the ease in discovering new CCAs with CCAnalyzer.

Other Unknowns: Our ability to cluster traces using DTW distances makes discovering new CCAs a simple and straightforward process of reviewing queue occupancy traces, dendrograms and DTW distances. We see the potential for further reverse engineering of these CCAs using recent work [23]. While some unknowns can be re-classified as existing designs as part of the clustering process, we still see other behaviors that remain classified as unknown, like clusters unknown #1-#3 in Fig. 18. In our broader study of websites, we find several websites using unknown CCAs which we highlight in Fig. 20. Note that further study is needed to determine if these are truly novel CCA designs or a known CCA in an unexpected or pathological state.

5.3 Addressing Limitations

We find our RTT limit (85ms), bandwidth utilization threshold (80% of 5Mbps and 10Mbps), and `h2load` settings, limit the coverage we have for the websites we could measure in this study. As noted earlier, 13% had too high RTTs. 9% had too low bandwidth, and 34% of the servers did not respond to `h2load`. This is not a fundamental



(a) Reno (b) Westwood (c) BBR (d) Cubic

Figure 21: 0.3bw-275rtt-16q: Example training samples traces from Cloudlab

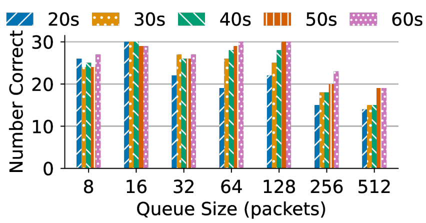


Figure 22: 0.3bw-275rtt: Accuracy for different queue sizes and trace lengths for 0.3Mbps and 275ms RTT setting using training and testing samples from Cloudlab server. A queue size of ~1BDP (16 pkts) works well.

limitation of 1NN-DTW and can be mitigated in several ways. In this section, we discuss how we could increase coverage, to potentially classify the 'unresponsive' websites and gather more valid traces in this study.

An obvious mitigation for the lack of response to `h2load` is to issue HTTP/1.1 requests if HTTP/2 requests fail. In addition, we could avoid using `h2load` by requesting a large file from the website using `wget` as we did in our evaluation. To reduce the number of servers with too high RTTs or too low bandwidth, we consider the smallest bottleneck bandwidth and largest RTT that we can use to generate clear queue occupancy traces. We consider an RTT constraint of 275ms based on Fig. 4. To determine a low bandwidth, we observe the queue occupancy plots as we decreased bandwidth by 0.1 Mpbs at a time. We did this until we visually observed a distinctive change in the shape. We find 0.3 Mbps produces distinguishable traces as shown in Fig. 21.

Given this bandwidth and RTT constraint, the core question is: what queue sizes will produce distinguishable traces? To answer this question, we generate 5 queue occupancy traces for varying queue sizes from within our testbed using `iperf` with a server that is a Cloudlab machine. We split these traces into 3 training and 2 testing samples for each of the 15 CCAs, using Cloudlab traces to classify other Cloudlab traces to determine which queue sizes result in high accuracy. Fig. 22 shows the accuracy for the 0.3 Mbps and 275ms RTT network settings with varying queue sizes and varying trace lengths. A queue size of about ~1BDP (16 packets) has an accuracy of 100% for traces as short as 20s, similar to the accuracy we see in §3.3.2. Fig. 21 shows the queue occupancy traces using a 16 packet queue.

CCAnalyzer can work with a wide range of network settings to support a large majority of web servers. We only need network settings that produce distinct queuing behavior for different CCAs for 1NN-DTW to work. While we only explore a few settings in this work, we also show that there other settings, especially ones with lower expectations from servers, that could be used in practice. Future work could use alternative network settings to classify websites that we did not in this study, improving coverage further without sacrificing accuracy.

5.4 Takeaways

Widespread deployment of BBRv1: While we are not able to classify all websites, we do find that the majority of those we can classify are classified as BBRv1. While we cannot conclude that there is an increasing deployment of BBRv1, we do see widespread deployment at "Hypergiants"[28] like Akamai which was previously known to have deployed a different CCA [41]. We so see some large CDNs (*e.g.* Fastly) still use Cubic.

Discovery of new CCAs: Using CCAnalyzer, we can automatically discover new CCAs, as we show in this work with our discovery of BBRv3. We do not need specialized or hand-crafted tests to classify new CCAs (like Gordon and Inspector Gadget) nor do we need to know details of how the algorithm works (like Gordon). We can add support for new CCAs, like BBRv3, simply by collecting queue occupancy traces and adding them to our training set.

6 Conclusion

CCAnalyzer takes a significant step forward in CCA classification. While only relying on collecting bottleneck queue occupancy traces, CCAnalyzer achieves accuracy that is equal to or better than state-of-the-art classifiers. In addition, CCAnalyzer is efficient, unobtrusive, has interpretable results, and supports open-set classification. We use CCAnalyzer to analyze the CCAs of 5000+ websites. CCAnalyzer's DTW-based distance measure allows it to not only detect unknown CCAs, but also cluster them into groups of similar unknowns, simplifying the detection and classification of new CCA variants as they appear on the Internet. Unlike prior work, CCAnalyzer's approach has the potential to classify the rising popularity of user-space protocols (*e.g.* QUIC) and other popular applications (*e.g.* video streaming), a promising direction for future work.
Ethics: This work does not raise any ethical issues.

Acknowledgments

We would like to thank SIGCOMM reviewers for excellent feedback that helped shape the final version of the paper. We thank Anne Kohlbrenner, Rukshani Athapathu, and Matt Mukerjee for their contributions to early iterations of this work. This work is supported by the NSF under award numbers 2212390 and 2007733 as well as a CMU Cylab Seed Grant, a Google Faculty Research Award, and a Facebook Emerging Scholars Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

References

[1] 2023. BESS: A Software Switch. https://github.com/NetSys/bess.

[2] 2023. findcdn. https://github.com/cisagov/findcdn.

[3] 2023. Gordon. https://github.com/NUS-SNL/Gordon/blob/master/Scripts/tcpClassify.py.

[4] 2023. h2load. https://nghttp2.org/documentation/h2load.1.html.

[5] 2023. Inspector Gadget. https://github.com/Brown-NSG/inspector-gadget.

[6] 2023. iperf3. https://software.es.net/iperf/.

[7] 2023. Private communication with Neal Cardwell.

[8] 2024. Private communication with Ayush Mishra.

[9] Akamai. 2012. Akamai Acquires FastSoft. https://www.prnewswire.com/news-releases/akamai-acquires-fastsoft-169678966.html

[10] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 329–342. https://www.usenix.org/conference/nsdi18/presentation/arun

[11] Anthony Bagnall, Jason Lines, Aaron Bostrom, James Large, and Eamonn Keogh. 2017. The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data Mining and Knowledge Discovery* 31, 3 (2017), 606–660.

[12] Andrea Baiocchi, Angelo P Castellani, Francesco Vacirca, et al. [n. d.]. YeAH-TCP: yet another highspeed TCP.

[13] Donald J. Berndt and James Clifford. 1994. Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining* (Seattle, WA) *(AAAIWS'94)*. AAAI Press, 359–370. http://dl.acm.org/citation.cfm?id=3000850.3000887

[14] Lawrence S Brakmo, Sean W O'malley, and Larry L Peterson. 1994. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*. 24–35.

[15] Bob Briscoe, Koen De Schepper, Olivier Tilmans, Mirja Kühlewind, Joakim Misund, Olga Albisser, and Asad Sajjad Ahmed. 2019. Implementing the'Prague Requirements' for Low Latency Low Loss Scalable Throughput (L4S). *Netdev 0x13* (2019).

[16] Carlo Caini and Rosario Firrincieli. 2004. TCP Hybla: a TCP enhancement for heterogeneous networks. *International journal of satellite communications and networking* 22, 5 (2004), 547–566.

[17] Neal Caldwell. 2017. TCP BBR congestion control comes to GCP – your Internet just got faster. https://cloud.google.com/blog/products/networking/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster

[18] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR Congestion Control. In *Presentation in ICCRG at IETF 97th meeting*.

[19] Dean Carmel and Isaac Keslassy. 2023. Dragonfly: In-Flight CCA Identification. In *2023 IFIP Networking Conference (IFIP Networking)*. 1–9. https://doi.org/10.23919/IFIPNetworking57963.2023.10186432

[20] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, Berkeley, CA, USA, 395–408. http://dl.acm.org/citation.cfm?id=2789770.2789798

[21] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 343–356. https://www.usenix.org/conference/nsdi18/presentation/dong

[22] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[23] Margarida Ferreira, Akshay Narayan, Inês Lynce, Ruben Martins, and Justine Sherry. 2021. Counterfeiting Congestion Control Algorithms. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks* (Virtual Event, United Kingdom) *(HotNets '21)*. Association for Computing Machinery, New York, NY, USA, 132–139. https://doi.org/10.1145/3484266.3487381

[24] S. Floyd. 2003. HighSpeed TCP for Large Congestion Windows. RFC 3649.

[25] Cheng Peng Fu and Soung C Liew. 2003. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on selected areas in communications* 21, 2 (2003), 216–228.

[26] Nitin Garg. 2019. Evaluating COPA congestion control for improved video performance. https://engineering.fb.com/2019/11/17/video-engineering/copa/

[27] Jim Gettys and Kathleen Nichols. 2011. Bufferbloat: Dark Buffers in the Internet. *Queue* 9, 11, Article 40, 15 pages. https://doi.org/10.1145/2063166.2071893

[28] Petros Gigis, Matt Calder, Lefteris Manassakis, George Nomikos, Vasileios Kotronis, Xenofontas Dimitropoulos, Ethan Katz-Bassett, and Georgios Smaragdakis. 2021. Seven years in the life of Hypergiants' off-nets. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 516–533. https://doi.org/10.1145/3452296.3472928

[29] Sishuai Gong, Usama Naseer, and Theophilus A Benson. 2020. Inspector Gadget: A Framework for Inferring TCP Congestion Control Algorithms and Protocol Configurations.. In *Network Traffic Measurement and Analysis Conference*.

[30] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 64–74. https://doi.org/10.1145/1400097.1400105

[31] David A Hayes and Grenville Armitage. 2011. Revisiting TCP congestion control using delay gradients. In *International Conference on Research in Networking*. Springer, 328–341.

[32] T Henderson, S Floyd, A Gurtov, and Y Nishida. 1999. The NewReno Modification to TCP's Fast Recovery Algorithm.

[33] Young-Seon Jeong, Myong K Jeong, and Olufemi A Omitaomu. 2011. Weighted dynamic time warping for time series classification. *Pattern Recognition* 44, 9 (2011), 2231–2240.

[34] Tom Kelly. 2003. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM computer communication Review* 33, 2 (2003), 83–91.

[35] Eamonn J Keogh and Michael J Pazzani. 2000. Scaling up dynamic time warping for datamining applications. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 285–289.

[36] Douglas Leith and Robert Shorten. 2004. H-TCP: TCP for high-speed and long-distance networks. In *Proceedings of PFLDnet*, Vol. 2004. Citeseer.

[37] Zachary C. Lipton. 2018. The Mythos of Model Interpretability. *Queue* 16, 3, Article 30 (June 2018), 27 pages. https://doi.org/10.1145/3236386.3241340

[38] Shao Liu, Tamer Başar, and Ravi Srikant. 2006. TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*. 55–es.

[39] Saverio Mascolo, Claudio Casetti, Mario Gerla, Medy Y Sanadidi, and Ren Wang. 2001. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 287–297.

[40] Pedro R Mendes Júnior, Roberto M De Souza, Rafael de O Werneck, Bernardo V Stein, Daniel V Pazinato, Waldir R de Almeida, Otávio AB Penatti, Ricardo da S Torres, and Anderson Rocha. 2017. Nearest neighbors distance ratio open-set classifier. *Machine Learning* 106, 3 (2017), 359–386.

[41] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. 2019. The Great Internet TCP Congestion Control Census. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 3, Article 45 (dec 2019), 24 pages. https://doi.org/10.1145/3366693

[42] Fionn Murtagh and Pierre Legendre. 2014. Ward's hierarchical agglomerative clustering method: which algorithms implement Ward's criterion? *Journal of classification* 31, 3 (2014), 274–295.

[43] Jitendra Pahdye and Sally Floyd. 2001. On Inferring TCP Behavior. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (San Diego, California, USA) *(SIGCOMM '01)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/383059.383083

[44] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '12* (2012). https://doi.org/10.1145/2339530.2339576

[45] Kimberly Ruth, Deepak Kumar, Brandon Wang, Luke Valenta, and Zakir Durumeric. 2022. Toppling Top Lists: Evaluating the Accuracy of Popular Website Lists. In *Proceedings of the 22nd ACM Internet Measurement Conference* (Nice, France) *(IMC '22)*. Association for Computing Machinery, New York, NY, USA, 374–387. https://doi.org/10.1145/3517745.3561444

[46] Stan Salvador and Philip Chan. 2007. Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intell. Data Anal.* 11, 5 (Oct. 2007), 561–580. http://dl.acm.org/citation.cfm?id=1367985.1367993

[47] Constantin Sander, Jan Rüth, Oliver Hohlfeld, and Klaus Wehrle. 2019. DeePCCI: Deep Learning-Based Passive Congestion Control Identification. In *Proceedings of the 2019 Workshop on Network Meets AI & ML* (Beijing, China) *(NetAI'19)*. Association for Computing Machinery, New York, NY, USA, 37–43. https://doi.org/10.1145/3341216.3342211

[48] Pasi Sarolahti, Markku Kojo, Kazunori Yamamoto, and Max Hata. 2009. Forward RTO-recovery (F-RTO): An algorithm for detecting spurious retransmission timeouts with TCP. RFC 5682.

[49] Quirin Scheitle, Oliver Hohlfeld, Julien Gamba, Jonas Jelten, Torsten Zimmermann, Stephen D. Strowes, and Narseo Vallina-Rodriguez. 2018. A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists. In *Proceedings of the Internet Measurement Conference 2018* (Boston, MA, USA) *(IMC '18)*. Association for Computing Machinery, New York, NY, USA, 478–493. https://doi.org/10.1145/3278532.3278574

[50] Joel Sing and Ben Soh. 2005. TCP New Vegas: Improving the performance of TCP Vegas over high latency links. In *Fourth IEEE International Symposium on Network Computing and Applications*. IEEE, 73–82.

[51] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (Princeton, NJ, USA) *(HotNets '19)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/3365609.3365855

[52] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Modeling BBR's Interactions with Loss-Based Congestion Control. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. ACM, New York, NY, USA, 137–143. https://doi.org/10.1145/3355369.3355604

[53] Keith Winstein and Hari Balakrishnan. 2013. TCP Ex Machina: Computer-generated Congestion Control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (Hong Kong, China) *(SIGCOMM '13)*. ACM, New York, NY, USA, 123–134. https://doi.org/10.1145/2486001.2486020

[54] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(NSDI'13)*. USENIX Association, Berkeley, CA, USA, 459–472. http://dl.acm.org/citation.cfm?id=2482626.2482670

[55] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. 2011. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, Berkeley, CA, USA, 99–112. http://dl.acm.org/citation.cfm?id=1972457.1972468

[56] Lisong Xu, Khaled Harfoush, and Injong Rhee. 2004. Binary increase congestion control (BIC) for fast long-distance networks. In *IEEE INFOCOM 2004*, Vol. 4. IEEE, 2514–2524.

[57] Peng Yang, Juan Shao, Wen Luo, Lisong Xu, Jitender Deogun, and Ying Lu. 2013. TCP congestion avoidance algorithm identification. *IEEE/ACM Transactions On Networking* 22, 4 (2013), 1311–1324.

[58] Zakir Durumeric. 2023. crux-top-lists. https://github.com/zakird/crux-top-lists.

Appendix

Appendices are supporting material that has not been peer-reviewed.

A Artifacts

The testbed will be made available to use via Cloudlab [22].

B Formal DTW definition

To understand DTW, we first consider a naive approach to compare two traces. Consider two queue occupancy traces, $X = (x_1...x_n)$ and $Y = (y_1...y_m)$, where x_i is the queue occupancy at time i in trace X, where X is n time steps long. A simple approach to measuring the difference between the two traces is to to calculate the Euclidean distance (assuming X and Y are the same length $n = m$):

$$ED(A, B) = \sqrt{\sum_{i=0}^{n}(X[i] - Y[i])^2}$$

Unlike ED, DTW allows a one-to-many mapping between X and Y: a given index from each trace can map to one or more indices in the other trace. DTW finds the best point mapping between two traces to minimize the sum of distances between all their points with two constraints: 1) The first and last indices must be mapped to one another and 2) the mappings must be monotonically increasing.

DTW finds the optimal "warp path", the one-to-many mapping between points in an NxM matrix where N and M are the lengths of two time series, that minimizes the overall distance between the time series. Exact mappings would be a diagonal line, but DTW accounts for phase shifts with horizontal and diagonal lines which show when many points in one trace are mapping to the same point in the other trace. For DTW the time series need not be the same length, although in this work we truncate all the traces to be the same size.

More formally, let $DTW(i, j)$ be the optimal distance between the first i and j elements in time series X and Y. Then, the value of $DTW(i, j)$ is defined recursively as follows:

$$DTW(i, j) = distance(x_i, y_i) + \min \begin{cases} DTW(i, j-1) & \text{repeat } x_i \\ DTW(i-1, j) & \text{repeat } y_i \\ DTW(i-1, j-1) & \text{repeat neither} \end{cases}$$

where $distance(x_i, y_i)$ may be defined in different ways including the squared difference which we use in Fig. 3; in the rest of this work we find the absolute difference works better for our use case $|x_i - y_i|$.

C Inspector Gadget Measurement Results

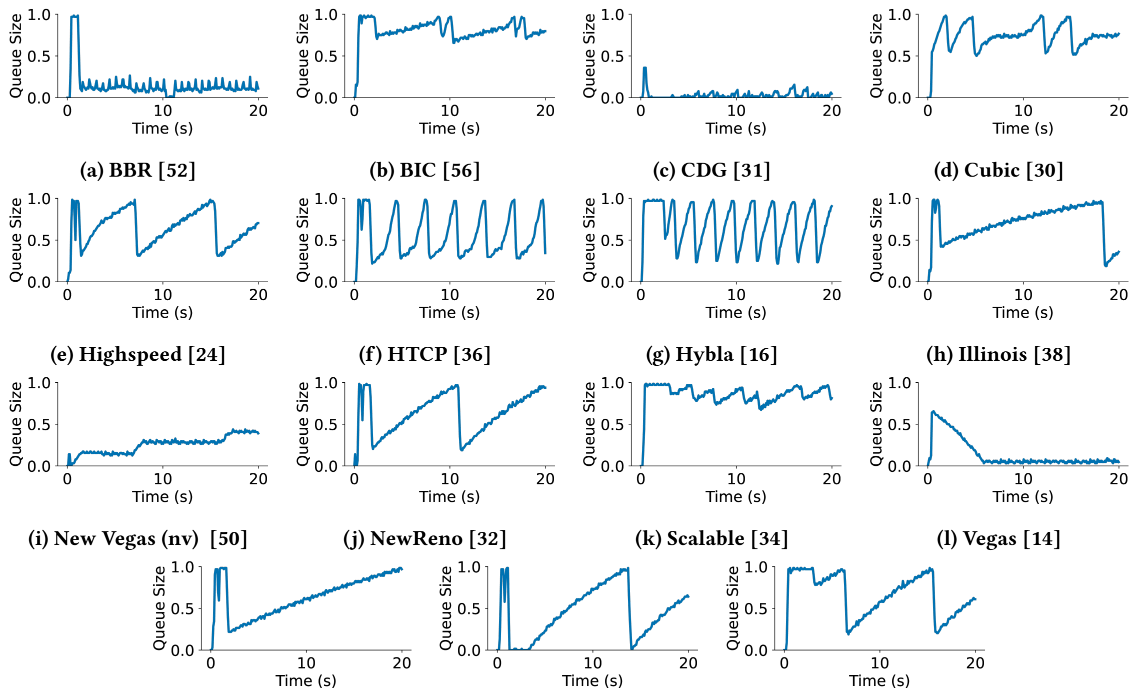
Table 4: IG measurement results for 10K websites

Result	Count
BBR	71
Cubic	25
Yeah	10
Highspeed	5
BIC	2
Not large enough object	9533
Trace collection fail	354
Total	1000

We re-implement IG to the best of our ability, because we could not run classification with the published code [5]. We are able to use the IG web crawler code to try and find large enough objects on the 10K websites we want to classify. From our controlled measurements (§4) we find that we need a file size of at least 1.5 MB. However, for 95% websites we could not not find files that large despite crawling up to 500 links per website. Ultimately, we are only able to successfully classify 113 websites. The labels for these websites are shown in Table 4 if we restrict the training set to the the CCAs in the IG paper.

D Example training samples

We highlight some example training samples for the 5bw-85rtt-64q setting in Fig. 23 to give some sense for what the traces look like for 20s for each CCA. These traces each capture some of the cyclical behavior of each CCA which helps CCAnalyzer to be accurate and interpretable.



(a) BBR [52] (b) BIC [56] (c) CDG [31] (d) Cubic [30]

(e) Highspeed [24] (f) HTCP [36] (g) Hybla [16] (h) Illinois [38]

(i) New Vegas (nv) [50] (j) NewReno [32] (k) Scalable [34] (l) Vegas [14]

(m) Veno [25] (n) Westwood [39] (o) Yeah [12]

Figure 23: Example CCAnalyzer CCA training sample traces from AWS-Virginia (5bw-85rtt-64q setting)

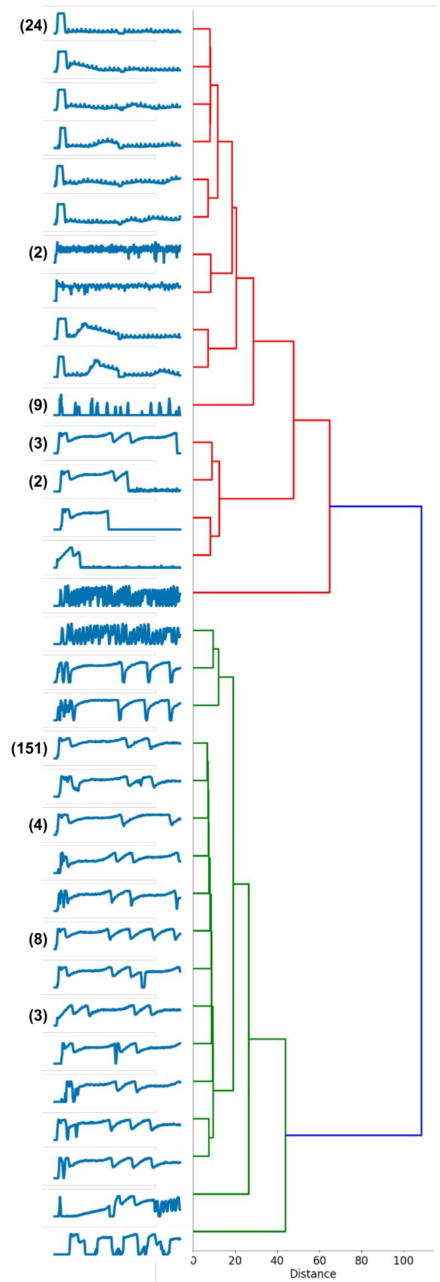


Figure 24: The full dendrogram of results from clustering across all the Fastly websites (5bw-85rtt- 128q setting) using the distance threshold. (§5) Each leaf shows one testing sample from each of the resulting clusters. Clusters with numbers indicate how many samples are in that cluster.