Inspector Gadget: A Framework for Inferring TCP Congestion Control Algorithms and Protocol Configurations.

Sishuai Gong
Purdue University

Usama Naseer
Brown University

Theophilus A. Benson
Brown University

***Abstract*—Over the last decade, in an attempt to improve end-user performance, the community has proposed a multitude of changes to the networking stack's configuration parameters. These changes range from new default values (e.g., initial congestion window size) to the development of new configuration options (e.g., congestion control protocols). While the networking community has performed extensive studies on the performance implications of configuration optimizations, these studies have been performed in isolation and, moreover, there are no holistic and general tools to infer, analyze, and understand the *actual* configuration choices employed by online content providers and content distribution networks.**

To this end, we present Inspector Gadget, a flexible and accurate framework for characterizing and fingerprinting a web server's networking's stack configuration parameters. Inspector Gadget leverages domain-specific heuristics to reverse engineer configuration parameters and options. To demonstrate the efficacy of Inspector Gadget, we implemented a prototype of Inspector Gadget and used it to survey the configuration parameters for the Alexa top 5K sites. To motivate this work, we surveyed network operators to get a qualitative understanding of their approach towards tuning their networking stacks and the root cause of configuration heterogeneity. To further illustrate our tool's strength, we fed the results into an emulator and analyzed the optimality and fairness of the current configurations for various websites.

I. Introduction

Society's evolving expectations for better Quality of Experience (QoE) and richer web services are transforming today's web server's networking stack. In particular, the community is continuously churning out "better" congestion control algorithms (CCAs), TLS, and HTTP protocols and configurations to deliver higher performance (QoE) [1]–[3]. This aggressive evolution of the networking stack is further spurred by the increasing diversity of networking conditions [2], [4], [5], end-user device capabilities [6], [7], and web page complexity [8], [9], all of which necessitate improvements to existing protocols. These protocols are often exposed as configuration parameters that must be explicitly tuned by content providers' operators and administrators. In Table I, we present a representative subset of TCP's configuration parameters and observe that some of the default or suggested settings, e.g., Initial Congestion Window (ICW), have changed multiple times over the last three decades.

Consequently, server administrators for online content providers and content distribution networks referred to as *content providers*, are faced with a jungle of configuration parameters, many of which promise improved performance. Unfortunately, recent studies on networking protocol configurations [2]–[5], [10] demonstrate that tuning networking configuration parameters is a non-trivial task that requires intimate knowledge about the complexity of the deployed websites (i.e., # of objects and size of objects) and the expected networking conditions (i.e., bandwidth, loss, and RTT) of the targeted clients. To further complicate configuration-tuning, the supported configuration parameters and their default values vary across OSes and even across different versions of the same OS.

Configuration Parameter	Suggested Value(s)
Init. Congestion Window	1 ('97) [11], 2 ('98) [12], 4 ('02) [13], 10 ('11) [14]
Initial RTO	3 secs ('00) [15], 1 secs ('11) [16].
Congestion Control	Reno [17], Bic ('04) [18], Cubic ('06) [19]
Connection startup	FastOpen ('14) [20]

TABLE I: Suggested TCP configuration values over time.

A direct result of this complexity is that it is not clear which configuration parameters are employed by modern websites or how these parameters vary across different websites and content providers. Yet, knowledge of these configurations remains crucial for reasoning about the current dynamics of the Internet in terms of performance, fairness, and protocol equilibrium. Without information about the current configurations, it is difficult to reason about how new protocols, or even existing protocols, will behave in the wild.

Our goal is to develop a framework that allows operators and researchers to conduct a census of the configuration parameters used by content providers and analyze the impact of parameter-tuning or lack thereof on global questions. Our work is motivated by the following questions:

(i) Do content providers tune their stacks, i.e., choose values other than the default values? (Section VI), (ii) Given the configured parameters, how do their flows interact with each other, in terms of fairness (i.e., bandwidth sharing)? (Section VIII-A), (iii) How optimal are the configurations currently employed by content providers? (Section VIII-B)

Although these questions are fundamental to understanding the health of the Internet and the implications of new protocols, they are difficult to answer because we lack the (1) algorithms and frameworks to infer the configuration of content providers, (2) a standardized setup consisting of a data set specifying the number of flows for each web service, and an emulator for replaying and analyzing these interactions. While emulators and datasets exist, the existing tools can not holistically infer a web server's configurations across the transport and application layers. This deficiency is problematic because performance and fairness are a function of interactions between the different configurations and the magnitude of traffic using the different configurations,

The first step to answering these questions lies in designing a framework that can passively analyze a web server and infer the server's networking stack's configurations and parameters. Given such a tool, many of these questions can be answered through

978-3-903176-27-0 ©2020 IFIP

controlled experiments in simulations or emulations. In this paper, we present Inspector Gadget, a system that allows operators and researchers to answer these questions by inferring the protocol configurations and parameters of the network stack of modern content providers. It consists of the following components:

- A set of active measurement techniques for identifying a web server's current protocol configurations.
- A framework for using our inferences techniques to scan websites on the Internet scalably.
- A toolbox of analysis methodologies and an emulator to analyze the interactions between different configurations and protocols within realistic network conditions.

Although Inspector Gadget includes an emulator and data, our key contribution lies in the framework and accompanying algorithms for accurately inferring the configuration parameters and values for modern web services. The design of these frameworks and algorithms is complicated by the following factors: (i) while some settings are part of the packet headers (e.g., TCP or HTTP options), the most interesting ones (e.g., congestion control) can only be inferred using domain-specific heuristics; (ii) any inference algorithms must be agnostic to network conditions, server OS and other factors that are outside control of end user frameworks such as ours; (iii) the framework must be flexible enough to support future configuration parameters and general enough to capture the broad range of existing configuration parameters.

We implemented Inspector Gadget and evaluated it on the Alexa top 5K websites. To understand the root cause of configuration heterogeneity, we surveyed top network operators and captured information about their approach to configuration tuning.

The key findings and insights of our measurements are:

- **Protocol Dominance:** Contrary to popular opinion, there is no overwhelmingly dominant protocol. In fact, the existence of multiple dominant protocols indicates a need to revisit modeling frameworks: from modeling and analyzing interactions between two protocols to interactions between various protocols.
- **Regional Differences:** Configurations in N. America differ significantly from configurations used in other regions, indicating a need for regional modeling of performance and fairness. Based on our measurements and surveys, there is evidence that many web servers are not using default values but rather specifically tweaked values based on internal objectives.
- **Optimality of Selected Configurations:** While many web services are tuned to provide good performance, these configurations do not provide optimal performance across all network conditions. Thus, there is a need to perform region specific tuning, as users in developing regions may face radically different network conditions than in developed regions [21].
- **Fairness:** Unsurprisingly, the current Internet is unfair. However, the degree of unfairness changes depending on which configurations are used, and thus there is a need to analyze the Internet to determine current protocol configurations periodically.

Inspector Gadget is implemented in Python and is open-source. The code can be found at https://github.com/Brown-NSG/inspector-gadget.

Layer	Parameters	TBIT [22]	CAAI [23]	Gordon [24]	Deep CCI [25]	Inspector Gadget	Explicit defined
TCP	initcwnd	✓	✓	✓	X	✓	X
	Congestion Control	X	✓	✓	✓	✓	X
	slow start after idle	X	X	X	X	✓	X
	Timeout (RTO)	X	X	X	X	✓	X
	FastOpen (TFO)	X	X	X	X	✓	✓
	rcv. buffer	X	X	X	X	✓	✓
TLS	TLS Version	X	X	X	X	✓	✓
	Session ticket	X	X	X	X	✓	✓
	OCSP stapling	X	X	X	X	✓	✓
	ALPN	X	X	X	X	✓	✓
	TLS Renegotiation	X	X	X	X	✓	✓
	Forward secrecy	X	X	X	X	✓	✓
HTTP	HTTP version	X	X	X	X	✓	✓
	keep_alive timeout	X	X	X	X	✓	✓
	max conc. streams	X	X	X	X	✓	✓
	initial window size	X	X	X	X	✓	✓
	max header list size	X	X	X	X	✓	✓
	max frame size	X	X	X	X	✓	✓
	header table size	X	X	X	X	✓	✓
	stream priority	X	X	X	X	✓	X

TABLE II: Inferred configuration parameters: while the tool can infer TLS/HTTP parameters, this paper focuses on TCP. Please refer to [26] for application layer measurements.

II. Related Works

The most closely related works [22], [25], [27]–[30] analyze the configuration and behavior of TCP. Unlike Inspector Gadget, CAAI [30] and TBIT [27] do not interoperate with TLS/SSL. Although TBIT [27] can support TLS with the help of Scamper [31], it is limited to just a few TCP settings and does not infer modern CCAs (Table II). Compared with Gordon [24], our tool employs optimizations to tackle various domain-specific problems (e.g., pacing). Unlike DeepCCI, which supports three CCAs (i.e., Cubic, BBR, Reno), Inspector Gadget supports a broader set of CCAs. While other approaches have studied the implications of configuration and protocol choice at the HTTP and the TLS layers [2], [10], [32], our current work introduces innovation along a complementary direction – namely the transport layer (i.e., TCP). Inspector Gadget characterizes a broader range of CCA protocols and networking stack parameters than prior works (e.g., TCP *slow start after idle*, *RTO*, *FastOpen*, and *RMem*, crucial web-tuning configurations [33]) and also outperforms them in terms of inference accuracy (as we show in Section V).

The crucial difference with our previous work [26] is that, while our previous work [26] presented an option parser module for the TLS and HTTP layers, here we introduce the behavior parser which allows our tool to explore the transport layer. In addition to the tool, we also present a novel measurement study on the TCP related anomalies that we encountered while fingerprinting different Linux implementations in-the-wild (Section VI-F).

While our work analyzes the configurations and protocols of deployed web-services, others [2], [4] have performed a quantitative analysis of the implications of different parameters and protocol versions. These measurement studies motivate our desire to understand practical deployments. Our work goes a level deeper by conducting testbed-based and in-the-wild experiments to empirically evaluate fairness, performance, and the operational implications of the choices made at the transport level.

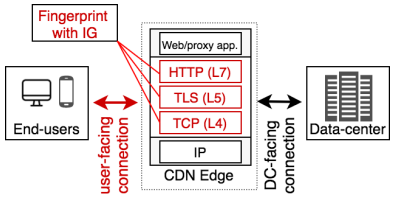

Fig. 1: Overview of content provider infrastructure.

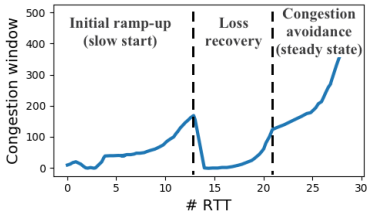

Fig. 2: CWND evolution for Cubic.

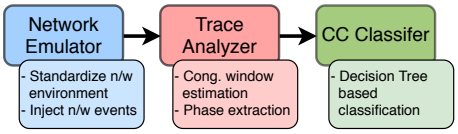

Fig. 3: Workflow for Behavior Parser.

III. Background

In this section, we provide a brief overview of the networking stack of modern content providers (Section III-A) and present detailed background on the transport layer (the central focus of our tool) (Section III-B).

A. Networking Stack

The web serving stack, Figure 1, consists of the TCP/TLS/HTTP protocol implementations, and the content provider's web application (e.g., PHP or Java code). Traditionally, these servers employ broadly two different network stacks – one for user-facing connections and another for the data-center facing connections. The goal of this work is to fingerprint and analyze the user-facing networking stacks. Content providers tune them to improve performance across all the users [1]. Table II presents a representative list of such tunable parameters across the different layers of the stack.

B. Transport Background

The transport protocol controls the sending rate and aims to ensure full utilization of the network and fairness among connections sharing the bottleneck link. To this end, the protocol monitors the network and uses signals from the network to infer its fair-share while maximizing utilization. There are roughly three types of CCAs: delay-based (e.g., Vegas), loss-based (e.g., Cubic), and hybrid (e.g., BBR-v2). Regardless of the type of CCA, data is exchanged between a sender and a receiver. The latter acknowledges the receipt of the data packets by sending special acknowledgment packets, frequently called ACKs. The congestion control algorithm (CCA) limits the number of packets that the sender can send at once – called the *congestion window* or *CWND*. The size of the CWND evolves over the lifetime of the connection.

Figure 2 plots the evolution of CWND for Cubic's CCA. The distinct sending rate behaviors and their transition for modern CCs can be categorized into three phases: (1) An initial ramp-up phase (AKA slow-start) where the protocol tries to understand available network bandwidth. (2) A steady-state (AKA congestion avoidance) during which the protocol probes the network to react to network dynamics; and (3) A loss recovery phase where-in the protocol reacts to packet loss.

During the ramp-up phase, the initial window size is a predefined protocol configuration called the initial congestion window (ICW). The transition between ramp-up and steady-state is often due to a protocol-defined configuration value called initial SlowStartThreshold (initSSThresh): the transition occurs when the CWDN becomes larger than initSSThresh. To detect losses, the protocol includes a predefined Timeout, called a retransmission timeout (RTO), which the protocol uses to decide how long to wait before a packet is considered lost.

Additionally, for delay-based congestion control protocols, the protocol reacts to changes in delay (or RTT) during steady-state, and often, there is no explicit loss recovery phase.

IV. Inspector Gadget

The goal of Inspector Gadget is to infer a *web server's* networking stack's configuration parameters and values. In this work, we focus mainly on the transport protocol because we recently analyzed other aspects, i.e., TLS and HTTP, in our workshop paper [26]. Moreover, today we lack a highly accurate tool to fingerprint the transport layer — congestion control algorithms and their parameters.

Given our objectives, the main design requirements for Inspector Gadget are:

- **Generality:** It must be OS and network agnostic, and its accuracy must not be affected by heterogeneity in the network's conditions between the framework and the target web server.
- **Compatibility:** It must interoperate with TLS/HTTPS to be compatible with modern web-services. While trivial, this is an important requirement because most existing tools are incompatible with modern websites (Table II).
- **Evolvable:** The protocols are constantly changing and evolving, Inspector Gadget must be easily extensible to account for new protocols and protocol evolution.
- **Accuracy:** Lastly, and perhaps most obvious, Inspector Gadget must provide highly accurate results under varying network conditions and server configurations.

Inspector Gadget overcomes these challenges by leveraging the following observation: most configuration parameters impact the spatial, structural, or temporal aspects of a flow's packets. While some parameters impact the size of packets (e.g., MSS option), others impact the spacing between packets (e.g., congestion control), and, yet, some impact the header values of the packets (e.g., TCP options). Thus, by examining flows within a *controlled setting* along these three dimensions, we can infer the protocol's configuration parameters.

Although the target server and the physical Internet infrastructure are beyond an end user's control, and hence beyond our control, the reliance of the TCP CCAs on client-generated ACKs allows Inspector Gadget to virtualize the network infrastructure and emulate a virtual network with appropriate network events (e.g., packet loss). This controlled virtual network allows Inspector Gadget to infer a server's CCA based on the CCA's reaction to the induced events while enabling us to limit network variance and heterogeneity.

Inspector Gadget is composed of the following two modules, which allows it to address the different configuration parameters:

A. *Options Parser Module*

This module parses TCP headers for options and header fields to help infer the configuration values that are exposed and exchanged within the header, e.g., TCP Fast Open. Our current options parser accepts a list of rules which maps the bits in a packet's header to the corresponding configuration values and enables us to extract them easily. By designing the options parser around a set of predefined rules, we can easily extend Inspector Gadget by adding new rules. This module is used to fingerprint the "explicitly-defined" configurations in Table II.

B. *Behavior Parser Module*

The congestion control protocol used in a connection is not explicitly exchanged as an option in TCP headers or agreed explicitly upon during the TCP handshake. Thus, to determine these configurations values, we designed Behavior Parser module, which infers the configurations based on the set of packets exchanged between the server and our tool.

Recall, a traditional CCA has three phases, and it transitions through these phases in response to different triggers (i.e., internal and external events). The initial transition between ramp-up and steady-state (also between loss-recovery and steady-state) is an internal state transition triggered by protocol dynamics (e.g., Cubic uses initSSThresh). In contrast, the transition to loss recovery requires an external event (e.g., loss-event). For delay-based congestion control algorithms, e.g., Vegas and BBR, the transitions during steady-state are in response to delay variations. Thus, to fingerprint these CCAs, our tool must emulate events that force these transitions.

Workflow: In Figure 3, we present the Behavior Parser's workflow. Step-1 is to interact with the target server, inject network events, and extract a clean trace of packet timings and *cwnds* (§IV-B1). Step-2 comprises of processing the *cwnd* trace to extract a *cwnd* vector (§IV-B2). Finally, in step-3, a classifier is used to infer the CCA protocol based on the CWND trace (§IV-B3).

1) ***Step 1: cwnd Trace Creation****:* As CCAs control the rate of flow (i.e., *cwnd* size), the trace comprises of the observed *cwnd* over time. Further, network events (loss or delay variations) are introduced, and the trace captures their impact on *cwnd*. There are several practical challenges in consistently generating a "clean" trace over the Internet: (1) standardizing the impact of network dynamics, (2) efficiently injecting network events, (3) accurately capturing the *cwnd*, and (4) dealing with practical network challenges. Below, we explain how we deal with practical challenges in generating a "clean" *cwnd* trace over the Internet.

- **Standardizing network environment:** The captured packet trace may be noisy due to the Internet's inherent variability (e.g., organic loss or delay). To keep the network equivalent across different protocols and servers, we artificially inflate the RTTs by controlling (or pacing) the rate at which ACKs are sent. This enables us to maintain the latency and, in turn, emulate a network that obscures network variability. Recall, from the protocol's perspective, that latency and bandwidth are a function of the Acknowledgement packets (ACKs) received in response to the sent data packets.

In designing our virtual network abstraction, we observed that emulating short RTTs is ineffective as they are unable to mask network conditions, and emulating huge RTTs are ineffective because they trigger RTOs. Thus, an appropriate RTT value must strike a balance between being large enough to mask organic noise while being small enough to avoid triggering RTOs. Through empirical measurements, we settled on emulating an RTT value of 0.8s, and we note that we can dynamically tune this value as network conditions change.

- **Injecting network events:** There are two options for injecting loss: (i) dropping packets and emulating timeout [34], (ii) sending duplicate ACKs [11]. While the latter keeps the protocol in steady-state and does not force a transition into loss recovery, we opt for the former because it allows us to circumvent OS and network idiosyncracies, e.g., Linux's burstiness control [30].

Additionally, for delay-based congestion control algorithms, e.g., BBR and Vegas, where losses are unimpactful, we modulate the delay within our virtual network to enable Inspector Gadget to fingerprint them. Specifically, during the last phase (Phase 3 in §IV-B2), we inflate the RTT by a constant factor (i.e., 50ms) every RTT.

- **Congestion window estimation:** Traditionally, systems estimate *cwnd* by counting the number of packets received within an RTT. However, these approaches assume TCP is synchronous and ordered: at the start of an RTT, packets are sent in a batch and ACKs for the batch are received towards the end of the RTT. In practice, we observe that packets from a single *cwnd* are spread out over multiple RTTs (e.g., due to pacing). Further, packet duplication and loss renders this simple approach inaccurate. To accurately capture the *cwnd*, we designed the following optimizations:

– *Sequence Check (SC):* Due to packet reordering and duplication, the number of packet inflight may differ from the actual congestion window. Thus, simply counting the number of inflight packets without accounting for duplication or reordering will yield an incorrect estimate of window size. Our first optimization is to re-use TCP's sequence number to identify and eliminate such redundant and reordered packets.

– *Window Emptying (WE):* For reasons like TCP pacing, the in-flight packets may not accurately reflect the current CWND. We force the server to transition into a more synchronous mode by batching ACKs and sending them at the end of the RTT to address this class of issues. This batching ensures that when Inspector Gadget sends ACKs at the end of the RTT, the server's window is empty, and the number of inflight packets is a more accurate representation of the server's state. Although WE can lead to slightly inflated RTTs, it has minimal effects on accuracy, as we demonstrate in Section V.

2) ***Step 2: Extracting cwnd Analysis****:* Given the *cwnd* trace, the next step is to convert the traces into a time series of *cwnd*. Recall, each CCA consists of three phases, and our vector captures all three phases.

In capturing the vector, our goal is to consistently capture the underlying invariants for a specific CCA. This implies that we should not capture the raw *cwnd* because they may vary slightly between runs. Instead, our vector (Equation 1) is divided into three components one for each phase. Each component of our vector, captures the *cwnd* trace as a series of offsets. In particular, for each componnent, we capture each *cwnd* as an offset from the first *cwnd* in the phase (e.g., W_{l+2} - W_{l+1}, W_{l+3} - W_{l+1}).

$$\nu = (\sum_{i=1}^{l} W_i - W_1, \sum_{i=1}^{x} W_{l+i} - W_{l+1}, \sum_{i=1}^{y} W_{s+i} - W_{s+1}) \quad (1)$$

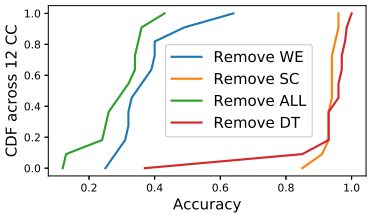

Fig. 4: Analysis of Inspector Gadget's optimizations on accuracy.

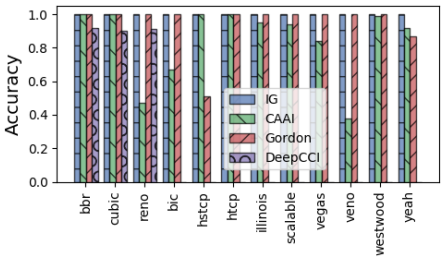

Fig. 5: Comparison of Inspector Gadget with CAAI, DeePCCI, and Gordon.

ID	Config{CC, ICW, IRTO RWIN, HTTP, F-RTO}	coverage [%]	Popular CDN
1	{Cubic, 10, 1, 29200, 1.1, 1}	24%	Fastly
2	{Cubic, 10, 0.3, 42780, 2,1}	17%	N/A
3	{Cubic, 10, 1, 29200, 2, 1}	14%	FBCDN
4	{BBR, 10, 0.3, 42780, 2, 1}	10%	Google
5	{Cubic, 10, 1, 14600, 1.1, 1}	7.5%	N/A
6	{Cubic, 4, 0.3, 29200, 2, 1}	7%	Akamai
7	{Cubic, 24, 1, 29200, 1.1, 1}	5.8%	Amazon
8	{Reno, 4, 1, 29200, 1.1, 1}	4%	N/A
9	{Reno, 4, 1, 29200, 2, 1}	3.2%	N/A
10	{BBR, 30, 0.3, 29200, 2, 1}	3%	Cloudflare

TABLE III: Description of the top ten configurations.

*3) **Step-3: Protocols Classification**:* The last step is to classify a protocol based on its *cwnd*-vector. To do this, we train a learning algorithm, i.e., a decision tree. We select the decision tree over alternative learning approaches for its ease, simplicity, and proven effectiveness within the networking domain [35]. Our decision tree classifier uses the CART algorithm [36]. We do not limit the depth of the tree and use "entropy" information gain to measure the quality of a decision node's split. We show in Section V, that using a learning algorithm improves the accuracy and effectiveness of Inspector Gadget by allowing Inspector Gadget to generalize and effectively classify protocols in the presence of protocol noise.

C. Inference Limitations

Central to Inspector Gadget's accuracy is the task of reliably collecting a sufficient number of *clean* packet traces from the target server to extract *cwnd* vectors and infer CCA fingerprints. The following practical issues limit Inspector Gadget's ability to collect these traces.

Large Object: Inference requires a long stream of packets from the server. In addition to setting the MSS to a small value to increase the number of packets transmitted (similar to [29]), we identify large objects on a website by scraping the websites before fingerprinting.

Network Middleboxes: Proxies or application-layer middle-boxes often terminate user connections; in such situations, Inspector Gadget will infer the middlebox's configuration instead of the web-server. However, existing techniques for detecting middle-boxes [37] can be used to identify such cases.

ECN-based Protocols: In our current design, we do not use ECN. However, we note that our framework can identify ECN-enabled TCP senders by extending the Options Parser, which we plan to do in the future.

UDP-based Protocols: Although our current focus is on TCP, we believe that our techniques can be extended to infer the CCA's for UDP-based protocols (e.g., QUIC).

V. Validation

In this section, we validate the accuracy of our techniques.

Experiment Setup: To validate Inspector Gadget, we perform controlled experiments where we run web servers in the cloud(AWS) and the Inspector Gadget tool on a server on the east coast to fingerprint these servers. Since we have control over the cloud web servers, we can determine the ground truth and validate our techniques. Additionally, by running the servers in the cloud and the Inspector Gadget tool at our local campus, we ensure that we analyze Inspector Gadget across realistic network dynamics. Additionally, we also used the Linux traffic control tool to emulate different network conditions. On our web servers, we run Linux 4.15 servers with Apache as the web application, which hosts a 5MB file. We fingerprint each CCA 30 times for each network condition and request objects multiple times concurrently to capture at least 4000 packets. We use 10-fold cross-validation: we train Inspector Gadget's decision tree classifier on 90% of the data and test on the remaining 10%.

Related work DeepCCI [25] infers CCA based on packet arrivals at the network's bottleneck. To perform comparisons with DeepCCI, we add a NAT to the front of our client to emulate a bottleneck and synchronize the captured packet arrival times before and after this bottleneck. Unfortunately, our analysis of DeepCCI is limited because it comes pre-trained with models of three CCAs: Cubic, Reno, and BBR.

Ablation analysis: We begin, in Figure 4, by analyzing the impact of the different optimizations within Inspector Gadget on its accuracy. For classification without a decision tree, we use conventional feature extraction techniques. We analyze the trace to get congestion control features like multi-decrease value or congestion window offsets from the congestion avoidance phrase and use these features for classification. We observe that without all our techniques, accuracy falls by as much as 62.00% (in the worst case).

Among all three optimizations, the window estimation (WE) optimization is the most important one. From Figure 4, we observe that removing WE has a significant impact on prediction accuracy across all CCA. We observe up to 31% false-positive rate in classification in the absence of WE. Since the *cwnd* trace characterizes a CCA, accurate *cwnd* estimation is the foundation of CCA inference. WE is responsible for correctly synchronizing the start of each different quantum in a *cwnd* trace. In its absence, the Sequence Check feature may incorrectly disregard packets as being outside of the current window. Moreover, in the classification step, both the decision tree and feature classifier, rely on the information from the trace. If the trace itself is inaccurate, the inference would also suffer.

Comparison against existing techniques: Last, in Figure 5, we compare the accuracy of Inspector Gadget against three closest related work (CAAI [30], Gordon [24] and DeepCCI [25]). CAAI's open-source implementation in unmaintained and does not support TLS. Thereby, we reimplemented CAAI with support for HTTPS.

We observe that Inspector Gadget provides almost perfect identification across various CCA implementations. With our re-implementation of CAAI, we observe accuracy between 41% and 94%. Although CAAI's performance improves with our optimizations, CAAI is still suboptimal to Inspector Gadget in all

scenarios. Additionally, with many essential CCAs, CAAI's performance is unacceptable, e.g., in BBR or Cubic, where accuracies are 78% and 64%, respectively. We observe that while Gordon provides similar accuracy to Inspector Gadget in most CCA. However, in particular, CCA, e.g., hstcp, it's accuracy is as low as 50%.

To understand why Inspector Gadget outperforms the competition, we analyze the different techniques. For CAAI, we observe that CAAI emulates only two network conditions, both of which are not sufficient to capture the distinctions among certain CCAs. For example, Veno and Reno are identical except under very narrow conditions, which CAAI is not able to capture. For Gordon, we observe that the inaccuracies may be due to poor estimation by Gordon's congestion window size estimation algorithm.

Although we observed that DeepCCI works well (accuracy>96%) across the three CCAs that it supports when server and client are both hosted on the same cloud deployment (i.e., no WAN interactions), however, it degrades when the WAN is involved. For the scenarios where the client is in a university network and the server in AWS's cloud, the accuracy drops to 90-92%. This is a severe limitation since the goal of such a tool is to fingerprint configurations across the Internet. We believe that DeepCCI's performance is limited because it is trained using testbed generated data, which lacks the Internet's inherent noise. Further, we observe differences in CCAs across different kernels (§ VI-F), and ideally, DeepCCI's model would need to be re-trained for different kernels to capture such behavioral differences appropriately.

Takeaway: Our optimizations enable Inspector Gadget to tackle protocol idiosyncrasies and network dynamics, which, in turn, allows Inspector Gadget to out-performing the existing state of the art techniques.

VI. Configuration Census

In this section, we discuss the results of our empirical study of the transport layer configuration values used by the Alexa top 5K websites.

A. Global Configuration Census

We begin by analyzing the number of unique configurations. We define a configuration as a specific set of parameters across the transport and HTTP layer parameters listed in Table II. We observed a total of 620 unique configurations across the 5K sites surveyed. In Table III, we present the top ten configurations and observe that they account for 92.06% of the websites, with three configurations account for over 50% of the sites. Although there is significant heterogeneity, many websites re-use common configurations. Note that the top three configurations use the Linux default for CC and ICW (Cubic with 10MSS).

B. Congestion Control Algorithms

Next, we focus on analyzing the impact of site ranking, country (as captured by Geolocation services) and CDN (as captured by CDN identification tool [29], [38]) on the distribution of CCAs employed.

Regardless of the rank, we observe that Cubic is the most popular choice (Figure 7). However, we observe that towards the lower spectrum of popularity, there is an increase in the use of Reno and a decrease of BBR. This can be explained by the fact that BBR is currently adopted by the more popular sites, e.g., Google.

In analyzing adoption across regions, we observe a similar trend, i.e., Cubic has the highest adoption (Figure 6). The critical difference between the regions is that within N.A. (North America), Cubic is dominant, and BBR accounts for a significant share of the CCAs. Whereas in the other areas, Reno is often the dominant protocol with BBR used by very few sites. Across CDNs (Table III) we observe: (1) that CDNs use either Cubic or BBR, with Cubic being preferred, and (2) that there are no CDNs using unconventional protocols [1].

Takeaway: We observe that across the different regions, ranks, and CDNs, no single protocol strictly dominates. We find a stark difference between North America, which is governed by newer protocols and other regions that are still dominated by older protocols. The implications are that any efforts to model, analyze, and understand the performance and fairness of any protocol needs to account for regional differences by creating per-region models and performing per-region analysis.

C. Initial congestion window analysis

We observed that ICW ranges from 1 to 32 with three prevalent modes: 4, 10 and 30. We observe a non-trivial set of websites (i.e., 21.07%) are using the old default values of 4. We observe most of these websites to be in the lower spectrum of popularity (100-10K). 10, being the Linux default and suggested by recent RFC [14], is the most prevalent, with 55% adoption. We observe 30 to be the 3rd most common ICW setting. A significant proportion of websites to be using non-default values indicate explicit effort (tuning) and can have implications, both for performance and fairness.

D. initRTO analysis

We observed three distinct initRTO values: 300ms, 1s, and 3s. The default initRTO value changed from 3s (RFC2988) to 1s (RFC6298 [16]). While 3s is still used by a minority of websites (7%), ∼34% of websites use 1s, including Amazon, Facebook, eBay, Instagram, CNN, BBC, PayPal etc. Around 60% of tested web servers use 300 ms as initRTO, much smaller than the suggested value. These websites include Google's search page for various geographical regions, bing, youtube, office etc. The shorter initRTO enables the web server to react more quickly to packet loss and proves useful in high loss environments like wireless networks or cellular networks.

E. Regional differences

Fingerprinting configurations in five regions, North America (N.A.), South America, Asia, Europe, and Australia, we observe that four regions (South America, Asia, Europe, and Australia) are identical in their configurations. Whereas N. America behaves in a distinctly different manner. For a set of websites, we observed higher BBR adoption in N. America (N.A.) than the rest of the world (Figure 6).

[1] Akamai uses FastTCP for certain network conditions; however, our emulated network conditions fall into the range of conditions for which they use Cubic

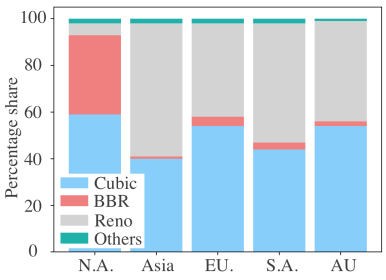

Fig. 6: Protocols by region.

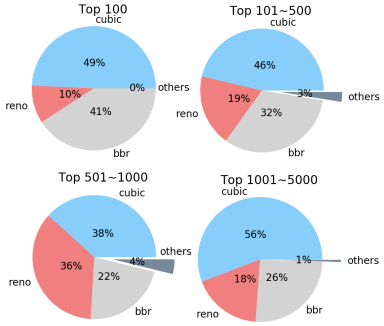

Fig. 7: Protocols by ranking.

Web service	TCP CC	ICW	# conn.	RTO
rai.it	Cubic	5	26	1
vimeo	Cubic	10	1	1
sky.it	BBR	10	19	1
pornhub	BBR	10	17	1
other	BBR	10	7	1
deezer	Cubic	10	5	1
facebook	Cubic	10	3	1
default	Cubic	10	1	1

TABLE IV: Fingerprinted configurations.

F. Anecdotes and Protocol Quirks

During our measurements, we observed some non-conventional behaviors:

- After a loss event, TCP dynamically sets SSThresh based on *cwnd* at the moment of the loss. However, we observed a kernel patch in Linux Kernel 3.18 RC5 violated the RFC specification and reverted SSThresh to initSSThresh.
- We observed that for some kernel versions (4.3, 4.4, and 4.5), TCP was stuck in a F-RTO loop, and *cwnd* remains fixed at 1MSS even after the retransmitted packets are acknowledged.

VII. Configuration Survey

We conducted a survey to understand better the current approach that content providers and their operators take to tune their networking stacks. The target audience of our survey are NANOG members [39], a group of top network operators across the world.

Survey population: Individuals from six distinct organizations (CDN/CSP) responded to the survey. The participants covered a diverse range of CDNs/CSPs, with active users ranging from 10K to over 100 million and server deployments ranging from 100 to over 10K servers. Three of the participants hosted over 1000 customers/services.

We summarize the highlights of the survey below:

- **Do operators tune their networking stacks?** Surprisingly, all of the surveyed operators noted that they did, in fact, tune the following parameters: TCP CC, ICW, TCP w_mem, pacing, queuing, and HTTP version[2]. Interestingly, one operator noted that they tuned their TCP CCA configurations dynamically, with the ideal CCA chosen based on end-user characteristics (i.e., AS) and workload (i.e., object type).
- **How do operators tune their networking stacks?** The surveyed operators followed one of the following two strategies for finding the right configurations: (i) use the latest recommendations provided by IETF, IRTF, NANOG, or the research community, or (ii) select the configurations that provide the best performance based on manual tests.
- **Which features control configuration tuning?** The top feature used by operators in our study is the workload, i.e., object type, with the operators using different configurations for different workloads (e.g., web, video, etc.). However, four of the operators also indicated that region (e.g., geo-location, AS) played a role in tuning.
- **Do CDNs allow their tenants to tune transport protocols?** While anecdotal evidence suggests that some CDNs do enable their tenants to tune configurations, surprisingly, the surveyed operators do not allow their customers to tune these transport configuration parameters. The operators cited fairness as a principle reason for not exposing these tuning functionality. In particular, aggressive protocols by one customer can harm other customers sharing the server.
- **Inter-PoP/DC-facing vs user-facing connection tuning.** Three of the operators noted that they optimized inter-PoP/DC-facing connections differently from user-facing ones, e.g., optimizations to re-use TCP connections efficiently, due to differences in characteristics between these connections.

Takeaway: Our survey with the network operators suggests that CDNs/CSPs significantly tune their networking stacks. Given these observations, understanding the protocol heterogeneity is not only crucial but required.

[2]tuning is defined as either using non-kernel default setting or customizing their kernel implementation

Config	Jain's 10MB	Jain's 100MB
Default	0.99	0.99
Fingerprinted	0.88 ±0.03	0.85 ±0.02
Fingerprinted w. single conn.	0.97 ±0.01	0.9 ±0.02

TABLE V: Jain's fairness for the different configurations.

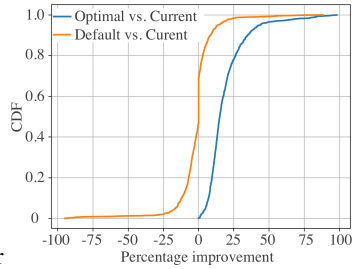

Fig. 8: Performance comparison of the best configuration against the default configuration.

VIII. Use-cases

In this section, we leverage our framework to analyze the Internet, with a focus on understanding the fairness and performance-optimality of the currently employed transport configurations.

A. Use-case 1: Fairness

Our goal with these experiments is to explore the impact of configuration on fairness as captured by Jain's fairness [40].

Experiment Setup: To emulate realistic conditions, we used data from a large ISP traffic [41] to determine: (1) the distribution

of flows, (2) destination websites of these flows, and (3) popularity of each website, i.e., number of flows for each website. Using Inspector Gadget, we fingerprinted destination websites. We present a summary of the fingerprints for the top 10 websites based on the number of flows, in Table IV.

In our local testbed, we emulate a simplified version of this ISP network with a single bottleneck link for all flows. We configured each flow to use its fingerprinted configuration. To understand the interactions between the network and configurations, we ran this experiment with several different bottleneck link configurations and explored the impact of object size on performance. To help ground our experiments, we created two baseline experiments. The first, called *default*, represents an experiment where the only thing we changed was to force all connections to use the default configuration. The second, called *Fingerprinted w. single conn.*, represents an experiment where representative configurations are used; however, each website has only one flow. This second baseline equalizes the websites and allows us to understand how they compete with each other if they had an equal number of connections.

Results: In Table V, we present Jain's fairness [40] for the ISP connections, across different object sizes. We make several observations. First, large objects result in higher unfairness when different content providers are allowed to use different configurations (column two is always larger than column one). Second, the current state of the internet (at this large ISP) is unfair (row two); however, this unfairness is significantly reduced if each website is forced to maintain the same number of flows (column three).

Takeaway: These distinct observations highlight the need for representative configurations and also for representative traffic. A subtle observation from our results is that analysis with different traces will yield fundamentally different results that may not necessarily comport with each other.

B. Use-case 2: Configuration Optimality

In this section, we analyze the optimality of the current transport configuration used by the top 10K websites. In particular, we aim to answer the following question: "Are the current configurations optimal across different network conditions? If not, how far is the performance of each configuration from the optimal configuration?".

Experiment Setup: Leveraging a testbed of 16 servers, we performed a brute-force exploration of the entire configuration space (i.e., TCP CC, ICW, RTO, HTTP version). For each combination of configurations, we emulate representative network conditions (i.e., cable, 3G, 4G, DSL, etc.) using NetEm and TC and set the bottleneck buffer to the bandwidth-delay product. We use Linux's "sysctl" interface and "ip route" for tuning the configurations. We load ten websites, picked randomly from Alexa top 100, five times for each network condition and configuration, and measure page load time (PLT). For each combination of website and network conditions, we select the configuration that consistently provides the lowest PLT as the *optimal* configuration. [3]

Optimality Analysis: Figure 8 presents a comparison of the fingerprinted configurations against two cases: (i) Linux's *default*, (ii) *optimal* for a given combination of website and network conditions. We observe that for the median combination of website and network conditions, the default and current configurations perform comparable, and in analyzing the configurations, we discovered that the median website currently uses the default configurations. However, the default is not optimal, and these content providers at the median can improve performance by as much as 10% by switching from the default to the optimal configuration.

Additionally, at the tails, the performance difference becomes even more pronounced. The *current* configurations perform worse than the *default* (e.g., -15% at p10) and *current* configurations also performs worse than optimal at the other tail (e.g., p99.99), indicating that current efforts to tune are suboptimal and can be improved.

Configuration Parameter Analysis: We conclude by analyzing the individual configuration parameters to understand the relative importance of each parameter. To measure the impact of tuning specific parameters, we set all parameters except those being tuned to their default setting. We observed that the top three knobs are: HTTP version, TCP CC, and ICW. Additionally, simply tuning TCP CC and ICW together, improves median PLT by 16% (as compared to *default*) while tuning *HTTP-only* improves performance by 11%.

Takeaway: Our analysis shows that the diversity of network conditions makes it challenging to pick optimal configurations. Moreover, existing techniques for choosing configurations fall short of discovering the optimal ones and often find configurations that perform worse than the default. We believe that our results raise an important question. Namely, how can a content provider dynamically and automatically tune the configuration of her networking stack? To date, this question remains an open question. We believe it will become increasingly important as the number of network conditions continues to grow and that websites continue to become complex. Our preliminary analysis shows content providers can reap the benefit of tuning by designing a system that tunes even a small number of knobs, which may reduce the barrier for entry for such tuning systems.

IX. Conclusion

There is a growing resurgence in research on improving web performance by enhancing transport protocols. A necessary step to evaluating and analyzing these efforts is to understand the dynamic interactions of the proposed protocols with other protocols (especially when competing for resources). Yet, today we lack sufficient tools for understanding how these protocols are configured in the wild and, in turn, what set of protocols to evaluate against realistically. We envision a "laboratory-in-a-box" framework, which includes tools to scan the Internet to infer configurations, including a sterile emulator for analyzing interactions between protocols.

Inspector Gadget presents the first step towards this vision. Inspector Gadget consists of a set of techniques for fingerprinting and inferring the configurations of modern web servers. Inspector Gadget also includes a set of analysis scripts and emulator-setup scripts to configure and analyze interactions between protocols.

X. Acknowledgments

We thank the anonymous reviewers and our shepherd, Simone Ferlin, for their invaluable comments. We, also, thank the anonymous NANOG operators for filling out our survey. This work is supported by NSF grants CNS-1409426 and CNS-1814285.

[3] different websites and network conditions may have diverse optimal configurations as web performance is a function of a given network and webpage structure [2], [3].

References

[1] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing tcp's initial congestion window." Computer Communication Review, vol. 40, no. 3, pp. 26–33, 2010.

[2] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How speedy is spdy?" in NSDI, 2014, pp. 387–399.

[3] U. Naseer and T. Benson, "Configtron: Tackling network diversity with heterogeneous configurations." in 9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17). Santa Clara, CA: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/hotcloud17/program/presentation/naseer

[4] M. Al-Fares, K. Elmeleegy, B. Reed, and I. Gashinsky, "Overclocking the yahoo!: Cdn for faster web page loads," in Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference. ACM, 2011, pp. 569–584.

[5] X. Nie, Y. Zhao, G. Chen, K. Sui, Y. Chen, D. Pei, M. Zhang, and J. Zhang, "Tcp wise: One initial congestion window is not enough." in IPCCC, 2017.

[6] S. Ahmad, A. L. Haamid, Z. A. Qazi, Z. Zhou, T. Benson, and I. A. Qazi, "A view from the other side: Understanding mobile phone characteristics in the developing world," in Proceedings of the 2016 Internet Measurement Conference, ser. IMC '16. New York, NY, USA: ACM, 2016, pp. 319–325. [Online]. Available: http://doi.acm.org/10.1145/2987443.2987470

[7] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman, "Impact of device performance on mobile internet qoe," in Proceedings of the Internet Measurement Conference 2018. ACM, 2018, pp. 1–7.

[8] M. Butkiewicz, H. V. Madhyastha, and V. Sekar, "Understanding website complexity: measurements, metrics, and implications," in Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference. ACM, 2011, pp. 313–328.

[9] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan, "Polaris: Faster page loads using fine-grained dependency tracking," in 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). USENIX Association, 2016.

[10] T. Zimmermann, J. Rüth, B. Wolters, and O. Hohlfeld, "How http/2 pushes the web: an empirical study of http/2 server push," in IFIP Networking Conference (IFIP Networking) and Workshops, 2017. IEEE, 2017, pp. 1–9.

[11] W. Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," 1997, rFC 2001. RFC Editor. 1–6 pages. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2001.txt

[12] M. A. N. L. S. S. F. L. C. P. B. Technologies, "Increasing tcp's initial window," 1998. [Online]. Available: https://tools.ietf.org/html/rfc2414

[13] M. Allman, S. Floyd, and C. Partridge, "Increasing tcp's initial window," United States, 2002.

[14] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, 2013. [Online]. Available: https://tools.ietf.org/html/rfc6928

[15] V. P. ACIRI and M. A. NASA, "Computing tcp's retransmission timer," 2000. [Online]. Available: https://tools.ietf.org/html/rfc2988

[16] V. P. I. Berkeley, M. A. ICSI, J. C. Google, and M. S. CWRU, "Computing tcp's retransmission timer," 2011. [Online]. Available: https://tools.ietf.org/html/rfc6298

[17] M. Allman, V. Paxson, and W. Stevens, "Tcp congestion control," United States, 2003.

[18] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (bic) for fast long-distance networks," in IEEE INFOCOM 2004, vol. 4, March 2004, pp. 2514–2524 vol.4.

[19] S. Ha, I. Rhee, and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," SIGOPS Oper. Syst. Rev., vol. 42, no. 5, pp. 64–74, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

[20] Y. Cheng, J. Chu, S. Radhakrishnan, A. Jain, and Google, "Tcp fast open," 2014. [Online]. Available: https://tools.ietf.org/html/rfc7413

[21] Akamai, "akamai's state of the internet, q1 2017 executive summary," 2017. [Online]. Available: https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-executive-summary.pdf

[22] J. Pahdye and S. Floyd, "On inferring tcp behavior," in Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 287–298. [Online]. Available: http://doi.acm.org/10.1145/383059.383083

[23] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu, "Tcp congestion avoidance algorithm identification," IEEE/ACM Trans. Netw., vol. 22, no. 4, pp. 1311–1324, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1109/TNET.2013.2278271

[24] A. Mishra, X. Sun, A. Jain, S. Pande, R. Joshi, and B. Leong, "The great internet tcp congestion control census," Proc. ACM Meas. Anal. Comput. Syst., vol. 3, no. 3, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3366693

[25] C. Sander, J. Rüth, O. Hohlfeld, and K. Wehrle, "Deepcci: Deep learning-based passive congestion control identification," in Proceedings of the 2019 Workshop on Network Meets AI & ML, 2019, pp. 37–43.

[26] U. Naseer and T. Benson, "Inspectorgadget: Inferring network protocol configuration for web services." in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), July 2018, pp. 1624–1629.

[27] J. Pahdye and S. Floyd, "On inferring tcp behavior," SIGCOMM Comput. Commun. Rev., vol. 31, no. 4, pp. 287–298, Aug. 2001. [Online]. Available: http://doi.acm.org/10.1145/964723.383083

[28] P. Yang, W. Luo, and L. Xu, "Towards measuring the deployment information of different tcp congestion control algorithms: The multiplicative decrease parameter," in 2010 IEEE Global Telecommunications Conference GLOBECOM 2010, Dec 2010, pp. 1–5.

[29] J. Rüth, C. Bormann, and O. Hohlfeld, "Large-scale scanning of tcp's initial window," in Proceedings of the 2017 Internet Measurement Conference, ser. IMC '17. New York, NY, USA: ACM, 2017, pp. 304–310. [Online]. Available: http://doi.acm.org/10.1145/3131365.3131370

[30] P. Yang, W. Luo, L. Xu, J. Deogun, and Y. Lu, "Tcp congestion avoidance algorithm identification," in Proceedings of the 2011 31st International Conference on Distributed Computing Systems, ser. ICDCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 310–321. [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2011.27

[31] M. Luckie, "Scamper: a scalable and extensible packet prober for active measurement of the internet," in Proceedings of the 10th ACM SIGCOMM conference on Internet measurement. ACM, 2010, pp. 239–245.

[32] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle, "Is the web ready for http/2 server push?" in Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies. ACM, 2018, pp. 13–19.

[33] S.-J. Yang and Y.-C. Lin, "Tuning rules in tcp congestion control on the mobile ad hoc networks," in 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06), vol. 1, April 2006, pp. 759–766.

[34] "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc793.txt

[35] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker, "Netprints: Diagnosing home network misconfigurations using shared knowledge," in Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 349–364. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1559001

[36] scikit-learn developers, "Decision trees." [Online]. Available: https://scikit-learn.org/stable/modules/tree.html

[37] D. Choffnes, P. Gill, and A. Mislove, "An empirical evaluation of deployed dpi middleboxes and their implications for policymakers," in Proc. of TPRC, 2017.

[38] "Cdn finder tool," https://www.cdnplanet.com/tools/cdnfinder/.

[39] NANOG, "The north american network operators' group," https://www.nanog.org/.

[40] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," Comput. Netw. ISDN Syst., vol. 17, no. 1, pp. 1–14, Jun. 1989. [Online]. Available: http://dx.doi.org/10.1016/0169-7552(89)90019-6

[41] M. Trevisan, D. Giordano, I. Drago, M. M. Munafò, and M. Mellia, "Five years at the edge: watching internet from the isp network," IEEE/ACM Transactions on Networking, vol. 28, no. 2, pp. 561–574, 2020.