TCP Congestion Control Fingerprinting

Kevin Ploch, Benedikt Jaeger*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: kevin.ploch@tum.de, jaeger@net.in.tum.de

***Abstract*—In order to make judgements on the spread of certain congestion control algorithms a way to fingerprint the algorithm of a host is needed. This can be done with congestion control identification (CCI) algorithms. This work presents the general approach of such an algorithm and summarizes possible categorizations for CCI. It presents Congestion Avoidance Algorithm Identification (CAAI) as an active and DeePCCI as a passive example. In an accuracy evaluation over a WAN connection these algorithms are compared to each other and to a more recent approach from 2020, Inspector Gadget (IG), which includes further optimizations. IG shows near perfect accuracy, DeePCCIs and CAAIs accuracies are rather humble, former with 90-92% and latter with 41-94%, and to conclude we explore how these results come to be.**

***Index Terms*—congestion control, congestion control identification**

1. Introduction

The biggest share of Internet traffic is under control of the Transmission Control Protocol (TCP) which combines concepts to provide properties like reliable and ordered data delivery or management of the sending-rate. The most performance significant part in data transmissions is the latter, namely Congestion Control (CC). Historically many TCP CC algorithms have been developed trying to find the best balance between bandwidth utilization and congestion in the network. While TCP Reno or TCP Cubic have been a wide-spread standard, whether now or in the past, newer algorithms like TCP BBR make their way into modern operating systems. [1, Section 1] [2]

To be able to make judgements on the current state of the Internet in topics like TCP performance, especially inter-algorithm-performance, or stability it is important to know the widespread adoption of each specific CC algorithm. Further examples that depend on this knowledge are topics like buffer sizing of routers, active queue management, fairness tuning of new CC variants or the creation of realistic traffic generators. To gather this needed, wide-spread deployment data we need a way to fingerprint the CC algorithm of a single host, leading us into the realm of Congestion Control identification (CCI) algorithms. [2], [3]

This paper aims to give an overview on current TCP CCI approaches. Section 2 revises important CC terminologies and differentiates the most important CC algorithms. Section 3 explains the general approach of a CCI method and presents different categories for them. Section 3.1 and Section 3.2 explain two unique approaches to CCI. Section 3.3 presents a more recent work with optimizations based on the two presented previous methods. Section 4 evaluates the accuracy of the three approaches and aims to explain the differences and pitfalls.

2. Background

Transport protocols determine the sending rate and have to balance between full utilization of network ressources and fairness among connections sharing a bottleneck link. Uncontrolled flows that exceed the speed at which a router can process packets leads to the build-up of packet queues and finally to dropped packets as the routers memory is exceeded [1, Section 2]. To accomplish this balance the protocol has to dynamically test for available bandwidth and congestion. There are 3 types of CC algorithms: delay-based (e.g. TCP Vegas), loss-based (e.g. TCP Cubic) and hybrid forms (BBR-v2) [4], [5]. Delay-based algorithms change their sending rate according to the delay of the connection, similarly loss-based change theirs in case of packet loss. [1], [5]

Every packet a sender transmits is acknowledged by the receiver with acknowledgement packets (ACK). The number of packets a sender is able to send unacknowledged in each round-trip time (RTT) is called the *congestion window (cwnd)*. Every CC has three phases: 1) slow start where the available bandwidth is estimated 2) a steady phase aka congestion avoidance during which we roughly stay at our calculated bandwidth limit and probe for more slowly; and 3) loss recovery where CC reacts to packet loss. [4], [5]

The value of the *slow start threshold (ssthresh)* determines the change from slow start to congestion avoidance. In case of a loss event it is usually changed according to $sstresh = \beta \cdot loss\_cwnd$ where β denotes the *Multiplicative Decrease Parameter* and *loss_cwnd* is the *cwnd* right before a loss event or timeout. The window growth function $g(\cdot)$ defines how TCP grows *cwnd* in the congestion avoidance state and it makes certain CC algorithms very recognizable, e.g. TCP Reno with linear growth or TCP Cubic with a cubic function to have a very sensitive growth around the *loss_cwnd* and a rapid one otherwise. [2], [4]

3. Congestion Control Identification

Table 1 lists a variety of CCI methods available to this date. Because reviewing every single one in detail would

doi: 10.2313/NET-2022-01-1_12

TABLE 1: Congestion Control Identification Overview

Method	Approach	Description	Included TCPs	Year
On Inferring TCP Behaviour (TBIT) [6]	active	trace congestion window to a given order of events, differentiate the 5 tcp methods based on this	4 (Tahoe, Reno, NewReno, TCP without Retransmit)	2001
Identification of different TCP versions based on Cluster Analysis [7]	passive	collect packets, extract features of cwnd based on a RTT estimate, cluster these to identify two competing CC variants	any 2 competing out of 14 (Reno, Cubic, BIC, CTCP, HSTCP, H-TCP, TCP Hybla, Scalable, Illinois, YeAH, Vegas, Veno, Westwood)	2009
TCP Congestion Control Avoidance Algorithm Identification (CAAI) [2]	active	extracts multiplicative-decrease parameter and window growth function, use machine learning to counter network conditions	14 (Reno, CTCP, BIC, Cubic, HSTCP, HTCP, HYBLA, ILLINOIS, LP, STCP, VEGAS, VENO, WESTWOOD+, YEAH)	2014
Identification of TCP Congestion Control Algorithms from Unidirectional Packet Traces [8]	passive	uses unidirectional packet trace, algebraic approach, approximate the whole SEQ-number to Time function, plot derivatives to differentiate TCP Congestion Control	5 (RENO, CUBIC, Hamilton TCP, Vegas, Veno)	2018
The Great Internet TCP Congestion Control Census (Gordon) [9]	active	similar to CAAI, different cwnd estimation algorithm	13 (BBR, Cubic, NewReno, BIC, HTCP, Scalable, Illinois, CTCP, YeAH, Vegas, Veno, Westwood, HSTCP)	2019
DeePCCI: Deep Learning based Passive Congestion Control Identification [3]	passive	only metric is packet arrival time, machine learning based classification with additional TCP Pacing differentation to increase identification accuracy, tests only in testbed	paper focused on BBR, CUBIC, RENO but trainable on any variant	2019
Inspector Gadget: A Framework for Inferring TCP Congestion Control and Protocol Configurations [5]	active	similar to CAAI with optimizations, especially improved network environments with changing RTT, Window Emptying and Sequence Check optimizations	12 (BBR, Cubic, Reno, BIC, hstcp, htcp, illinois, scalable, vegas, veno, westwood, yeah)	2020

go way beyond the scope of this work we are going to look at the general procedure of a CCI method and later dive into specific examples in Section 3.1 (CAAI) and 3.2 (DeePCCI). CAAI is from 2014 and while not being the newest active approach (see IG [5] and Gordon [9]), it gives a good understanding of the methodology. It's also based on a very early active approach in form of TBIT [6] from 2001. DeePCCI was chosen for pioneering an unconvential approach: ignoring TCP mechanics altogether and focusing only on packet arrival time.

Every CCI method follows a rough draft. First we need a way to get to the packet trace of the host we are interested in. Then we define features which enable us to differentiate between CC algorithms. These features are extracted from the packet trace and saved into a datastructure. Last but not least we match this processed representation of a host to some prepared data of each CC to classify the target. For this general procedure this work borrows the terminologies of **Trace Gathering**, **Feature Extraction** and **Algorithm Classification** from CAAI [2].

CCI methods can be generally categorized in two ways: TCP domain-dependent vs. TCP domain-independent and active vs. passive approaches. The first distinction differentiates between CCI methods that require knowledge of TCP in their implementation as they differentiate CC variants on subtle differences and methods that do not need knowledge of TCPs inner workings. Most methods are TCP domain-dependent but DeePCCI is a pioneer for the latter approach. The second distinction can be described in the following: Active approaches directly open up connections to a host to gain knowledge on the used CC by requesting data, manipulating the communication and observing the behaviour on the other side. Passive approaches do not interact with the observed host in any way. A host can be evaluated based on packet traces only, thus rendering this approach passive. One big advantage with passive approaches is the ability to work on real-world traffic. Packet traces could be captured on vantage points of a network or the Internet and therefore allow to identify alot of hosts without having to actively contact each one of them. A smaller subdistinction of passive methods can be made when looking at bidirectional vs. unidirectional packet traces. Most methods use bidirectional packet traces but for example Kato et al. [8] focused on unidirectional traffic only in 2018. [3]

3.1. Congestion Avoidance Algorithm Identification (CAAI)

CAAI is an active CCI method that is able to distinguish 14 different TCP algorithms as table 1 shows. Specifically its design goals aim to identify most default and non-default TCP algorithms while being insensitive to the operating system, network conditions and TCP components other than congestion avoidance of a webserver.

It characterizes each TCP congestion avoidance algorithm by two features:

- Multiplicative Decrease Parameter β

doi: 10.2313/NET-2022-01-1_12

- Window Growth function $g(\cdot)$

The context of these variables in TCP has been explained in Section 2.

The reasoning for these two variables lies in the fact that different TCP algorithms have different multiplicative decrease parameters and congestion window growth functions. For example RENO uses $\beta = 0.5$ and a linear growth function of $g(x, \textit{loss\_cwnd}) = 0.5 \cdot \textit{loss\_cwnd} + x$ where x denotes the number of elapsed RTTs in the congestion avoidance phase, while CUBIC uses $\beta = 0.7$ and a window growth function that not only depends on x but also on the duration of a RTT. [2]

Trace Gathering: In step one CAAI gathers the TCP congestion window trace of a target in two simulated network environments using Linux' netem [10] to introduce various network conditions. To be able to differentiate 14 different TCP versions P. Yang et al. argue one needs different network environments in terms of the RTT. They settled for two network environments, A and B, and defined Environment A with a static RTT of $1.0s$ and Environment B with a RTT of either $0.8s$ or $1.0s$ to more easily distinguish RTT based *cwnd* changes e.g. CUBIC. These simulated environments depend on two more variables: *cwnd_threshold*, which sets the boundary for the timeout to start and should be high to help distinguish TCPs, and *mss*, which sets the maximum TCP segment size of the connection and should be low to enable a higher maximum *cwnd* value. For detailed reasoning of specific values like the chosen RTT or minor problems like Slow Start Treshold Caching on the webservers refer to [2]. The data transmission itself is initiated with repeated HTTP requests in form of HTTP pipelining and a tool developed by the authors to search for the longest webpage of a webserver to give a long enough transmission.

As in Section 2 explained, *cwnd* equals to the number of packets sent in one RTT. CAAI uses this constellation to estimate the *cwnd* of the target. Packets can be assigned to a specific RTT as the RTT value is high enough to have a bandwidth-delay-product much larger than $\textit{mss} \cdot \textit{cwnd\_treshold}$. This leads to the packet trace having lots of packets at the start of each RTT and then a gap to the next one. Further CAAI uses the highest received sequence number in one emulated RTT to counter lost packets that would impact the $cwnd$ estimation. [2]

Feature Extraction: To extract the two features from the trace CAAI first determines the boundary RTT that marks the change from slow start to congestion avoidance. It then extracts β with $\beta = w_s / loss\_cwnd$ where w_s is the congestion window size at the boundary RTT and $loss\_cwnd$ denotes the window size right before the timeout. The ssthresh formula from section 2 was essentially solved after β. The second feature, window growth function $g(\cdot)$, is extracted from the congestion window sizes after the boundary RTT using two measuring points $w_{s+4} - w_{s+1}$ and $w_{s+9} - w_{s+1}$. The substraction allows values independent of $loss\_cwnd$, e.g. RENO would have a value of $w_{s+4} - w_{s+1} = 3$ as it features a linear growth. Two points are enough to distinguish the every CC, Yang et al. argue. The values from both environments are then saved into one feature vector.

Algorithm Classification: The problem with the gathered data is its dependence on the congestion in the network at the time of the trace gathering. As a countermeasure CAAI employs a machine learning algorithm trained on the feature vectors of the 14 different TCP variants in different network conditions. CAAI uses random forest in this regard as it achieved the highest classification accuracy among the tested methods. [2]

3.2. DeePCCI

DeepCCI is a passive and TCP domain-independent approach. Existing CCI methods have weaknesses like 1) complexity when adding new CC algorithms as detailed knowledge about parameters and configuration are needed, 2) assumptions of missing extern influences like TCP pacing or static parameters, and 3) reliablity on parsable TCP header information. One example outside of TCP itself for problem 1) and 3) could be QUIC which moves CC to the userspace and implements fully encrypted transports. To counteract assumptions like these Sander et al. developed DeePCCI which uses packet arrival time as its only feature to distinguish CC variants and therefore stays flexible. DeePCCI uses the packet arrival time as any CC algorithm controls the packet flow in terms of amount and timing Sander et al. argue. [3]

Packet traces do not need to be gathered by the tool itself in passive methods. The packets in the trace are sorted into same-sized bins according to their arrival time to build a histogram $X = [x_0, ..., x_t]$ of packet arrivals with equidistant timesteps. This histogram is then fed into a deep neural network consisting of a convolutional neural network (CNN) and a long short-term memory (LSTM) part. The former is regarded as the feature extraction phase while the latter builds up memory depending on previous behaviour and is needed to identify varying length traffic flows as they appear in the real world. After the LSTM layer the neural network predictions are applied for CC classification and classification whether TCP pacing was present in the trace to help in CC classification.

The testbed to train the neural network consists of two main topologies with different network conditions in terms of amount of TCP senders, link latency, bottleneck link bandwidth and bottleneck queue sizes. A bottleneck link is central for changing these variables. Topology 1 is a single-host network with one TCP sender connected to a router which is connected via a bottleneck link to another router followed by the receiving host. Topology 2 is a little more complex with 3 hosts on each side. The training data in the latter topology consists of all possible combinations of the 3 CC algorithms (RENO, CUBIC, BBRv1) that DeePCCI focused on. In each setting traffic is captured before and after the bottleneck link. If other senders exist they start sending 2 s prior to the sender we are interested in and it sends traffic for 60 s. [3]

The previously mentioned variables and other measurements decisions have an impact on the distinction of CC which will be discussed now. **Bandwidth**: DeePCCI was able to distinguish delay-based (BBRv1) and

doi: 10.2313/NET-2022-01-1_12

loss-based CC very well for bandwidths above 10 Mbps. Bandwidths $\geq$ 10 Mbps and delays $\geq$ 5 ms resulted in F1 scores above 90% in the multi-host scenarios and with F1 scores above 55% in the more unrealistic scenario with one TCP sender. By including smaller bandwidths also, the minimum F1 scores drop to 55% (multi-host) and 40% (single-host) respectively. As a general trend we see that larger bandwidths, larger delays and multiple simultaneous traffic flows are beneficial for the result.

A higher bandwidth results in a higher maximum congestion window and therefore more steps of e.g. CUBIC are executed leading to an easier distinction from linear behavior as with RENO. **Delay**: One effect of the delay can be accounted to the bin size. A low delay makes it harder to distinguish CC variants as the change in packet arrival time would be too fast for a distinguishable difference in the histogram bin sub-sampling. The CCs would have a less unique histogram composition. It also influences the decision whether TCP pacing was used, with a low delay too many packets fall into the same bin, with or without pacing, impacting the decision negatively. Lastly, the multi-host scenario increases the queuing delays and leads to more congestion and therefore a competition for packets in the queue. This effect counteracts the effect of a low bandwidth and delay as 1) the delay increases with competing flow and 2) competing flows influence the *cwnd* of our target host, identifying his CC easier as we are not that dependant on alot of steps as explained before. [3]

3.3. Inspector Gadget

Inspector Gadget (IG) is a more recent work from 2020 that aims to identify a whole webserver's network stack configuration ranging from new default values like initial window size to whole new CC protocols. Its approach is active, TCP domain-dependent and it evaluates self-captured, bidirectional packet traces. The work additionally surveyed individual network operators from six distinct content delivery networks to find out about their approach to tuning their network stacks and the root cause of configuration heterogenity. Also the tool itself was tested on the Alexa top 5k websites and the work discussed TCP related anomalies in form of a measurement study of different Linux implementations in the wild. While the work of IG covers alot of topics, in the context of this paper we are mostly interested in CCI itself which is covered in IV.B: *Behaviour Parser Module*.

To seperate IG from the two previously shown methods we first look at the differences. Unlike DeePCCI with 3 CC algorithms, IG originally supports a broader range of algorithms. In contrast to CAAI it's also interoperatable with TLS/SSL, offers more optimizations to tackle domain-specific problems like pacing and puts an emphasis on delay variations to fingerprint delay-based CC algorithms. [5]

Similar to CAAI IG manipulates the RTT through delaying ACKs, enabling a set RTT of 0.8 s. As mentioned before it is also varied to fingerprint delay-based CC. To inject loss-events IG, just like CCAI, provokes timeouts. The *cwnd* estimation bears the first bigger change. Estimating the window by counting the packets received in one RTT is prone to errors as this assumes that TCP is synchronous and ordered. Synchronous in the sense that at the start of a RTT the packets are sent in batch and the same for the ACKS at the end of a RTT. Also in the real world it can be observed that packets from one *cwnd* are spread over multiple RTTs due to e.g. pacing. Packet duplication and loss further worsen this scenario. To capture the *cwnd* accurately IG deploys two optimizations called *Sequence Check (SC)* and *Window Emptying (WE)*.

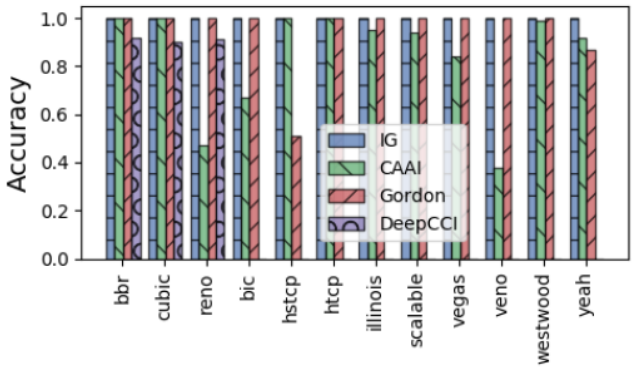

Figure 1: from [5]; Comparison of CAAI, DeepCCI, Inspector Gadget and Gordon

Packet reordering and duplication is a problem as the amount of packets in-flight differs from the actual congestion window. So instead of simply counting the number of packets in-flight IG uses SC to account for the TCP sequence number to identify and eliminate these cases.

Reasons like TCP pacing could also lead to a difference between packets in-flight and the real congestion window. Through the WE optimization ACKs are stored and sent batched at the end of a RTT. This ensures an empty sender window when sending the ACKs and a more accurate representation of the CC.

The *cwnd* trace is stored into the vector ν with each phase (Slow Start, Loss Recovery, Congestion Avoidance) seperated. Similar to Congestion Avoidance Algorithm Identification (CAAI) the values are saved as an offset, in this case from the first *cwnd* of each phase. For classification a decision tree with CART algorithm classifier was chosen. [5]

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

As we see it is quite similar to CAAI but offers some optimizations. How these are impacting the result will be evaluated in section 4.

4. Evaluation

To compare these works with each other, we will first look at the impact of the SC and WE optimization in IG. Then we will analyse a comparison of these three CCI methods and elaborate on the results and differences.

IG Optimizations. Excluding all improvements in IG yields an accuracy loss of 62% in the worst case. The WE

optimization had the biggest impact. Upon removal Gong et al. observed a false-positive rate of up to 31%. As a reminder, WE is responsible for synchronizing each RTT through ACK batching. In its abscence the SC feature has a higher probability to disregard packets as being not in the current window. This inaccurate trace leads to an inaccurate classification. Meanwhile a missing SC optimization leads to a minor drop to 92% from 100%. [5]

Accuracy Comparison. Gong et al. compared IG directly to CAAI, DeePCCI and Gordon, which we did not cover in this work except shortly in the CCI overview. To make results comparable, they extended CAAI's unmaintained source code with support for HTTPS and added a bottleneck to their setup to enable traffic capture on both sides for DeePCCI. The comparison with DeePCCI is somewhat restricted as it came with only 3 pre-trained CCs in form of Cubic, Reno and BBR. The environment itself consists of a server with the 4 CCI methods and webservers in an AWS Cloud to provide realistic network dynamics. The Linux traffic control tool was used to emulate different network conditions and each CC was fingerprinted 30 times for each network condition up to at least 4000 packets.

The results can be seen in figure 1. IG shows almost perfect identification across the different CC algorithms.

The re-implementation of CAAI featured accuracies between 41% and 94%. Even more disappointing is the poor result with major CC variants, for example BBR or Cubic with accuracies of 78% and 64% respectively. The reason for this may lie in the fact that CAAI emulates only two network conditions. Gong et al. argue that these two are not enough to capture small differences among certain CC algorithms, for example between Veno and Reno. In the original work of Yang et al. CAAI reached accuracies of 96.98% in their testbed but suffered of 53% invalid traces in Internet tests measuring 63124 popular webservers [2]. This number exists in the fact that CAAI could 1) not find a long enough webpage on a webserver to keep the connection up or 2) a webserver only accepts one or few HTTP requests in the same TCP connection and thus leading to slow start determining most of the data transmission [2].

Section 3.2 presented a first glance at the accuracy of DeePCCI under different testbed variables (e.g. amount of hosts, delay). It provides good results with accuracies above 96% in networks with client and server within the same area. But once WAN is introduced, as is the case in usual connections to webservers on the Internet, the accuracy drops to 90–92% in the tests of Gong et al. The main reason for this limitation might be the training with testbed generated data. Further DeePCCI might need to be retrained for CC variants across different kernels as they slightly differ. [5]

5. Conclusion

This work shed a light on CC fingerprinting. It reiterated on CC with its most important mechanics, phases and variables, and not only gave an overview on CCI algorithms, their methodology and possible categorizations but also reviewed three concrete examples with a finishing comparison in terms of accuracy under realistic WAN conditions. Furthermore an overview of CCI methods in form of table 1 has been provided.

References

[1] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock, "Host-to-Host Congestion Control for TCP," *IEEE Communications Surveys Tutorials*, vol. 12, no. 3, pp. 304–342, 2010.

[2] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu, "TCP Congestion Avoidance Algorithm Identification," *IEEE/ACM Transactions on Networking*, vol. 22, no. 4, pp. 1311–1324, 2014.

[3] C. Sander, J. Rueth, O. Hohlfeld, and K. Wehrle, "DeePCCI: Deep Learning-Based Passive Congestion Control Identification," pp. 37–43, 2019.

[4] M. Allman and V. Paxson, "RFC5681—TCP Congestion Control," *RFC*, no. 5681.

[5] S. Gong, U. Naseer, and T. A. Benson, "Inspector Gadget: A Framework for Inferring TCP Congestion Control Algorithms and Protocol Configurations." International Federation for Information Processing, 2020.

[6] J. Padhye and S. Floyd, "On Inferring TCP Behavior," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 287–298, Aug. 2001.

[7] J. Oshio, S. Ata, and I. Oka, "Identification of Different TCP Versions Based on Cluster Analysis," in *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, 2009, pp. 1–6.

[8] T. Kato, X. Yan, R. Yamamoto, and S. Ohzahata, "Identification of TCP Congestion Control Algorithms from Unidirectional Packet Traces," in *Proceedings of the 2nd International Conference on Telecommunications and Communication Engineering*, ser. ICTCE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 22–27.

[9] 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.

[10] "netem," https://wiki.linuxfoundation.org/networking/netem, accessed: 2021-08-03.

doi: 10.2313/NET-2022-01-1_12