Localizing Traffic Differentiation

Zeinab Shmeis
EPFL
Switzerland
zeinab.shmeis@epfl.ch

Muhammad Abdullah
EPFL
Switzerland
muhammad.abdullah@epfl.ch

Pavlos Nikolopoulos
EPFL
Switzerland
pavlos.nikolopoulos@epfl.ch

Katerina Argyraki
EPFL
Switzerland
katerina.argyraki@epfl.ch

David Choffnes
Northeastern University
United States
choffnes@ccs.neu.edu

Phillipa Gill
Google
United States
phillipagill@google.com

ABSTRACT

Network neutrality is important for users, content providers, policymakers, and regulators interested in understanding how network providers differentiate performance. When determining whether a network differentiates against certain traffic, it is important to have strong evidence, especially given that traffic differentiation is illegal in certain countries. In prior work, WeHe detects differentiation via end-to-end throughput measurements between a client and server but does not isolate the network responsible for it. Differentiation can occur anywhere on the network path between endpoints; thus, further evidence is needed to attribute differentiation to a specific network. We present a system, WeHe$\mathbb{Y}$, built atop WeHe, that can *localize* traffic differentiation, i.e., obtain concrete evidence that the differentiation happened within the client's ISP. Our system builds on ideas from network performance tomography; the challenge we solve is that TCP congestion control creates an adversarial environment for performance tomography (because it can significantly reduce the performance correlation on which tomography fundamentally relies). We evaluate our system via measurements "in the wild," as well as in emulated scenarios with a wide-area testbed; we further explore its limits via simulations and show that it accurately localizes traffic differentiation across a wide range of network conditions. WeHe$\mathbb{Y}$'s source code is publicly available at https://nal-epfl.github.io/WeHeY.

CCS CONCEPTS

• **Networks → Network performance analysis**.

KEYWORDS

Network Neutrality; Traffic Differentiation

ACM Reference Format:
Zeinab Shmeis, Muhammad Abdullah, Pavlos Nikolopoulos, Katerina Argyraki, David Choffnes, and Phillipa Gill. 2023. Localizing Traffic Differentiation. In *Proceedings of the 2023 ACM Internet Measurement Conference (IMC '23), October 24–26, 2023, Montreal, QC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3618257.3624809

IMC '23, October 24–26, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0382-9/23/10…$15.00
https://doi.org/10.1145/3618257.3624809

1 INTRODUCTION

Network neutrality—the notion that network providers should not differentiate between different applications and services—has been an important topic of debate. While early work on the topic focused on differentiation against BitTorrent traffic [10, 11], there is now strong evidence that many network providers differentiate against traffic from specific applications and/or services [23].

Neutrality is a divisive issue for the networking community. Some equate its violation with harmful, even malicious, behavior; most countries in Europe and Latin America, Canada, and several U.S. states take this position and have enacted neutrality regulations. Others argue that differentiation is part of a free economy [40]. Despite the myriad laws, rules, and opinions around neutrality, there is widespread agreement that transparency is vital for consumers to make informed and fair choices when selecting network providers.

Today, the most reliable test for differentiation against a given application's network traffic (e.g., a video streaming provider) is via end-to-end throughput comparison. This is the approach taken by WeHe [23], which comes with pre-recorded, made-in-the-lab, traces for testing popular network services (e.g., streaming, VoIP). To test for differentiation, the WeHe client and server replay both the original trace and a version of the trace with the original bits inverted. The latter retains packet sizes and timings from the original flows, but obfuscates the payload, including whatever criteria a network may be using to differentiate. If the bit-inverted replay achieves a significantly different throughput than the original one over multiple trials, WeHe concludes that there is traffic differentiation for the tested service somewhere on the path between server and client.

While WeHe does not assign blame when differentiation is detected, it is largely assumed that the client's Internet service provider (ISP) is the cause [1]. However, in principle, when two traffic flows between the same server and client achieve different throughput, the cause may lie *anywhere* on the network path. So, any claim that an ISP differentiates against a given application and/or service must come with harder evidence, especially given that differentiation is illegal in certain jurisdictions.

We present WeHe$\mathbb{Y}$: a system for localizing traffic differentiation within a client's ISP. In particular, we present WeHe$\mathbb{Y}$'s design and evaluation; we have a prototype, built on top of WeHe, that we have tested on real networks that apply traffic differentiation, but we have not yet released it to WeHe users. After WeHe detects differentiation on a path to a client, WeHe$\mathbb{Y}$ performs additional measurements

[1] In many cases (e.g., in the US), this can be confirmed with ground truth.

using the same client, analyzes them, and outputs one of two outcomes: (a) it found evidence that the differentiation happens within the client's ISP; (b) it found no such evidence and thus cannot provide any additional information relative to WeHe.

In this work, we assume a classic "Internet tomography" [7] setting: we need to reason about the behavior of a network (the client's ISP), but we can only perform measurements along end-to-end paths that traverse this network. In general, tomography systems perform measurements along multiple intersecting paths and rely on the performance relationship between these paths to reason about individual links. At a high level, this is what WeHe$\mathbb{Y}$ does, too: it performs measurements along paths that converge inside the client's ISP and end at the client; it relies on the performance relationship between these paths to reason about their common link sequence, which belongs to the client's ISP by construction.

Further, we assume no privileged access to ISP infrastructure and thus must treat ISPs as black boxes. This way, our approach can work across a wide range of ISPs. However, this also precludes us from benefitting from state-of-the-art telemetry systems like Everflow [38], which can accurately characterize a network's behavior and performance, partly because they benefit from measurements collected directly at the network's devices.

A key challenge we address in this work is that traffic differentiation creates an adversarial environment for network tomography. Informally, tomography relies on the fact that traffic flows crossing a common network bottleneck experience similar performance; it works well when it is able to capture this similarity with mathematical equations. However, we found that when a network bottleneck is the result of traffic differentiation, it can lead to significant short-term differences in loss rates that make it hard to capture performance similarity and cause traditional tomography to fail.

WeHe$\mathbb{Y}$ differs from classic network tomography in an important way: it accurately detects the performance correlation between flows that cross a common bottleneck, while being insensitive to the absolute loss rates experienced by these flows. The reason it can achieve this feature—whereas classic tomography cannot—is that it does not seek to accurately characterize the behavior or performance of links; rather, it seeks to accurately determine whether a particular link sequence is a bottleneck for some traffic flows but not others.

We tested WeHe$\mathbb{Y}$ on 5 real cellular networks that apply traffic differentiation; for 4 of them, WeHe$\mathbb{Y}$ managed to localize differentiation to the target network in more than 89% of our tests. We further studied WeHe$\mathbb{Y}$'s performance and limits via simulations and a wide-area testbed with an emulated rate-limiter. These experiments showed that it is robust to a wide range of network conditions and incurs a false-positive rate of around 5%, even in extremely adversarial scenarios. WeHe$\mathbb{Y}$'s source code is publicly available at https://nal-epfl.github.io/WeHeY.

2 BACKGROUND

2.1 WeHe and Traffic Differentiation

WeHe. [23] tests whether a network provider provides differential performance for a supported set of services (e.g., Skype, WhatsApp, Webex, Netflix, Zoom, etc.). It consists of WeHe clients, installed on devices of participating users, and WeHe servers, running on globally distributed infrastructure (EC2 and M-Lab [21]). When the user wishes to test for differentiation against a given service (e.g., Netflix), the client contacts a WeHe server and initiates a test: the client and server first replay an original copy of a prerecorded network trace for the service, then a modified (bit-inverted) version of that trace, which keeps the same packet sizes and inter-arrival times, but destroys any original bit patterns in the payload (we call these "bit-inverted" traces). The bit-inverted traces remove any packet payload patterns that a DPI-based differentiation device might use to identify traffic for differentiation, and thus, these traces serve as WeHe's "control" measurements. The client then compares the end-to-end performance achieved by the two traces: It divides time (replay duration) into 100 intervals and computes the throughput achieved by each trace in each interval; for each trace, it builds the CDF of the throughput values. Then, it compares the two CDFs with a Kolmogorov-Smirnov test: if they are significantly different, it outputs that there was traffic differentiation.

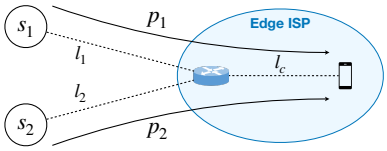


Figure 1: Example topology.

Differentiation Criterion. Li et al. [23] found that differentiation is typically triggered by the presence of a (partial) domain name for a targeted service, as determined from the unencrypted SNI TLS header (or other plaintext fields in the TLS handshake). We note that differentiation based on the server IP address is less common, as CDN-hosted services can be located on a variety of IPs owned by the CDN(s). This is why it is plausible that an ISP that differentiates against traffic from certain services will also differentiate against the corresponding original traces replayed by WeHe servers (because the original traces preserve the SNI).

Differentiation and Loss. As far as we know, traffic differentiation is typically implemented through rate-limiting and, in particular, policing or shaping [13]. In general, a *rate-limiter* takes as input a packet stream and uses a token-bucket algorithm to decide which packets to allow through. It is characterized by a *rate*, which determines the rate at which the token bucket is replenished; and a *burst size*, which determines the bucket's size. When a packet arrives and the bucket is empty, the rate-limiter may drop it, in which case it acts as a *policer*; or put it in a queue, in which case it acts as a *shaper*. Hence, policing and (to a lesser extent) shaping typically affect the loss experienced by the input packet stream.

Detection, but not Localization. WeHe detects the existence of traffic differentiation on an end-to-end path, and reports statistics on which ISPs the users were connected to when differentiation was detected. However, it fundamentally cannot localize the differentiation to any particular part of the path—in principle, it may have happened anywhere between the WeHe server and the WeHe client. For this, we turn to network tomography.

2.2 Network Performance Tomography

Network-performance tomography (from now on just "tomography") refers to a family of algorithms that take as input a network topology and the performance of end-to-end paths, and they estimate the performance of individual links or link sequences. The performance metric may be average loss rate or latency [5, 6, 31, 33, 37], binary loss status (whether a path or link is "lossy" or not) [9, 12, 30], the probability of being "lossy" [14], or neutrality (whether each path or link differentiates against a certain service or not) [45]. A path or link sequence is defined as "lossy" during a time interval when its loss rate during that interval exceeds a given "loss threshold."

Tomographic System of Equations. The common element of tomography algorithms is that they use the network topology to build a system of equations where: each known entity represents the performance of a path or set of paths, and each unknown represents the performance of a link sequence. E.g., consider the topology in Figure 1, where there are two paths, $\{p_1,p_2\}$, and three link sequences, $\{l_1,l_2,l_c\}$, with l_c the only common link sequence between the two paths. Suppose:

- x_k is the probability of link sequence l_k being non-lossy;
- y_i is the probability of path p_i being non-lossy;
- $y_{1,2}$ is the probability of both paths being non-lossy at the same time.

Assuming independent link sequences, we can write [14, 30]

$$\begin{aligned} y_1 &= x_c \cdot x_1 \\ y_2 &= x_c \cdot x_2 \\ y_{1,2} &= x_c \cdot x_1 \cdot x_2 \end{aligned} \tag{1}$$

The first two equations say that each path is non-lossy at a given moment if and only if the two link sequences it comprises are non-lossy at that moment. The last equation says that the two paths are both non-lossy at a given moment if and only if the three link sequences they comprise are non-lossy at that moment.

Identifiability and Rank. Informally, we say that a link sequence is "identifiable" when it is possible to create a tomographic system of equations that enables us to characterize this link sequence. For a link sequence to be identifiable, it is necessary (though not sufficient) that there exists a set of paths that intersect exactly at that link sequence. E.g., in Figure 1, link sequence l_c may be identifiable because paths p_1 and p_2 intersect exactly at l_c. Conversely, none of the links within l_c may be identifiable because, from the vantage points that we have, we cannot distinguish the individual impact of any one of these links on our measurements.

Identifiability is closely related to the rank of the tomographic system of equations: if the system is full-rank, then every link sequence is identifiable as long as there exists a set of paths that intersect exactly at that link sequence. In principle, full-rank is preferable because it yields a unique solution [14, 30, 31]; a rank-deficient system admits multiple solutions, and one needs additional criteria to pick one, e.g., one can favor the solution that includes the smallest number of congested links [9, 12, 33, 37], or the one that is most consistent with link history [30].

Path Performance Correlation. To create a full-rank system of equations, it is not enough to consider the performance of individual paths; one must consider the performance *correlation* among multiple paths. E.g., in the above example, if we consider only the first two equations (which describe the performance of individual paths), the system is trivially rank-deficient, as we have three unknowns [2] (x_c, x_1, and x_2). Once we add the last equation (which captures the performance correlation between the two paths), the system becomes full-rank, and we can solve it to obtain the performance of all the link sequences. Creating full-rank systems may not be an option, e.g., when tomography is used on top of passive measurements.

Typical Setting. In the typical tomography setting, one has access to a given set of vantage points in a network, and the goal is to characterize the performance of as many links in the network as possible, potentially given a measurement budget, e.g., a maximum number of measurement probes. So, a common challenge is picking the vantage points/paths that yield the best results [15, 34, 35]. More generally, tomography research typically focuses on scalability, e.g., minimizing the inference error and/or the required measurements, as the size of the network topology increases.

3 OUR SYSTEM

3.1 Overview and Rationale

In this work, we consider a path p_0, between a server and a client, that is known to differentiate between an original and a bit-inverted trace[3]. Given this, WeHe$\mathbb{Y}$ looks for evidence that the differentiation happens within a target network area that contains the client. In this paper, the target network area is always the client's ISP, but it could be any area around the client.

Our setting differs from the typical tomography setting described at the end of Section 2.2 in that (a) our goal is to characterize the behavior of one particular network area, while (b) our vantage points are one particular client and a set of servers that can communicate with the client. In principle, we could involve other clients in order to create a richer set of vantage points and paths and emulate the typical tomography setting. However, that would require clients that run measurements at particular points in time, e.g., through pop-up messages or remote control, and we believe either approach would discourage participation in differentiation measurements.

When WeHe$\mathbb{Y}$ is invoked, it performs four operations:

(1) **Topology construction** (§3.3): It identifies a topology like the one in Figure 1: two paths, p_1 and p_2, that start at different servers, end at the client, converge exactly once, and the convergence happens within the target network area. One of the two paths may coincide with p_0, but it is not necessary. The rationale is to construct the simplest topology that enables us to apply tomography to reason about the target network area. More specifically, the topology we construct must contain a link sequence that (a) is fully contained within the target network area and (b) is identifiable (§2.2). Requirement (b) makes it possible to reason about the behavior of this link sequence, while requirement (a) ensures that if this link sequence differentiates, the differentiation happens within the target network area. The simplest topology that meets these requirements is one with two paths that converge within the target network area. In principle,

[2] Adding equations by considering more individual paths does not work. If we add one equation per path, the rank is always smaller than the number of links, except for the trivial case of a network with no switches or routers [30].

[3] E.g., after a standard WeHe test determines this to be the case.

WeHe$\mathbb{Y}$ could use more paths; however, we want to minimize the amount of work the client has to do, including the data consumption from measurements (given that users may have metered data plans).

(2) **Simultaneous replays** (§3.4): WeHe$\mathbb{Y}$ replays a modified version of the original trace on the two paths simultaneously; we call this the "original simultaneous replay". Then, it does the same with a modified version of the bit-inverted trace; we call this the "bit-inverted simultaneous replay". During these simultaneous replays, it performs throughput, packet-loss, latency, and topology (traceroute) measurements along each path. The rationale is to perform measurements that enable us to write a full-rank system of equations like System 1. Hence, our measurements must capture the performance correlation between paths p_1 and p_2 (§2.2), and the most efficient way to do this is through a simultaneous replay.

(3) **Differentiation confirmation**: WeHe$\mathbb{Y}$ re-uses WeHe's algorithm (based on throughput differences) to determine whether each of the paths p_1 and p_2 differentiated between the original and bit-inverted traces. Unless both of them did, WeHe$\mathbb{Y}$ outputs that it did not find evidence of traffic differentiation within the target network area.

(4) **Common bottleneck detection** (§4): WeHe$\mathbb{Y}$ considers measurements collected along p_0, p_1 and p_2, and it determines whether p_1 and p_2 shared a common bottleneck. If yes, then the differentiation must have happened within the target network area—since the two paths may share a bottleneck only within that area— and WeHe$\mathbb{Y}$ outputs that it did find evidence of traffic differentiation within the target area. In any other scenario, WeHe$\mathbb{Y}$ outputs that it did not find such evidence.

Common bottleneck detection consists of two algorithms: (a) First, it compares the throughput achieved along p_0 during the original replay, against the aggregate throughput achieved along p_1 and p_2 during the original simultaneous replay. If these two quantities are approximately the same, it outputs that it detected a common bottleneck. (b) Otherwise, it examines the loss patterns experienced along p_1 and p_2 during the original simultaneous replay. If the two paths experienced significantly correlated loss trends, it outputs that it detected a common bottleneck. Otherwise, it outputs that it did not.

The rationale behind these two separate algorithms is to distinguish between the following scenarios: (a) The client's ISP implements traffic differentiation via per-client throttling, e.g., per-client policers. In this scenario, p_0 traverses the per-client bottleneck alone, then p_1 and p_2 share the per-client bottleneck only with each other. Hence, this scenario can be detected through the simple throughput comparison of the first algorithm. (b) The client's ISP implements traffic differentiation via collective throttling, e.g., a collective policer for all traffic from a given service or application. In this scenario, p_0, and then p_1 and p_2 share the collective bottleneck with other flows, of an unknown number and rate. This sharing has an unknown impact on the throughput achieved by WeHe$\mathbb{Y}$'s replays, hence requires a more sophisticated algorithm.

We implemented WeHe$\mathbb{Y}$ as part of WeHe, which is hosted by the M-Lab measurement platform [20]. In particular, we created a new "topology construction" module that performs operation (1), extended the WeHe client and server to perform operation (2), and extended the WeHe server to perform operations (3) and (4).

3.2 Main Limitations

Our approach can only localize traffic differentiation that involves a common bottleneck and causes packet loss. This is the case when differentiation is implemented through per-client throttling or collective per-service/per-application throttling; and the throttling relies on policers or shallow shapers. Conversely, our approach cannot localize (in its current form) traffic differentiation that is implemented through per-TCP-flow and/or per-UDP-flow throttling, or relies on deep shapers that avoid packet loss.

These limitations do not impact WeHe's functionality in any way: if WeHe detects traffic differentiation that involves a common bottleneck and causes packet loss, WeHe$\mathbb{Y}$ will find evidence that the differentiation happened inside the client's ISP; otherwise, WeHe$\mathbb{Y}$ will output that it did not find any evidence, i.e., it cannot provide any additional information relative to WeHe. We discuss how to address these limitations in Section 7.

3.3 Topology Construction

The topology-construction module (TC) periodically ingests and analyzes data from M-Lab's traceroute database [28]. This happens as frequently as the latter is updated, which is currently once a day.

In particular, TC's input consists of two BigQuery tables from M-Lab's dataset: the first one contains traceroute records collected using scamper [24], while the second one contains additional geolocation and Autonomous System Number (ASN) information on each hop, collected from the MaxMind, IPinfo.io, and RouteViews databases. Merging these two tables yields a set of traceroute records annotated with the additional per-hop information. From this set, TC discards all traceroutes that do not meet the following two conditions: (a) the last reported hop has the same ASN as the destination, and (b) two subsequent links always meet at the same IP address. The first condition may not hold, e.g., because an ISP may block ICMP packets close to the client. The second condition may not hold due to IP aliasing. We could reduce the number of discarded traceroutes by leveraging IP alias resolution techniques as in [19], but we have not implemented this yet.

After filtering the input, TC performs four steps for each traceroute destination d found in the retained traceroute records:

(1) It finds all traceroutes destined to d. If none exists, it finds all traceroutes with destinations with the same ASN as d.
(2) For each traceroute found in step (1), it identifies all hops with the same ASN as d. These are *candidate* intermediate nodes, i.e., nodes located in the same ISP as d, where the two paths of the final topology could potentially converge.
(3) It considers all pair combinations of the traceroutes found in step (1). For each pair, it checks if: (a) the two traceroutes have at least one candidate intermediate node in common, and (b) they have no common node located outside d's ISP. To determine that two traceroutes have a "node in common," TC directly compares the IP addresses of their respective hops (so, again, TC does not benefit from IP alias resolution).
(4) For each traceroute pair that passes the check, it computes a {traceroute destination, server pair} tuple and stores it in a topology database.

TC's output is a table containing for each destination d: d's IP prefix (/24 for IPv4 and /48 for IPv6) and ASN; and a list of potential M-Lab server pairs that form a suitable topology with d.

A key question is: given a client, how likely is it that a suitable topology that involves that client exists? To get a sense, we pulled from M-Lab all the traceroute measurements collected through WeHe experiments in April 2023. We ran our topology-construction algorithm with one month's worth of traceroute data and for different weeks of the month. On average, there was at least one complete traceroute for 52% of WeHe clients, and at least one suitable topology for 74% of these clients. These numbers give us a rough *lower* bound on the number of suitable topologies for the following reason: Currently, when M-Lab suggests a server to a WeHe client, it favors the servers that are closest to the client, which is not favorable for constructing suitable topologies. We expect to improve the number of suitable topologies, e.g., by explicitly adding traceroute measurements from a more diverse set of M-Lab servers to each WeHe client.

3.4 Simultaneous Replay

When traffic differentiation is detected on the path between a server s_0 and a client, the client asks the user if they wish to perform additional measurements to localize the differentiation. If the user answers yes:

(1) The client queries the topology database (§3.3) for the IP addresses of two servers, s_1 and s_2.
(2) The client asks both s_1 and s_2 to replay a modified version of the original trace simultaneously, then do the same for the bit-inverted trace.
(3) During these two simultaneous replays, the servers and client collect throughput, packet-lose, and latency measurements. At the end of each replay, the corresponding server performs a traceroute to the client. At the end of both replays, all measurements are gathered at one of the servers.
(4) The server that gathers the measurements verifies that the topology (the paths from servers s_1 and s_2 to the client) was still suitable at the end of the replays. If not, it discards the measurements and updates the topology database. Otherwise, it proceeds with operations (3) and (4) from Section 3.1.

Synchronization. Ideally, s_1 and s_2 should start replaying each trace at exactly the same time, to maximize the chance that their traffic experiences the same network conditions on the common link sequence. However, given that we cannot, in general, control when exactly each packet reaches the common link sequence, the client simply tells the two servers to start via two commands sent back-to-back.

UDP Replay: Poisson. When a server replays an original UDP trace, it maintains the original packet sizes, content, and average transmission rate, but modifies the packet-transmission times such that they follow a Poisson process [4]. This modification enables WeHe$\mathbb{Y}$ to benefit from the PASTA property: a sequence of measurement probes whose transmission times follow a Poisson process can asymptotically "see" the true loss rate of the underlying bottleneck [2, 42]. Put another way, without this modification, the loss rates computed by WeHe$\mathbb{Y}$ (Alg. 1, line 7) might not be unbiased estimates of the true underlying average path loss rates, while its correlation test (line 10) would incorporate some undefined level of measurement bias [5].

TCP Replay: Pacing. When a server replays an original TCP trace, it maintains the original packet sizes and content, but: (a) lets TCP congestion control and TCP pacing dictate the packet-transmission times, and (b) potentially extends the trace duration by replaying it again (more about this below).

TCP pacing serves the same purpose as Poisson transmission times: When a path carries bursty traffic, it may lose packets in bursts, causing packets transmitted close to each other to experience correlated loss; the burstier the loss, the higher its correlation and the less the path's average loss rate approximates the average loss rate of the underlying bottleneck [2]. Pacing reduces burstiness by guaranteeing a minimum distance between the packets; in a sense, it enables the packets to "jump over" correlation-inducing bursts, provided that the packet rate is large relative to the correlation scale. The reason for not simply using Poisson transmission times is that it is hard to do so while complying with a congestion-control scheme.

Extending a trace may be necessary, because the original trace may be too short and yield too few loss measurements for drawing a reliable conclusion. In that case, the server repeats the trace until the replay reaches a duration of at least 45 sec.

Packet-loss Measurements. During each replay, WeHe$\mathbb{Y}$ tracks packet loss. Who does this depends on the transport layer: for UDP traces, it is the client; for TCP traces, it is the server. In the TCP case, the client is typically unable to track packet loss because most clients run on mobile devices with standard OSes, which offers limited access to the transport layer. The server estimates packet loss based on retransmissions, using existing tools [3]. We think that this approach is sufficiently accurate for WeHe$\mathbb{Y}$ and discuss its limitations in Section 7.

4 COMMON BOTTLENECK DETECTION

We first describe our two detection algorithms (§4.1 and §4.2), then share insight on how we arrived at the second algorithm, which is the more sophisticated of the two (§4.3).

4.1 Throughput Comparison

The first detection algorithm takes as input throughput measurements, and it outputs whether p_1 and p_2 shared a common bottleneck. In particular, the input consists of:

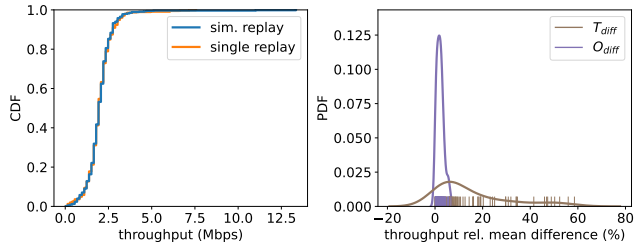
- The set $X \equiv \{X_1, X_2, \ldots X_n\}$, where X_i is the i-th throughput sample collected during the original-trace WeHe replay along p_0.
- The set $Y \equiv \{Y_1, Y_2, \ldots Y_m\}$, where Y_j is the sum of the j-th throughput samples collected along p_1 and p_2 during the original-trace simultaneous replay (§3.4).

In summary, the algorithm checks whether the aggregate throughput measured along p_1 and p_2 roughly adds up to the throughput measured along p_0. This should be the case if, during all the replays, the client's traffic traverses a queue that is dedicated to that client *and* is the bottleneck.

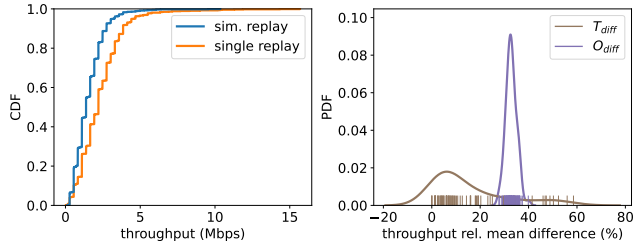
The algorithm relies on two distributions:

[4] Li et al. found that differentiation is typically triggered by packet content, and thus, changing transmission times should not affect the observed differentiation [23].

[5] It would also incorporate statistical errors, but these are bounded or can even be mitigated using jackknife or bootstrap methods, similarly to [23, 43].



(a) Per-client throttling scenario. The two CDFs (left) and the peak of the two PDFs (right) overlaps signifcantly.



(b) An alternative scenario. The two CDFs (left) and the two PDFs (right) do not overlap at all.

Figure 2: CDF of single and sim. replay throughputs (left) and PDF with rug plot of O_{diff} and T_{diff} (right).

(1) T_{diff} is an empirical distribution that represents normal throughput variation. It is computed from past WeHe tests [6] as follows: We look for test pairs that were performed less than 10 minutes apart and belong to the same client, application, and carrier. For each such test pair, we compute

$$t_{diff} = \frac{\overline{T_1} - \overline{T_2}}{\max(\overline{T_1}, \overline{T_2})},$$

where $\overline{T_1}$ and $\overline{T_2}$ are the throughput means measured, respectively, in the two tests during the bit-inverted replay. The set of these values (from all the chosen test pairs) forms T_{diff}.

(2) O_{diff} is an empirical normal distribution that represents the difference between X and Y. It is computed using standard Monte-Carlo simulation (Ch. 6 in [22]). In more detail: We run multiple iterations. In each iteration, we create two sets X' and Y', each one including a randomly chosen half of the samples from X and Y, respectively. Then, we compute their relative mean difference

$$o_{diff} = \frac{\overline{X'} - \overline{Y'}}{\max(\overline{X'}, \overline{Y'})},$$

where $\overline{X'}$ (resp. $\overline{Y'}$) is X''s (Y''s) mean value. The set of these values (resulting from all the iterations) forms O_{diff}. The number of iterations is the number of data points in T_{diff}, such that O_{diff} and T_{diff} have the same size.

If O_{diff} is significantly smaller than T_{diff}, that indicates that the difference between X and Y is justifiable as normal throughput variation. In this case, the algorithm outputs that p_1 and p_2 shared a common bottleneck. Otherwise, the algorithm outputs that it did not find evidence of this.

[6] https://wehe-data.ccs.neu.edu/

Figure 2 shows representative examples of these distributions: Figure 2a corresponds to the scenario of per-client throttling, where p_1 and p_2 traverse the same bottleneck queue, which is dedicated to that client; this is the scenario that this algorithm aims to detect. Figure 2b corresponds to an alternative scenario, where p_1 and p_2 share a bottleneck queue with other traffic. On the left, we show the cumulative distribution functions (CDFs) of X and Y. On the right, the probability density functions (PDFs) and rug plots [7] of O_{diff} and T_{diff}. O_{diff} has significantly lower variance (narrower PDF) than T_{diff}, because the former represents throughput achieved during different intervals of the same test, whereas the latter represents throughput achieved during different past tests.

We observe the following: First, X and Y overlap significantly more in the per-client throttling scenario (Figure 2a, left) than in the alternative (Figure 2b, left). Second, O_{diff}'s and T_{diff}'s peaks overlap significantly in the former (Figure 2a, right), but not at all in the latter (Figure 2b, right). Differently said, in the per-client throttling scenario, the difference between X and Y (captured by O_{diff}) is small enough that it can be conservatively justified as normal throughput variance (captured by T_{diff}).

The comparison between O_{diff} and T_{diff} is done with the Mann-Whiteny U (MWU) test, also known as Wilcoxon Rank Sum test (Ch. 4 in [22]), with the following alternative hypothesis: O_{diff} has significantly smaller rank-sum than T_{diff}. The algorithm outputs that it detected a common bottleneck if the p-value of the MWU test is less than 0.05; otherwise, it outputs that it did not find evidence of a common bottleneck. For example, in the per-client throttling scenario shown in Figure 2a, the p-value is $7.54e-18 < 0.05$, while in the alternative shown in Figure 2b, it is $0.99 > 0.05$. Two more common alternatives than MWU are the T-Test and the Kolmogorov-Smirnov test; we do not use the former because it requires assumptions about the distribution of the throughput samples, and we do not use the latter because it is less robust to outliers.

4.2 Loss Trend Correlation

The second detection algorithm (Alg. 1) takes as input measurements collected along p_1 and p_2 during the original-trace simultaneous replay, as well as the maximum acceptable false-positive rate FP; and it outputs whether p_1 and p_2 shared a common bottleneck. In summary, the algorithm checks whether the loss rates of the two paths follow a similar trend over time. This should be the case if, during the simultaneous replay, traffic from the two paths traversed a common bottleneck *and* constituted a small fraction of the overall traffic traversing that bottleneck.

One element that shaped the design of Alg. 1 is that the input measurements are noisy. This comes from the fact that, for TCP traffic, we can only measure packet loss on the server side, and we do so by interpreting TCP retransmissions as loss events (§3.4). This introduces two types of error: First, loss events may be overcounted. Second, each loss event is registered at a different time than when it actually occurred (at the moment of the resulting TCP timeout or duplicate ACKs). The latter introduces a certain amount of "desynchronization" in the measurements: suppose a packet from p_1 and a packet from p_2 are dropped at the same network queue due to the same overflow event; these two loss events may be registered at

[7] A rug plot helps project the location of raw data points onto a particular axis.

Algorithm 1 LossTrendCorrelation

Input: Packet-loss measurements M
Acceptable false-positive rate FP
Output: True or False

1: $correlations \leftarrow 0$
2: $\Sigma \leftarrow$ Create set of interval sizes s.t. $\forall \sigma \in \Sigma$, $10 \leq \frac{\sigma}{\max_i\{p_i - \min - RTT\}} \leq 50$
3: **for** $\sigma \in \Sigma$ **do**
4: $(LostPkts_1, TxedPkts_1, LostPkts_2, TxedPkts_2) \leftarrow$ Create time series from M, σ
5: **for** $t \in 0..T-1$ **do**
6: **for** $i \in 1,2$ **do**
7: $LossRate_i[t] \leftarrow \frac{LostPkts_i[t]}{TxedPkts_i[t]}$
8: **end for**
9: **end for**
10: **if** $Spearman-p-value(LossRate_1, LossRate_2) < FP$ **then**
11: $correlations \leftarrow correlations+1$
12: **end if**
13: **end for**
14: **return** $correlations > (1-FP)|\Sigma|$

different times, depending on the round-trip times (RTTs) and TCP timeouts of the two paths.

At a high level, Alg. 1 takes two steps to mitigate the effects of measurement noise: (a) It tracks each path's loss rate as an average over a sufficiently large set of packets that span multiple RTTs. (b) To check for trend similarity, it relies on Spearman's correlation coefficient, which does not depend much on the absolute values of the average loss rates but rather on their rank in the data set.

In more detail, Alg. 1 works as follows: It iterates over different interval sizes (line 3). For each size σ, it divides time into intervals of size σ; counts, per interval and per path, the number of packets transmitted and lost; and discards intervals where one or both paths did not transmit a minimum number of packets (10 in our implementation) or none of the two paths lost any packets (line 4). Then, it computes the loss rate of each path during each interval (lines 5-9). Next, it computes the p-value of Spearman's correlation coefficient over the two resulting time series, under the null hypothesis that the two series are *not* correlated, and checks whether the p-value indicates correlation with false-positive rate FP (line 10). Finally, it outputs that it detected a common bottleneck if the latter check is true for at least a fraction $1-FP$ of the interval sizes (line 14). In our prototype and experiments, we set $FP=0.05$.

The first key element of Alg. 1 is that it computes a *loss-rate time series* for each path and analyzes the Spearman correlation of the two loss-rate time series. We explain the rationale: When a flow crosses a network bottleneck, its loss rate naturally follows the rate at which traffic arrives at the bottleneck; the higher the arrival rate, the more traffic the flow has to contend with (for the bottleneck's bandwidth), and hence the higher its loss rate. Thus, when two flows cross a common bottleneck, their loss rates follow similar patterns over time. This does not mean that the loss rates of the two flows are always the same or even close to each other (as we discovered the hard way), only that they tend to increase and decrease together. Alg. 1 captures this correlation through hypothesis testing: the null hypothesis is that the loss-rate time series of the two paths are *not* correlated. We use the Spearman correlation coefficient because it is normalized (it captures trend, not absolute-value similarity) and, out of all correlation metrics, it is considered the least sensitive to strong outliers (because it limits the outlier to the value of its rank). The p-value of such a correlation test says how likely it is that the Spearman correlation between the two time series was due to chance. We reject the null hypothesis iff this likelihood is below our acceptable false-positive rate FP.

The second key element is that the interval size (the step) of the time series ranges from 10 to 50 RTTs. Choosing the interval size involves the following trade-off: On the one hand, the shorter the interval, the higher the risk of a false negative (not detecting an existing common bottleneck). E.g., suppose we set the interval size to a few milliseconds; suppose the two paths lose packets to the same overflow events, but p_1 registers each loss tens of milliseconds after p_2 (so, in a different time interval) due to the desynchronization mentioned above; as a result, the loss-rate time series of the two paths may not be significantly correlated, even if the losses were due to a common bottleneck. On the other hand, the larger the interval, the higher the risk of a false positive (detecting a common bottleneck when there is none). E.g., consider a set of measurements that last 30sec, and an interval size of 15sec (resulting in two intervals); if, for both paths, the loss rate happens to increase from the first to the second interval, the Spearman-correlation analysis may indicate correlation, yet that would hardly be evidence of a common bottleneck. So, the interval size should be at least multiple RTTs long, in order to mask the desynchronization resulting from different RTTs and TCP dynamics, and avoid false negatives; but not longer than necessary in order to reduce false positives.

The third key element is that the algorithm iterates over multiple plausible interval sizes. We explain the rationale: Ideally, the Spearman-correlation analysis would guarantee the target false-positive rate FP, independently from the interval size. In practice, this is not the case: As mentioned earlier, even if the two loss-rate time series are correlated, that does not necessarily indicate a common bottleneck (e.g., if the interval size is too coarse and yields too-short time series). Moreover, the analysis guarantees the target false-positive rate when the null hypothesis is that the correlation coefficient is *exactly* 0. Differently said, if the correlation coefficient of the two time series is not exaclty 0, the analysis may yield additional false positives. By iterating over multiple interval sizes, and requiring that the null hypothesis can be rejected for at least some minimum fraction of these, we make the algorithm more conservative toward yielding a false positive. We empirically observed that setting this minimum fraction to $1-FP$ enables us to never exceed the target false-positive rate FP, at the expense of a moderate increase in false negatives.

4.3 Design Evolution

Our loss-trend correlation algorithm is simple in retrospect, but concluding that it is the right algorithm for our task is the result of multiple iterations. In this subsection, we share our journey, which involves understanding the nature of the connection between tomography and performance correlation.

Revisiting Tomography Assumptions. Considering Figure 1, suppose both paths occasionally experience significant loss, and we want hard evidence that the problem lies in the common link sequence,

l_c. If we consider only the performance of each individual path, we cannot get such evidence: the problem may lie either in l_c, or in the two non-common link sequences, $\{l_1, l_2\}$, or even in all three of them. However, intuitively, if we consider the performance correlation between the two paths, we can: if there is no correlation, then the problem cannot lie in l_c; if there is *significant* correlation, that constitutes evidence that the problem lies in l_c—under the mild assumption that l_1 and l_2 are independent from each other.

In an attempt to capture this correlation, traditional tomography estimates the joint performance distribution of the two paths (via the 3rd equation in System 1); this *could* yield a unique solution, but it requires further and stronger assumptions to hold. For example, [30] assumes that packet loss is identically distributed for all flows over time: if two packets from different flows arrive at a common bottleneck at the same time, they are lost with equal probability, whi is also constant over time. If loss events follow that pattern, then System 1 captures correlation well.

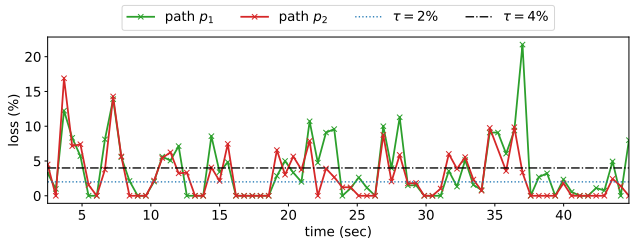
In practice, however, such a loss model may not capture how flows experience a common bottleneck. For instance, one flow may experience more significant loss than another due to the same overflow/throttling event, because it happened to send significantly more bursty traffic during the event. The bigger this difference (between how different flows experience a common bottleneck), the harder it becomes for System 1 to capture the resulting correlation. More formally, whereas System 1 requires a simple loss model (e.g., that packets from paths p_1 and p_2 are lost at the common bottleneck with equal probability), the packets may actually be dropped with probabilities that depend on their arrival times.

V0: Binary Loss Tomography. Our first idea was to use tomography out-of-the-box: invoke the most appropriate state-of-the-art tomography algorithm to infer the performance of each link sequence in Figure 1; if the common link sequence l_c had worse performance than the other two, this means that the two paths had a common bottleneck (l_c).

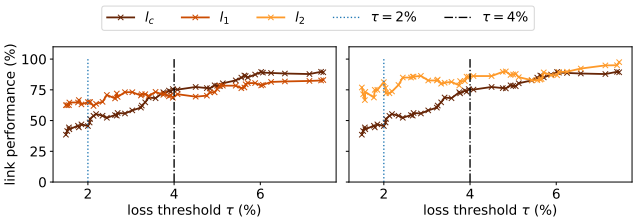
We started from the state-of-the-art tomography algorithm for constructing a full-rank system of equations [14] (BinLossTomo, Alg. 2, Appendix, §B). In addition to packet-loss measurements, this algorithm takes as input an interval size σ and a "loss threshold" τ. It divides time into fixed-size intervals of size σ (line 1). For each interval and each path p_i, it checks whether p_i's loss rate exceeded the loss threshold τ during the interval; if yes (resp., no), it labels p_i as "lossy" (resp., "not lossy") during that interval (lines 2-6). It computes path performance y_i as the fraction of intervals in which p_i was "not lossy" (line 7); it computes joint path performance y_{12} as the fraction of intervals in which both paths were "not lossy" (line 8). Finally, it solves System 1 to infer the performance of each link sequence (the probability of being "not lossy") and outputs the solution (x_c,x_1,x_2) (line 9).

Our first attempt (BinLossTomo++, Alg. 3, Appendix, §B) simply invokes the above algorithm (line 1) and detects a common bottleneck if the common link sequence has worse performance than both non-common link sequences (line 2).

The Parameter-Sensitivity Problem. The challenge we faced was the sensitivity of binary tomography to the loss threshold (which determines how high of a loss rate makes a path "lossy") and the interval size (which determines the time granularity at which loss rates are computed). Fundamental tomography work does not focus on these



(a) Loss rates of p_1 and p_2 over time.



(b) Inferred performance of p_1's links (left) and p_2's links (right).

Figure 3: Example with rate-limiter on the common link (30sec measurement duration and σ=0.6sec).

parameters because they do not affect the nature of the underlying mathematical problem. In our context, however, the sensitivity to these parameters made it impossible to draw reliable conclusions.

One way in which BinLossTomo fails is when two paths share a common bottleneck, yet measurement noise and/or TCP dynamics cause their loss rates (at some granularity) to diverge. Fiddling with the loss threshold and interval size may suppress some of these divergences and prevent them from causing a false negative. However, without ground truth, we run the risk of picking parameters that bias the algorithm in favor of detecting a common bottleneck.

Another, more subtle way in which BinLossTomo fails is when the loss rates of two paths are similar, yet they fall on opposite sides of the loss threshold. In general, binary tomography works well when path/link loss rates fall naturally into two classes, and there is a significant gap between the two [45]. E.g., suppose a link's loss rate can be either below 0.001 ("non-lossy") or above 0.05 ("lossy"); if τ is set in the middle between these two values, BinLossTomo infers link performance correctly. However, when loss rates are not bimodal in this way, it is easy to pick a loss threshold that makes BinLossTomo fail.

We illustrate with an experiment: We instantiate the topology of Figure 1, where paths p_1 and p_2 carry a long-running TCP flow each; this traffic, together with other background traffic, goes through a rate-limiter located in the common link sequence l_c, which introduces an average loss rate 0.04 and constitutes the sole cause of packet loss. Figure 3a shows the loss rates of the two paths over time, as measured end-to-end; each curve corresponds to a different path. Figure 3b (left) shows the performance of link sequences l_c and l_1, as inferred by BinLossTomo, for different loss thresholds τ.

We focus on Figure 3b (left). The y-axis represents inferred link performance (the probability of being "not lossy"), while the x-axis represents the loss threshold τ. The dark curve shows l_c's performance (x_c), while the light curve shows l_1's performance (x_1).

If BinLossTomo inferred link performance correctly, we would see the following: (a) l_1's performance (the light curve) would be

a straight line at 100%. This is because, in our experiment, l_1 has 0 packet loss (probability 100% of being "not lossy"). (b) l_c's performance (the dark curve) would monotonically increase with the loss threshold τ. This is because the higher the loss threshold, the less frequently it is exceeded, and the less frequent paths and links are "lossy."

We do not see this behavior, however, for two reasons: First, despite sharing a common bottleneck, the loss rates of the two paths occasionally diverge enough to fall on opposite sides of the loss threshold. As a result, BinLossTomo determines that the two paths occasionally experience loss due to different bottlenecks, and it mistakenly attributes some of p_1's packet loss to l_1. Second, as the loss threshold approaches $\tau = 0.04$, BinLossTomo fails completely: given that l_c's true average loss rate is 0.04, the loss rates of the two paths oscillate around this value, e.g., 0.039, 0.041, etc; hence, their loss rates frequently fall on opposite sides of 0.04 (as frequently as in half the intervals). As a result, BinLossTomo decides that they frequently have different loss statuses, and it mistakenly attributes a significant part of p_1's packet loss to l_1. This is why there exists a range of loss thresholds where l_c and l_1 appear to have similar performance (the dark and light curves in Figure 3b (left) are close and even cross).

V1: Tomography without Parameters. The first way we tried to eliminate parameter sensitivity was by eliminating the parameters: *not* pick a single loss threshold and time interval, but sweep the entire range of reasonable parameters and check if most combinations led to the correct conclusion.

Hence, our next attempt (BinLossTomoNoParams, Alg. 4, Appendix, §B) defines a sequence of interval sizes $\mathcal{S}$ and loss thresholds $\mathcal{T}$ (lines 1,2). The interval sizes range from 10 RTT to 50 RTT. The loss thresholds are chosen such that $0.1 \leq y_i \leq 0.9$ (where y_i is the path performance computed in V0, line 7), i.e., none of the paths is found "lossy" too often or too rarely. For each interval size and each loss threshold, Alg. 4 invokes BinLossTomo (line 5) and keeps track of the gap between the performance of the common link sequence, x_c, and each of the non-common link sequences, x_1 and x_2 (lines 6,7). Finally, it computes the two average gaps across all loss thresholds and interval sizes, and it detects a common bottleneck if both of them are positive (line 9); that is, if the common link sequence had *on average, across all parameter combinations* worse performance than both non-common link sequences.

The rationale behind averaging the gaps across parameters is the following: Even if BinLossTomo infers link performance incorrectly, there exists a range of "good" parameter combinations (e.g., $\tau < 0.035$ in Figure 3b (left)), for which BinLossTomo++ draws the correct conclusion. For these "good" parameter combinations, the inferred performance of the common link sequence (x_c) is worse than the inferred performance of the other link sequences (x_1 and x_2), and the gap is significant (even if the values are incorrect in the absolute). Then, there exists a range of "bad" parameter combinations (e.g., $\tau > 0.04$ in Figure 3b (left)) for which BinLossTomo++ keeps making incorrect conclusions; for all of these parameter combinations, the gap is significantly smaller (because the inferred performance of the common and non-common link sequences is similar). Averaging the gap across all the reasonable parameter combinations eliminates the error introduced by the bad ones.

This approach eliminates the false positives and some of the false negatives that result from picking an unfortunate loss threshold or interval size. However, we ran into many scenarios where two paths share a common bottleneck, yet BinLossTomo decides that they are not "lossy" during the same time intervals, and no parameter combination can change this conclusion. This is because, as discussed earlier, when two paths share a common bottleneck, their loss rates tend to increase and decrease together, but they are not necessarily similar. BinLossTomo is fundamentally unable to capture this kind of correlation.

V2: Tomography with Loss Trends. Hence, the second way we tried to eliminate parameter sensitivity was by rethinking the underlying tomography algorithm. We designed an algorithm (not shown) that labels a path "lossy" during an interval when its loss rate increases relative to the previous interval. This modification naturally eliminates the notion of a loss threshold and significantly reduces the sensitivity to the interval size.

WeHe$\mathbb{Y}$: From Tomography to Correlation. Our final loss-trend correlation algorithm is V2 with a simplification: While studying V2's behavior, we realized that we did not actually need a complete tomography algorithm; we only needed to compute the performance correlation of the two paths. A tomography algorithm outputs the performance of each link sequence; V2 needs this output only to check if the common link sequence had worse performance but does not need the precise performance values. Now, V2's underlying tomography algorithm infers that the common link sequence has worse performance iff it determines that the performance of the two paths was correlated. Hence, we only needed to compute the performance correlation of the two paths (similarly to how V2 does it). This is precisely what our final algorithm does: it computes the performance of each path as a time series of loss rates; then, it computes the Spearman correlation coefficient of the two time series.

5 EVALUATION IN THE WILD

We tested WeHe$\mathbb{Y}$'s throughput-comparison algorithm (§4.1) on five U.S. cellular ISPs [8] that apply traffic differentiation as part of their offered plans (e.g., disclosed as "video streaming at DVD quality (480p)" [16]). This type of policy is typically implemented via per-client throttling [23]. We used this as "ground truth" to test the throughput-comparison algorithm.

We were not able to test WeHe$\mathbb{Y}$'s loss-trend correlation algorithm (§4.2) in the wild, because we are not aware of any ISP disclosing that it implements collective per-application/per-service throttling. We tried to craft experiments that would trigger realistic collective-throttling behavior but did not succeed: We replayed concurrently a large number of WeHe traces to the same client (and pretended that each flow belonged to a different client). However, this yielded unrealistically high loss rates, presumably, because the policers used for per-client throttling are not configured to handle a large number of concurrent streams. The loss-trend correlation algorithm was not able to detect the common bottleneck because it does not perform well when the loss rate exceeds 20% (§6.3).

[8] Verizon, T-Mobile, Visible, TracFone, and GoogleFi.

ISP_1	ISP_2	ISP_3	ISP_4	ISP_5
89.8%	89.83%	94%	98.18%	16.28%

Table 1: Successful localization rate of traffic differentiation in five real ISPs.

Setup. We used two WeHe𝕐 servers located in different zones of the Google Cloud Platform (GCP) Iowa ("us-central-a" and "us-central-c") and a WeHe𝕐 mobile client running on a Google Pixel 7 Pro smartphone. Each server ran the WeHe𝕐 server code on an e2-standard virtual machine (VM) with an Ubuntu 20.4 LTS boot image. For each of the five ISPs, we purchased an unlimited prepaid SIM card.

Experiments. We performed around 50 "basic" tests per ISP, using traces from Netflix, YouTube, Disney+, Amazon Prime and Twitch. We ran these tests the same way a WeHe𝕐 user would: used the client to request a standard WeHe test, then clicked "yes" when the client asked if we wanted to run an additional test for localization (because it detected traffic differentiation). If WeHe𝕐's throughput-comparison algorithm worked perfectly, it should detect a common bottleneck in all these tests.

We also performed a set of "sanity check" tests. These did not capture any realistic scenario; their goal was to confirm that throughput comparison works correctly. We repeated the basic experiments; however, during the original simultaneous replay, a third server replayed a third trace to the same client. If WeHe𝕐's throughput-comparison algorithm worked as expected, it should *not* detect a common bottleneck in any of these tests. This is because, in these tests, p_1 and p_2 shared the same bottleneck with an additional path; hence, their aggregate throughput should not add up to that achieved by p_0.

Results. In the basic tests, for four out of the five ISPs, the algorithm detected a common bottleneck at least 89% of the time; for one ISP, it did so only 16% of the time (Table 1).

We have the following hypothesis for why the algorithm performs poorly with this particular ISP: This ISP's throttling policy changes as a function of the traffic rate that the client is receiving. In particular, fixed-rate throttling at 2.5Mbps kicks in after some criterion is met. During the simultaneous replay, this criterion is met faster (but not equally fast across tests), presumably, because two servers are streaming to the client concurrently. As a result, during the original simultaneous replay, fixed-rate throttling starts earlier (but not at an easily predictable moment), causing the throughput-comparison test to fail. Figure 4 shows throughput over time for one of these tests, during the single (blue) and simultaneous (orange) original replay: during the simultaneous replay, throughput drops to 2.5Mbps after 5sec; during the single replay, the same thing happens after 22sec. As a result, the aggregate throughput achieved during the original simultaneous replay does not add up to the throughput achieved during the original single replay, and the algorithm does not detect a common bottleneck. We believe that we could correctly handle such cases by identifying the point in time where fixed-rate throttling starts and applying throughput comparison only thereafter. However, we have not implemented this yet.

In the sanity-check tests, the algorithm only exhibited the wrong behavior (detected a common bottleneck) once.

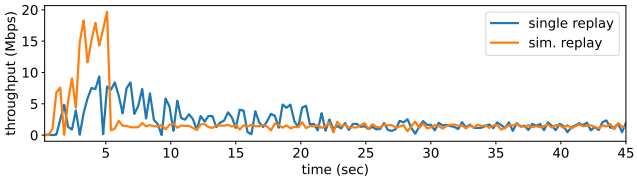


Figure 4: Example of throughput over time achieved during the single and simultaneous original replays to ISP_5's network.

6 EVALUATION VIA EM/SIMULATION

We tested WeHe𝕐's loss-trend correlation algorithm (§4.2) via emulation and simulation. After stating our experimental setup (§6.1), we answer two questions: Does the algorithm work as expected (§6.2)? What are its limits, i.e., what traffic patterns, RTT differences, and congestion/policing levels break it (§6.3)? We use our prototype to answer the former and ns-3 simulation [32] for the latter. In summary, the algorithm incurs no false-negatives under realistic network conditions, and a false-positive rate close to or better than the configured target (5%) even in extremely adversarial scenarios.

6.1 Experimental Setup

Table 2 lists the parameters of our experiments (ranges and default values). A parameter has its bolded value unless otherwise stated.

Topology and Traffic. Each experiment instantiates the topology of Figure 1, where paths p_1 and p_2 perform WeHe𝕐's simultaneous replays (§3.4). Each path replays an (original, bit-inverted) trace pair. We consider one TCP trace pair and five UDP trace pairs, one from each of the UDP applications that WeHe replays: Skype, WhatsApp, MS Teams, Zoom, and Webex [9]. In addition to the trace replays, we send along the two paths background traffic generated from CAIDA Internet traces [4]. Each background traffic stream corresponds to a different segment of a CAIDA trace. To create realistic TCP dynamics, we do not replay TCP packets at the network layer; we extract from the trace the entire TCP flow payloads and replay them from the application layer. For all the experiments we show in this paper, we used the equinix-chicago trace of 29 Oct. 2010. This has an average rate = 168Mbps with around 400 active TCP flows every second.

Performance Metrics. We consider two metrics: (a) *False-negative rate (FN)*: A "false negative" is an experiment where WeHe𝕐's loss-trend correlation algorithm does not detect a common bottleneck even though one exists. (b) *False-positive rate (FP)*: A "false positive" is an experiment where the algorithm does detect a common bottleneck even though one does not exist.

Rate-limiter Configuration and Location. In each experiment, we emulate/simulate one or two rate-limiters, i.e., elements that implement policing or shaping, depending on their queue size. We set the throttling rate and queue size so as to achieve a target average loss rate and queuing delay. We always set the burst size to *rate*×RTT, which guarantees that the throttling rate is achieved (on average) during the experiment. We provide more implementation details in the Appendix, §C.1.

[9] We experiment with real traces from UDP applications because the rate of a UDP replay is determined by the trace. In contrast, the rate of a TCP replay is determined by congestion control (§3.4).

Policer Parameters	
Input Traffic / *rate*	1.3,**1.5**,2,2.5
burst (Byte)	*rate* × RTT
queue (× *burst*)	**0.25**,0.5,1
% of background	25,**50**,75
Rate-limiter location	common link (c), non-common links (nc_i)
Network Parameters	
Input Traffic / Link's bandwidth	**0.2**,0.95,1.05,1.15
RTT_1 (msecs)	**10**,35
RTT_2 (msecs)	**10**,15,25,35,60,120

Table 2: Parameters for emulation/simulation experiments. Default values in bold.

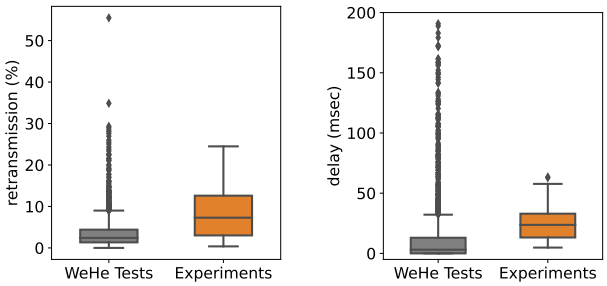
The location of the rate-limiter(s) depends on what we test for: when we test for FN, there is one rate-limiter on the common link sequence, l_c; when we test for FP, there are two identically configured rate-limiters, one on each of the non-common link sequences, l_1 and l_2.

We emulate/simulate per-application/per-service throttling: When a link sequence deploys a rate-limiter, it directs to it: (a) any traffic that belongs to an original trace, and (b) some fraction of the background traffic. The remaining background traffic and any traffic that belongs to a bit-inverted trace bypasses the rate-limiter. The fraction of the background traffic that is directed to the rate-limiter plays the role of traffic from the same application or service as the original WeHe trace, which competes for the rate-limiter's resources. The fraction ranges from 25% to 75%, but we adjust the throttling rate and queue size to achieve a particular target loss rate and queuing delay.

Testbed. Our testbed consists of: (a) Two servers located in different zones of a GCP data-center. Each server runs the WeHe$\mathbb{Y}$ server code on an e2-standard virtual machine (VM) with an Ubuntu 20.4 LTS boot image. (b) A client in the same continent as the GCP data-center. This is a Linux machine with an Intel Core i7-4770 CPU Processor and 32 GiB of memory. It runs the WeHe$\mathbb{Y}$ client code and potentially a software switch that relies on Linux's Traffic Control (tc) tools (§6.2).

6.2 Wide-area Testbed with Emulated Rate-Limiter

Experiments. We emulated scenarios where traffic experiences throttling at the common link sequence l_c. We varied the rate-limiter's *rate*, *burst* size, and *queue* size such that: (a) Traffic arrived at the rate-limiter at 1.3×, 1.5×, 2×, and 2.5× the *rate*; the larger this factor, the higher the resulting loss. (b) The *queue* size was 0.25×, 0.5×, and 1× the *burst* size; the larger this factor, the higher the resulting queuing delay, and the more we emulated shaping rather than policing. This yielded 12 rate-limiter configurations; for each one and for each of our six trace pairs (one TCP and five UDP), we ran five experiments, each with different (randomly selected) background traffic. So, in total, we ran 360 emulation experiments. In 41 of these, WeHe did not detect traffic differentiation (the throttling was not significant enough). We discuss only the remaining 319 experiments.



(a) Retransmission rate. **(b) Delay.**

Figure 5: Original-replay properties observed in past WeHe tests and in our emulation experiments.

To demonstrate that these experiments cover a diverse set of realistic network conditions resulting from real policing or shaping, we compare them with data gathered from real WeHe tests. In Figure 5, we present boxplots of the average retransmission rate and queuing delay measured during the original replays from both our TCP experiments (orange) and real WeHe tests (performed in the wild) that detected traffic differentiation (gray); we describe how we derived the latter in the Appendix, §C.2. Regarding retransmissions (Figure 5a), we see that the first and third quartile from our experiments (orange boxplot) covers the full range of retransmission rates from past WeHe tests (gray boxplot). Regarding delay (Figure 5b), our experiments cover a significant fraction of the delays from past WeHe tests.

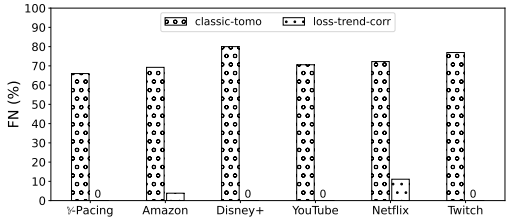
Results. In all 319 experiments, WeHe$\mathbb{Y}$'s loss-trend correlation algorithm correctly detected a common bottleneck (FN = 0).

We further explore what WeHe$\mathbb{Y}$'s FN *would* be with different design choices: (a) If it relied on classic tomography for common bottleneck detection, and (b) if it replayed the (original, bit-inverted) traces unmodified. Figure 6a shows the results for TCP traffic: The first pair shows FN when replaying modified traces. Each of the other pairs shows FN when replaying unmodified traces from the corresponding application. In each pair, the bar on the right shows FN when relying on loss-trend correlation, while the bar on the left shows FN when relying on BinLossTomoNoParams (Alg. 4, our best algorithm that relies on classic tomography). So, the right-most bar of the first pair shows WeHe$\mathbb{Y}$'s FN (which is 0), while all other bars correspond to different design choices. We see that classic tomography increases FN by 66–82%, while the unmodified traces further increase it by 3–11%, Figure 6b shows similar results for UDP traffic: All pairs show FN when relying on BinLossTomoNoParams. In each pair, the bar on the left shows FN when replaying unmodified traces from the corresponding application, while the bar on the right shows FN when replaying Poisson traces. We see that classic tomography does better with UDP than TCP, but it still yields non-zero FN rates. So: WeHe$\mathbb{Y}$ needs both its loss-trend correlation algorithm (§4.2) and the simultaneous replay of modified traces (§3.4) to achieve its good FN performance.

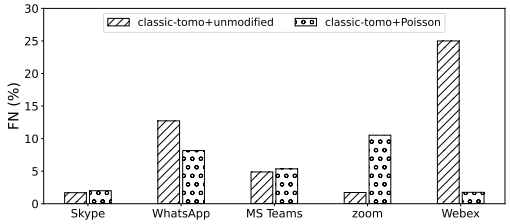
6.3 Simulations (Limit Exploration)

We now explore when and how WeHe$\mathbb{Y}$'s loss-trend correlation algorithm breaks.

FN under Severe Throttling. First, we simulated scenarios where the sole cause of packet loss was throttling at the common link sequence



(a) TCP.



(b) UDP.

Figure 6: False-negative rate of alternative designs.

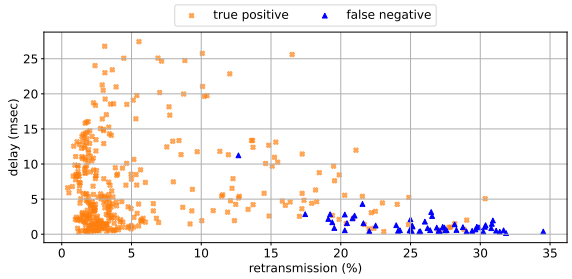


Figure 7: False-negative rate for different TCP retransmission rates.

	15msec	25msec	35msec	60msec	120msec
UDP - FN	0%	0%	0%	0%	21.33%
TCP - FN	21.66%	25.86%	28.33%	31.66%	50%

Table 3: False-negative rate for different RTTs.

l_c. We set the experimental parameters as in §6.2, except: the RTTs were around 35msec, and we varied the fraction of background traffic that was directed to the rate-limiter according to all the values in Table 2. WeHe$\mathbb{Y}$'s overall FN in these experiments was = 19.2%. Most false negatives were TCP experiments with a retransmission rate above 20%, which resulted from directing 75% of the background traffic to the rate-limiter. Figure 7 illustrates this: each data point is an (average retransmission rate, average queuing delay) tuple resulting from a TCP experiment; the orange points are true positives, while the blue ones are false negatives. In summary, when the retransmission rate exceeds 20%, the too-frequent loss events cause significant desynchronization between the two TCP flows of the simultaneous replay (in many intervals, only one flow can "see" some of its packets surviving the rate-limiter), and TCP pacing is not enough to outbalance this. *Conclusion*: When throttling causes TCP retransmission rates above 20%, the FN performance of the loss-trend correlation algorithm deteriorates. On the positive side, such high TCP retransmission rates are uncommon in the wild (Figure 5a, gray boxplot).

	0.95 (low)	1.05 (medium)	1.15 (high)
UDP - FN	0%	0.38%	2.38%
TCP - FN	19.3%	28%	34.88%

Table 4: False-negative rate under severe congestion.

TCP	Skype	WhatsApp	MS Teams	Zoom	Webex
1.13%	2.5%	1.67%	3.75%	3.27%	2.5%

Table 5: False-positive rate under identical rate-limiters.

FN under Large RTTs. Next, we repeated the same experiments, except: we set p_1's RTT to 35msec, and we varied p_2's RTT from 15 to 120msec. We selected these RTTs based on the 5th, 25th, 50th, 75th, and 95th percentile of the RTTs observed by the bit-inverted traces in past WeHe tests. Table 3 shows the FN of the loss-trend correlation algorithm for different RTTs. In general, FN does not change significantly, except when p_2's RTT is 120msec (so, the RTT difference is 85msec), at which point FN increases to 50% and 21.33% for TCP and UDP experiments, respectively. The main reason for the deterioration is the following: the larger the RTT, the larger the interval size of the loss-trend correlation algorithm (which is set to multiple RTTs), hence, the fewer the intervals per experiment; in general, fewer intervals make it more likely for the Spearman correlation test to be inconclusive. *Conclusion*: The loss-trend correlation algorithm is, in general, robust to the RTTs.

FN under Severe Congestion. Next, we simulated scenarios where packet loss was due to throttling on the common link sequence, as well as standard network congestion on the non-common link sequences, l_1 and l_2. We set the experimental parameters as in "FN under Severe Policing," except: we varied l_1 and l_2's transmission rates according to all the values in Table 2. Table 4 shows the FN of the loss-trend correlation algorithm for different levels of congestion on l_1 and l_2. Under medium (resp. high) congestion, traffic experiences 2x (resp. 3x) more loss on l_1 and l_2 than under low congestion. As expected, FN increases as l_1 and l_2 become increasingly congested because they become the dominant bottlenecks for the respective paths, causing the two paths' loss rates to become decorrelated. *Conclusion:* When the non-common link sequences become severely congested, the FN performance of the loss-trend correlation algorithm deteriorates. However, these are arguably not real false negatives, in the sense that the dominant bottleneck for each path is not the common link sequence anymore. In any case, when an ISP's traffic differentiation is not the dominant cause of packet loss, we think that it is reasonable for WeHe$\mathbb{Y}$ to *not* localize that differentiation to the ISP.

FP under Identical Rate-Limiters. Finally, we simulated scenarios where the sole cause of packet loss was throttling at the *non-common* link sequences, l_1 and l_2. We varied the rate-limiters' configuration according to Table 2, but kept the two configurations identical in each experiment. In our context, this is the ultimate FP test: it tests whether the loss-trend correlation algorithm can distinguish between two paths being subjected to the same rate-limiter, versus two paths being subjected to different, *identically configured* rate-limiters. We use it not because it is realistic, but because we cannot imagine any realistic scenario that would make it more likely for our algorithm to yield a false positive. Table 5 shows the FP of the loss-trend correlation algorithm for different trace pairs. We see that

all experiments yielded an FP that was close to or better than the target (5%). *Conclusion*: The loss-trend correlation algorithm achieves the target FP, even under the extremely adversarial scenario of an independent but identically configured rate-limiter on each path.

7 DISCUSSION

Common bottleneck assumption: WeHe$\mathbb{Y}$'s main limitation is the common-bottleneck assumption (§3.2): because of it, WeHe$\mathbb{Y}$ cannot localize traffic differentiation implemented with per-flow policers/shapers. We can remove this limitation as follows: In Step #2 (§3.1) when servers s_1 and s_2 replay the original trace along paths p_1 and p_2, we can slightly modify the replayed trace instances such that they appear to belong to the same flow; this way, traffic from the two paths will be assigned to the same policer/shaper, and WeHe$\mathbb{Y}$ should be able to detect the common bottleneck. However, there is an additional challenge to solve: the traffic from the two paths will be the *only* traffic assigned to the policer/shaper, meaning that the two paths will significantly affect each other's performance; accurately detecting performance correlation in that scenario is harder and will require different statistical tools.

TCP loss inference: We infer packet loss for TCP flows at the server using standard approaches that are not 100% accurate (e.g., packet retransmissions can occur if packets are delivered late). We believe these events are random and thus do not bias our results. Further, middleboxes such as transparent TCP proxies may hide end-to-end packet loss from the server. For such cases, WeHe already uses client-side application-layer throughput samples to detect differentiation that is not seen at the server. In this scenario, one can simply use a technique similar to TraceBox [8] to identify the location of the proxy.

BBR/QUIC: We evaluated WeHe$\mathbb{Y}$ on TCP Cubic, and it is an open question how loss rate correlations would occur with BBR flows. On the one hand, BBR uses pacing like our approach. On the other hand, BBR adjusts its sending rate such that loss should occur only during the probe-bandwidth phase. We did not evaluate our system using QUIC; we believe it would perform similarly to whatever underlying congestion control algorithm is selected by QUIC (e.g., Cubic or BBR).

8 RELATED WORK

Network neutrality inference [45] is the intellectual ancestor of our work, as it proposes network performance tomography to detect and localize traffic differentiation. That work, however, is more theoretical than ours, focusing on (in)feasibility results. It does include an algorithm for identifying non-neutral link sequences within large topologies; however, when applied to our context, that algorithm performs on par with Alg. 3.

WeHe$\mathbb{Y}$'s loss-trend correlation algorithm is related to the one presented by Kurose et al. [36], in that they both detect a common bottleneck based on end-to-end measurements, and they both rely on correlation metrics. That algorithm, however, operates on measurements collected through packet pairs: each packet in a pair travels along a different path, and the two packets are expected to reach the common bottleneck at approximately the same time. The algorithm then computes the correlation of the packet-loss events along the two paths. We tried to leverage such packet-level correlation, but it did not work well in our context: even if two packets arrive at a common policer/shaper close to each other, it is highly likely that only one of them is lost.

WeHe$\mathbb{Y}$ is also related to 007 [1], in that, they both leverage network performance tomography to reason about the behavior of a target network without access to the latter's network devices. In particular, 007 labels a link as problematic if enough problematic paths traverse it. That system, however, does not need to perform concurrent measurements, nor to compute the performance correlation of different paths that converge at a target link sequence. In our context, these elements are necessary because they enable us to find hard evidence of traffic differentiation.

Finally, DiffProbe [17] and Glasnost [10] were conceptually similar to WeHe—they detected traffic differentiation on an end-to-end path based on end-to-end performance comparisons—but focused on BitTorrent traffic. NetPolice [43] followed the same principle but additionally used traceroute-like probes to directly estimate the loss rate of specific path segments, a technique reused later by NVLens [44]. ShaperProbe [18] and Packsen [41] detected a particular differentiation mechanism—shaping—on an end-to-end path. Complementary to the above, Nano detected whether "an ISP causes performance degradation for a service when compared to performance for the same service through other ISPs" [39].

9 CONCLUSION

WeHe detects the presence of traffic differentiation on an end-to-end path based on throughput comparisons; we have presented WeHe$\mathbb{Y}$, which determines whether any detected differentiation is due to the user's ISP. WeHe$\mathbb{Y}$'s main contribution is that it provides concrete evidence that the traffic differentiation indeed happened within the user's ISP (as the user might have suspected but been unable to prove). Under the covers, WeHe$\mathbb{Y}$ builds on network tomography, evolved to eliminate sensitivity to "magic numbers" (that are hard to get right in our context) and short-term, packet-level correlations (that do not always exist in our context). Our experiments showed that WeHe$\mathbb{Y}$ works—with a low false-negative rate and a low, configurable false-positive rate—under the most adversarial network conditions we were able to create, using both a wide-area testbed and simulations. Armed with sound localization results from WeHe$\mathbb{Y}$, users can make more informed decisions for ISP adoption, and policymakers can better understand the landscape and implications of non-neutrality in network providers.

ACKNOWLEDGMENTS

This work was funded by an M-Lab/ISOC fellowship. We would like to thank the M-Lab team for providing the infrastructure and support essential to our system, especially Lai Yi Ohlsen and Stephen Soltesz. We would also like to thank our shepherd, Matt Calder, for his thoughtful comments and his patience, as well as the anonymous reviewers for their constructive feedback. Finally, we would like to thank Derek Ng for his help in implementing WeHe$\mathbb{Y}$.

REFERENCES

[1] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 419–435.

[2] Francçois Baccelli, Sridhar Machiraju, Darryl Veitch, and Jean C. Bolot. 2006. The Role of PASTA in Network Measurement. In *Proceedings of the 2006 Conference*

on Applications, Technologies, Architectures, and Protocols for Computer Communications (Pisa, Italy) *(SIGCOMM '06)*. Association for Computing Machinery, New York, NY, USA, 231–242. https://doi.org/10.1145/1159913.1159940
[3] Philippe Biondi and the Scapy community. 2023. Scapy API reference. https://scapy.readthedocs.io/en/latest/api/scapy.html. Accessed August 2023.
[4] CAIDA. 2023. The CAIDA Anonymized Internet Traces Dataset (April 2008 - January 2019). https://www.caida.org/catalog/datasets/passive_dataset/. Accessed August 2023.
[5] Rui Castro, Mark Coates, Gang Liang, Robert Nowak, and Bin Yu. 2004. Network Tomography: Recent Developments. *Statist. Sci.* 19, 3 (2004), 499 – 517. https://doi.org/10.1214/088342304000000422
[6] AHeroIII Coates, Alfred O Hero III, Robert Nowak, and Bin Yu. 2002. Internet tomography. *IEEE Signal processing magazine* 19, 3 (2002), 47–65.
[7] Mark Coates, Alfred Hero, Robert Nowak, and Bin Yu. 2002. Internet Tomography. *IEEE Signal Processing Magazine* 19, 3 (2002), 47–65.
[8] Gregory Detal, Benjamin Hesmans, Olivier Bonaventure, Yves Vanaubel, and Benoit Donnet. 2013. Revealing Middlebox Interference with Tracebox. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[9] A Dhamdhere, R Texeira, C Dovrolis, and C Diot. 2007. Netdiagnoser: Troubleshooting Network Unreachabilities Using End-to-end Probes and Routing Data. In *Proceedings of the ACM CoNEXT Conference*.
[10] Marcel Dischinger, Massimiliano Marcon, Saikat Guha, Krishna P Gummadi, Ratul Mahajan, and Stefan Saroiu. 2010. Glasnost: Enabling End Users to Detect Traffic Differentiation. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[11] M Dischinger, A Mislove, A Haeberlen, and K P Gummadi. 2008. Detecting BitTorrent Blocking. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[12] N Duffield. 2006. Network Tomography of Binary Network Performance Characteristics. In *IEEE Transactions on Information Theory, 52(12):5373-5388.*
[13] Tobias Flach, Pavlos Papageorge, Andreas Terzis, Luis Pedrosa, Yuchung Cheng, Tayeb Karim, Ethan Katz-Bassett, and Ramesh Govindan. 2016. An internet-wide analysis of traffic policing. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 468–482.
[14] Denisa Ghita, Katerina Argyraki, and Patrick Thiran. 2011. Shifting Network Tomography Toward a Practical Goal. In *Proceedings of the ACM CoNEXT Conference*.
[15] Yiyi Huang, Nick Feamster, and Renata Teixeira. 2008. Practical Issues with Using Network Tomography for Fault Diagnosis. *SIGCOMM Comput. Commun. Rev.* 38, 5 (sep 2008), 53–58. https://doi.org/10.1145/1452335.1452343
[16] T-Mobile USA Inc. 2023. Data Maximizer for Prepaid Plans. https://www.t-mobile.com/support/plans-features/data-maximizer-for-prepaid-plans. Accessed August 2023.
[17] Partha Kanuparthy and Constantine Dovrolis. 2010. DiffProbe: Detecting ISP Service Discrimination. In *Proceedings of the IEEE INFOCOM Conference*.
[18] Partha Kanuparthy and Constantine Dovrolis. 2011. ShaperProbe: End-to-end Detection of ISP Traffic Shaping Using Active Methods. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.
[19] Ken Keys, Young Hyun, Matthew Luckie, and K Claffy. 2011. Internet-scale ipv4 alias resolution with midar: System architecture. *Cooperative Association for Internet Data Analysis (CAIDA), Tech. Rep* (2011).
[20] Measurement Lab. 2023. Measure the Internet, save the data, and make it universally accessible and useful. https://www.measurementlab.net/. Accessed August 2023.
[21] Measurement Lab. 2023. Wehe. https://measurementlab.net/tests/wehe. Accessed August 2023.
[22] Jean-Yves Le Boudec. 2010. *Performance evaluation of computer and communication systems.* Epfl Press.
[23] Fangfan Li, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. 2019. A large-scale analysis of deployed traffic differentiation practices. In *Proceedings of the ACM Special Interest Group on Data Communication.*
[24] Matthew Luckie. 2010. Scamper: a scalable and extensible packet prober for active measurement of the internet. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 239–245.
[25] Linux manual page. 2023. tc-mirred(8). https://man7.org/linux/man-pages/man8/tc-mirred.8.html.
[26] Linux manual page. 2023. tc-tbf(8). https://man7.org/linux/man-pages/man8/tc-tbf.8.html.
[27] Linux manual page. 2023. tc(8). https://man7.org/linux/man-pages/man8/tc.8.html.
[28] Measurement Lab. 2023. Traceroute. https://measurementlab.net/tests/traceroute. Accessed August 2023.
[29] Juniper Networks. 2023. burst-size. https://www.juniper.net/documentation/us/en/software/junos/security-services/topics/ref/statement/burst-size-rate-limiting.html. Accessed August 2023.
[30] Hung Xuan Nguyen and Patrick Thiran. 2007. The boolean solution to the congested IP link location problem: Theory and practice. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications.*
[31] Hung X. Nguyen and Patrick Thiran. 2007. Network Loss Inference with Second Order Statistics of End-to-End Flows. In *Proceedings of the IEEE Internet Measurement Conference (IMC).*
[32] nsnam. 2023. ns-3 Network Simulator. https://www.nsnam.org/. Accessed August 2023.
[33] V N Padmanabhan, L Qiu, and H J Wang. 2003. Server-based Inference of Internet Performance. In *Proceedings of the IEEE INFOCOM Conference.*
[34] Teresa Pepe and Marzio Puleri. 2015. Network Tomography: A novel algorithm for probing path selection. In *2015 IEEE International Conference on Communications (ICC)*. 5337–5341. https://doi.org/10.1109/ICC.2015.7249172
[35] Yan Qiao, Jun Jiao, Yuan Rao, and Huimin Ma. 2016. Adaptive Path Selection for Link Loss Inference in Network Tomography Applications. *PLOS ONE* 11, 10 (10 2016), 1–21. https://doi.org/10.1371/journal.pone.0163706
[36] D. Rubenstein, J. Kurose, and D. Towsley. 2002. Detecting shared congestion of flows via end-to-end measurement. *IEEE/ACM Transactions on Networking* 10, 3 (2002), 381–395. https://doi.org/10.1109/TNET.2002.1012369
[37] H H Song, L Qiu, and Y Zhang. 2006. NetQuest: A Flexible Framework for Large-Scale Network Measurement. In *Proceedings of the ACM SIGMETRICS Conference.*
[38] Lizhuang Tan, Wei Su, Wei Zhang, Jianhui Lv, Zhenyi Zhang, Jingying Miao, Xiaoxi Liu, and Na Li. 2021. In-band network telemetry: A survey. *Computer Networks* 186 (2021), 107763.
[39] Mukarram Bin Tariq, Murtaza Motiwala, Nick Feamster, and Mostafa Ammar. 2008. Detecting Network Neutrality Violations with Causal Inference. In *Proceedings of the ACM CoNEXT Conference.*
[40] VOXMEDIA. 2023. Trump's FCC has revealed plans to wipe out net neutrality. https://www.vox.com/2017/11/21/16679114/fcc-ajit-pai-net-neutrality-rules-donald-trump. Accessed August 2023.
[41] Udi Weinsberg, Augustin Soule, and Laurent Massoulie. 2011. Inferring Traffic Shaping and Policy Parameters using End Host Measurements. In *Proceedings of the IEEE INFOCOM Mini-Conference.*
[42] Ronald W. Wolff. 1982. Poisson Arrivals See Time Averages. *Operations Research* 30, 2 (1982), 223–231. http://www.jstor.org/stable/170165
[43] Ying Zhang, Zhuoqing Morley Mao, and Ming Zhang. 2007. Detecting Traffic Differentiation in Backbone ISPs with NetPolice. In *Proceedings of the ACM Internet Measurement Conference (IMC).*
[44] Ying Zhang, Z. Morley Mao, and Ming Zhang. 2008. Ascertaining the Reality of Network Neutrality Violation in Backbone ISPs. In *Proceedings of the ACM Hot Topic in Networks (HotNets).*
[45] Zhiyong Zhang, Ovidiu Mara, and Katerina Argyraki. 2014. Network Neutrality Inference. In *Proceedings of the ACM SIGCOMM Conference.*

A ETHICS

This work does not raise any ethical issues: the only user data that we use is already publicly available through the WeHe project, which is gathered with informed consent, and contains no user identifiers.

B INTERMEDIATE ALGORITHMS

We list the pseudo code of the intermediate algorithms described in Section 4.3.

Algorithm 2 BinLossTomo

Input: Packet-loss measurements (M_1, M_2),
Interval size σ,
Loss threshold τ

Output: Link performance (x_c, x_1, x_2)

$(LostPkts_1, TxedPkts_1, LostPkts_2, TxedPkts_2) \leftarrow$ CreateTimeSeries$((M_1, M_2), \sigma)$
for $t \in 0..T-1$ **do**
  for $i \in 1,2$ **do**
    $LossStatus_i[t] \leftarrow \frac{LostPkts_i[t]}{TxedPkts_i[t]} > \tau$
  end for
end for
$(y_1, y_2) \leftarrow \left(\frac{\Sigma_t LossStatus_1[t]}{T}, \frac{\Sigma_t LossStatus_2[t]}{T}\right)$
$y_{12} \leftarrow \frac{\Sigma_t LossStatus_1[t] \; AND \; LossStatus_2[t]}{T}$
return $x_c = \frac{y_1 y_2}{y_{12}}, x_1 = \frac{y_{12}}{y_2}, x_2 = \frac{y_{12}}{y_1}$

BinLossTomo (Alg. 2) is the state-of-the-art tomography algorithm described in Section 2.2. It takes as input packet-loss measurements of p_1 and p_2 (collected according to Section 3.4), interval size σ, and loss threshold τ. The output is the links' inferred performances x_c, x_1 and x_2 (see Figure 1).

Algorithm 3 BinLossTomo++

Input: Packet-loss measurements (M_1, M_2),
Interval size σ,
Loss threshold τ
Output: True or False

$(x_c, x_1, x_2) \leftarrow \text{BinLossTomo}((M_1, M_2), \sigma, \tau)$
return $(x_1 > x_c)$ AND $(x_2 > x_c)$

BinLossTomo++ (Alg. 3) is our first intermediate algorithm in Section 4.3. It uses the output of BinLossTomo to check if the common link's performance (x_c) is worse than the performance of both non-common links (x_1 *and* x_2). If yes, the output is a common bottleneck; otherwise, no evidence.

Algorithm 4 BinLossTomoNoParams

Input: Packet-loss measurements (M_1, M_2),
Output: True or False

$\mathcal{S} \leftarrow$ sequence of interval sizes
$\mathcal{T} \leftarrow$ sequence of loss thresholds
for $\sigma \in \mathcal{S}$ **do**
  for $\tau \in \mathcal{T}$ **do**
    $(x_c, x_1, x_2) \leftarrow \text{BinLossTomo}((M_1, M_2), \sigma, \tau)$
    $\Delta_1[\tau][\sigma] \leftarrow x_1 - x_c$; $\Delta_2[\tau][\sigma] \leftarrow x_2 - x_c$
  end for
end for
return $(Avg_{\mathcal{S}_{\sigma,\tau}}(\Delta_1[\tau][\sigma]) > 0)$ AND $(Avg_{\mathcal{S}_{\sigma,\tau}}(\Delta_2[\tau][\sigma]) > 0)$

BinLossTomoNoParams (Alg. 4) is our second intermediate algorithm. Given the packet-loss measurements of p_1 and p_2, it infers a sequence of interval sizes $\mathcal{S}$ and loss thresholds $\mathcal{T}$. Then, it compute $Avg_{\mathcal{S}_{\sigma,\tau}}(\Delta_1[\tau][\sigma])$ and $Avg_{\mathcal{S}_{\sigma,\tau}}(\Delta_2[\tau][\sigma])$: the average gap between the non-common link performance x_1 (resp. x_2) and the common link performance x_c, across all the combinations of $\mathcal{S}$ and $\mathcal{T}$. If both gaps are positive, it outputs a common bottleneck; otherwise, there is no evidence.

C ADDITIONAL EVALUATION INFORMATION

C.1 Rate-Limiter Implementation

For the simulation experiments (§6.3), the rate-limiter is implemented in the ns-3 network simulator. It consists of three main components:

- A *classifier*: classifies incoming flows into 2 fictitious traffic classes according to the DSCP (differentiated services code point) field of the IP header. Those with dscp=1 experiences differentiation, while others doesn't. Packets that belongs to the original WeHe measurement traces are assigned dscp=1. To simulate collective per-application/per-service throttling (scenario (b) in §4), some of the background TCP flows (or UDP packets if the measurement traffic was UDP) are randomly assigned a dscp=1; this way they are throttled with the original traces. The remaining traffic uses the default dscp=0 value.
- Two queues: which uses ns-3's queuing disciplines. The first handles the default traffic which does not experience differentiation (i.e., dscp=0); it uses simple *FIFO (First-In First-Out)* policy. The second applies the *TBF (token bucket filter)* on flows that should experience differentiation (i.e., dscp=1).
- A *forwarding scheduler*: packets are pushed from the FIFO and TBF queues to the next pipeline in a round-robin manner.

For the wide-area testbed experiments (§6.2), the rate-limiter relies on the Linux's Traffic Control [27] tool. In particular, it uses `tc-mirred` [25] to direct the desired incoming traffic to an ifb (Intermediate Functional Block) interface with an attached TBF (configured using [26]).

In both scenarios, the rate-limiter is configured following the guidelines offered by Juniper Networks [29] and the `tc-tbf` documentation [26]. In details, the TBF parameters are set as follows: the *peakrate* is set to 0 since we are not interested in perfect millisecond timescale shaping; the *rate* is set to the desired throttling rate that the differentiated traffic class should experience; and the *burst*, which is the size of the token bucket, is set to *rate*×RTT. We chose 1-RTT for the allowable burst time to guarantee that the desired policing rate will be achieved (on average) during the experiment. Finally, the *limit* – which is the TBF queue size – is set based on the desired average delay the differentiated traffic class should experience waiting for tokens to become available.

C.2 Loss/Delay WeHe Measurements

Figure 5 shows boxplots of the average retransmission rate and queuing delay collected by the original replays from past WeHe tests (the gray boxplots). We obtained these as follows: We downloaded 1 year's (Sep.1 2022 to 2023) worth of WeHe data and identified all the tests that detected traffic differentiation. From these, we kept only the ones where the bit-inverted trace experienced 0 loss (to maximize the chance that any retransmissions experienced by the original trace were due to traffic differentiation, not general congestion). For each test, we estimated the average queuing delay experienced by each trace as the average minus the minimum RTT; then, we estimated the average queuing delay experienced by the original trace *due to traffic differentiation*, by subtracting the average queuing delay experienced by the bit-inverted trace from that experienced by the original trace.