Continuous In-Network Round-Trip Time Monitoring

Satadal Sengupta
Princeton University, USA
satadal.sengupta@cs.princeton.edu

Hyojoon Kim
Princeton University, USA
hyojoonk@cs.princeton.edu

Jennifer Rexford
Princeton University, USA
jrex@cs.princeton.edu

ABSTRACT

Round-trip time (RTT) is a central metric that influences end-user QoE and can expose traffic-interception attacks. Many popular RTT monitoring techniques either send active probes (that do not capture application-level RTTs) or passively monitor only the TCP handshake (which can be inaccurate, especially for long-lived flows). High-speed programmable switches present a unique opportunity to monitor the RTTs continuously and react in real time to improve performance and security. In this paper, we present `Dart`, an inline, real-time, and continuous RTT measurement system that can enable automated detection of network events and adapt (e.g., routing, scheduling, marking, or dropping traffic) inside the network. However, designing `Dart` is fraught with challenges, due to the idiosyncrasies of the TCP protocol and the resource constraints in high-speed switches. `Dart` overcomes these challenges by strategically limiting the tracking of packets to only those that can generate useful RTT samples, and by identifying the synergy between per-flow state and per-packet state for efficient memory use. We present a P4 prototype of `Dart` for the Tofino switch, as well our experiments on a campus testbed and simulations using anonymized campus traces. `Dart`, running in real time and with limited data-plane memory, is able to collect 99% of the RTT samples of an offline, software baseline—a variant of the popular `tcptrace` tool that has access to unlimited memory.

CCS CONCEPTS

• **Networks** → *Transport protocols*; **Network measurement**; Network performance analysis; **Programmable networks**; *In-network processing*; *Network management*;

KEYWORDS

round-trip time, passive measurement, network monitoring, high-speed programmable switch

ACM Reference Format:
Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous In-Network Round-Trip Time Monitoring. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3544216.3544222

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544222

1 INTRODUCTION

Round-trip time (RTT) is a key indicator of network performance, user Quality of Experience (QoE), and routing-protocol attacks [17, 31]. RTT relates directly to Transmission Control Protocol (TCP) throughput and also heavily influences higher-level metrics such as video QoE and page load time [6, 10]. Changes in RTTs can also be symptoms of malicious activities like traffic interception attacks [17]. In the following examples, RTT monitoring can inform important network adaptation decisions:

- RTT can be a good indicator of network congestion. When network performance starts to decline, and multiple paths are available, the network can reroute traffic to an alternate, less-congested path. This applies to routing in the wide-area network [3, 22] as well as within data centers [21].
- RTT monitoring is useful for latency-sensitive applications. For example, multiplayer cloud-gaming applications need to select the best game server for players spread across different geographical regions [11, 13]. The network can monitor the propagation delay (minimum RTT over time) en route to each potential server, and select the best one for each gaming session.
- RTT can help in inferring the QoE of on-demand video applications. For instance, an RTT hike can cause an increase in video startup delay and a decrease in video resolution [10], thus prompting the service provider to select an alternate server or path.
- RTT monitoring can help reveal routing attacks. Nation-state actors can eavesdrop on traffic by launching Border Gateway Protocol (BGP) interception attacks [5]. These attacks cause a sudden and substantial increase in RTT. The network could detect and immediately stop sensitive traffic on such occasions.

These use-cases illustrate that the network can perform control actions—such as dropping, marking, scheduling, or routing traffic—to rapidly mitigate declining network conditions. To adapt effectively, the network needs RTT measurements that are accurate, real-time, and actionable. For example, during a traffic-interception attack, the network should mitigate the attack before the adversary sees too much of the traffic.

Many popular measurement tools use *active* probes such as ICMP pings to estimate the RTT to remote hosts (e.g., iperf3 [15] and RIPE Atlas [25]). However, probe-based RTT estimates do not capture application-specific RTTs. Active probes also introduce extra traffic load and may be blocked by the remote host or network. Instead, measuring RTTs *passively* by observing the actual user traffic provides a more accurate estimate [18]. Passive RTT monitoring at the end-host requires special software (e.g., a native mobile app) or permissions. Instead, monitoring at a vantage point en route to many end-hosts (e.g., near the gateway router) enables easy aggregation of network-wide RTTs. Passive RTT monitoring for TCP involves

matching packets with their corresponding acknowledgments.[1] RTT measurements of TCP traffic can also be used to infer RTTs for UDP traffic (e.g., QUIC and RTP-based video conferencing) between the same end-points or IP prefixes [4, 24].

One prevalent passive-measurement technique estimates a flow's RTT based only on the TCP three-way handshake [14]. This approach can be inaccurate for long flows (e.g., video streaming), since RTTs can vary significantly over minutes let alone hours. Also, handshake RTTs tend to be smaller than a connection's average RTT [14]. Therefore, we must monitor RTTs continuously, beyond the initial handshake.

Computing RTTs continuously is hard. The idiosyncrasies of the TCP protocol—including retransmissions and reordering—can make some RTT samples inaccurate (Section 2). Software tools such as `tcptrace` and `pping` ensure correctness by maintaining expensive flow state and performing complex computations [26, 27]. Unfortunately, RTT monitoring in software is computationally expensive and therefore too slow for networks with high traffic volumes. For example, DPDK-based targets can process at most a few million packets per second [1]. Fortunately, the advent of commodity programmable switches (e.g., the Intel Tofino [2]) and the P4 language [9] opens up the possibility of monitoring RTTs and making adaptation decisions directly in the data plane [30].

However, high-speed data planes impose significant constraints on packet processing in terms of arithmetic operations, memory size, the number of pipeline stages, and recirculation bandwidth. The challenges are exacerbated by the aforementioned idiosyncrasies of TCP traffic, which require maintaining expensive per-flow state and performing computations. Additionally, when memory constraints make it impossible to collect all valid RTT samples, the monitoring mechanism must scale by collecting a representative RTT distribution, even under adversarial traffic (e.g., SYN floods).

In this paper, we present `Dart` (Data-plane Actionable Round-trip Times), a system that monitors on-path RTTs in real time. Our key insights are that we: (1) Strategically limit tracking of packets to only those that can lead to useful RTT samples, and (2) Identify the synergy between per-flow and per-packet state for efficient memory utilization (Section 3). We implement `Dart` in the P4 language on the Intel Tofino switch (Section 4). We evaluate `Dart` on traffic from our campus network and show how our system can detect a traffic-interception attack within only 63 packet exchanges (Section 5). We also implement a faithful Python simulator and report `Dart`'s performance under different configurations (Section 6). `Dart` is able to detect 98% of RTT samples compared to a variant of a software-based baseline `tcptrace` [27].

Ethics statement: The campus traces used in this study were anonymized at the source. All packet traces were inspected and sanitized by a network operator to remove all personal data before being accessed by researchers. The traffic interception attack was carried out using the PEERING testbed [29] with appropriate permission from the maintainers. We performed the attack on a special prefix allocated to us and between end-hosts controlled by us. This study has been conducted with necessary approvals from Princeton University, including its Institutional Review Board (IRB).

[1] TCP timestamps can also be used for passive RTT monitoring, but that method has distinct disadvantages (Section 8).

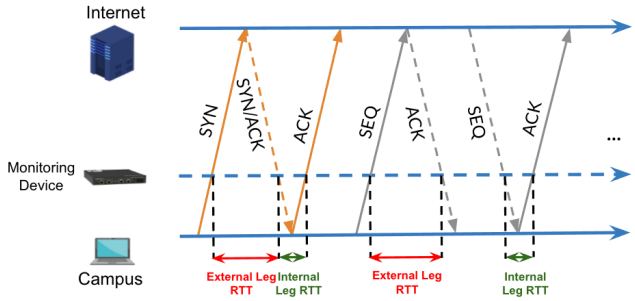


Figure 1: Continuous RTT measurement at a monitoring device by matching TCP data (SEQ) packets with corresponding acknowledgement (ACK) packets.

2 RTT MEASUREMENT CHALLENGES

In this section, we describe a simple strawman design [12] for continuous RTT measurement of TCP flows in the data plane (Section 2.1). We then discuss how the many idiosyncrasies of TCP raise correctness (Section 2.2) and efficiency (Section 2.3) challenges that we address in Section 3.

2.1 Strawman for Measuring RTT

TCP carries a bidirectional data stream between two end-hosts; bytes in one direction are acknowledged in the other direction by appropriately setting sequence and acknowledgment numbers in the TCP header. When placed strategically, a monitoring device can leverage its location to continuously monitor RTTs by matching data and ACK packets. The RTT measurements include end-host delays (e.g., processing time and delayed ACKs) in addition to network delays. We briefly discuss eliminating end-host delays in Section 7.

Seeing both directions of the traffic: To match data packets with corresponding ACKs, monitoring needs to run on a device that can "see" both sides of the traffic. As illustrated in Figure 1, we denote the direction of the TCP data segment as the *SEQ* (sequence) direction, and the direction of the acknowledgment segment as the *ACK* direction. Interestingly, different portions of the end-to-end path can be measured separately depending on the direction of each segment. For example, the RTT computed for a SEQ/ACK pair between the monitoring device and the Internet constitutes the *external leg* of the RTT, whereas that between the monitoring device and the client within the campus constitutes the *internal leg*. The external leg RTT is representative of the wide-area latency to the application server, whereas the internal leg RTT reveals the latency introduced by the campus infrastructure. A combination of consecutive external leg and internal leg RTTs provides the end-to-end application-level RTT for the client.

Matching data packets with ACKs: Continuous RTT measurement requires a data structure for storing SEQ information until the corresponding ACK arrives [12], as shown in Figure 2. The table is indexed by the SEQ packet's unique identifier—the flow identifier (4-tuple of client and server IP addresses and TCP port numbers) and a unique packet identifier within the flow (the expected ACK number or eACK). An entry is created when a SEQ packet arrives,

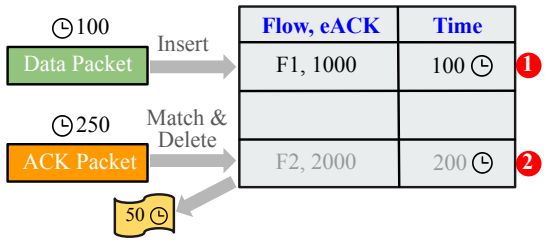


Figure 2: Strawman design: A hash table with flow ID + expected ACK as key and timestamp as value. Arrival of a data (SEQ) packet causes insertion into the hash table, whereas arrival of an ACK triggers deletion of the matching SEQ entry and collection of an RTT sample.

as shown by the entry labelled 1 (white literal within a red circle) in the figure. When a matching ACK packet arrives, we look up the SEQ entry using the key (see entry labelled 2), with the source and destination fields of the 4-tuple reversed. We subtract the entry's SEQ timestamp (200) from the new packet's ACK timestamp (250) to compute the RTT sample (50).

2.2 Challenges with Correctness

The strawman design can lead to incorrect RTT samples under certain common conditions that arise in TCP traffic.

Packet retransmission: Packet losses in a TCP connection can lead to the TCP retransmission ambiguity. Let us say that the sender sends a packet, which the data structure records as a SEQ entry. If the sender does not receive an ACK for this packet, then the sender eventually retransmits the packet, assuming it is lost. At the monitoring device, we see an exact replica of the existing SEQ entry—it is not easy to determine which entry (new or old) ought to be retained in this case. This is because when a corresponding ACK is received, it may be an ACK of the older packet (which simply got delayed due to congestion) or an ACK to the newer packet (because the receiver never saw the older copy due to packet loss).

Packet reordering: A similar ambiguity might arise due to reordered packets. Consider a scenario where the sender sends packets P_1, P_2, P_3, P_4 in order, but the receiver has received in the order of P_1, P_3, P_4, and P_2, i.e., P_2 is reordered by the network. The receiver would send back ACKs corresponding to the last in-order packet it has received (also called duplicate ACKs) to the sender. That is, the receiver would keep ACKing P_1 until P_2 arrives. When P_2 finally arrives, the receiver would immediately ACK not just P_2, but also P_3 and P_4 (a *cumulative ACK*), leading to an erroneously inflated RTT sample for P_4.

2.3 Challenges with Memory Efficiency

Data planes have limited memory, typically on the order of a few tens of megabytes [19]. Quirks in the operation of TCP can lead to inefficient use of this limited memory under the strawman design.

Packets that never receive a matching ACK: When packets arrive in quick succession, the receiver may send a cumulative ACK for every n^{th} packet (rather than per-packet ACKs) to reduce overhead. Any SEQ packet that does not receive an explicit ACK, but is ACKed implicitly by a subsequent packet, would see its entry stranded in the data structure indefinitely. A similar problem arises under SYN flooding attacks; if the TCP handshake never completes, the SEQ packet would be stranded in the data structure.

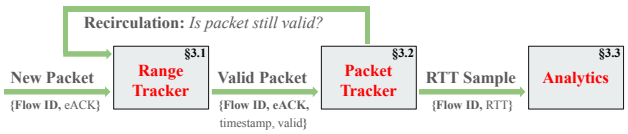


Figure 3: System architecture: New packets are checked at the Range Tracker table for validity. Valid packets update the RT measurement range and then await an ACK in the Packet Tracker table. Generated RTT samples are sent to the Analytics module.

Packets with large RTTs: SEQ packets may legitimately stay for a long time in the data structure before seeing a matching ACK, simply due to network paths with long RTTs.

Handling these two scenarios can be difficult, and expensive. One approach is to apply a timeout to evict SEQ entries that have not yet matched an ACK [12]. However, a small timeout would cause bias against long RTTs, and a large timeout would waste memory on storing SEQ entries that never receive an ACK. Another option is to allow new SEQ packets to evict old SEQ entries as needed. However, this approach also leads to bias against large RTTs. Instead, we need more sophisticated strategies that avoid wasting memory on SEQ entries that cannot lead to valid RTT samples.

3 DART SYSTEM DESIGN

To measure RTTs correctly and efficiently, `Dart` must avoid tracking packets that cannot produce valid RTT samples. The challenge is to make the right decision as each packet streams through the data plane. Figure 3 illustrates our architecture. To avoid storing SEQ entries that could cause ambiguous RTT samples, we track the valid range of sequence numbers for each active flow (Section 3.1). However, we do not always know in advance that a SEQ entry will never match a future ACK. Instead, the packet tracker applies a lazy eviction strategy to reevaluate an old SEQ entry against its flow's up-to-date range of valid sequence numbers (Section 3.2). Finally, the analytics module aggregates the RTT samples (e.g., to compute the minimum RTT per IP prefix). The analytics module also helps optimize memory by purging SEQ entries that cannot produce a *useful* RTT sample—i.e., a sample that affects the analysis results (Section 3.3).

3.1 Tracking Valid Measurement Ranges

Ultimately, for every SEQ packet, `Dart` needs to decide whether to track the packet or not. To this end, we introduce a Range Tracker (RT) table before the packet tracker. The RT table is a hash table with the TCP connection 4-tuple as the key, and a *measurement range* as a value. This range is a sequence number byte-range that can potentially produce correct RTT samples. The left edge of the window indicates the latest byte that was ACKed by the receiver; any future arrival of an *earlier* ACK must have been reordered. The

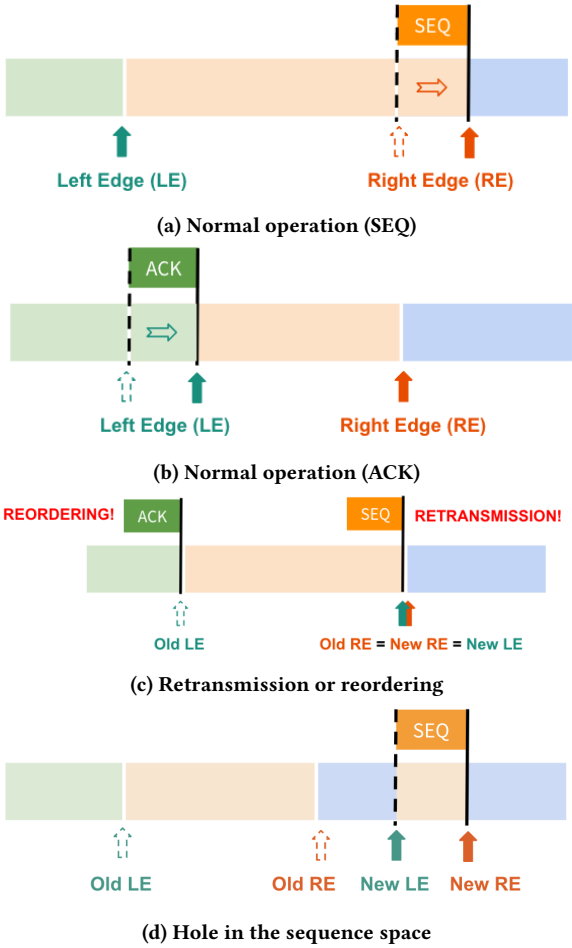


Figure 4: Adjustments to the flow measurement range upon arrival of new SEQ or ACK packets. Green space indicates bytes already covered by the left edge, Orange space indicates contiguous bytes in flight that can potentially lead to valid RTT samples, Blue space indicates sequence numbers not seen by the system.

right edge indicates the latest byte transmitted by the sender; any future arrival of an *earlier* SEQ must have been retransmitted.

Normal operation without ambiguities: Under normal circumstances, the SEQ packets appear in order, causing the right edge of the measurement range to move forward, as shown in Figure 4a. Similarly, ACK packets typically arrive in increasing order, causing the left edge of the measurement range to move forward, as shown in Figure 4b.

Operation with ambiguities: Figure 4c illustrates the operation under TCP ambiguities:

- When a SEQ packet arrives with the expected ACK (eACK) *smaller* than the right edge of the measurement range, we infer a packet retransmission event. For a retransmitted packet, when an ACK arrives in the future, it is ambiguous whether the ACK acknowledges the old or the new copy of the SEQ. Furthermore, if selective acknowledgment (SACK) is not enabled, a retransmission might indicate that the receiver has been waiting on some intermediate SEQ packet(s) before sending out an ACK for a packet it had already received, thus artificially inflating the RTT.
- If an ACK packet arrives for the left edge, we conclude it is a duplicate ACK. We infer a reordering event, since duplicate ACKs are explicit markers of lost or reordered SEQ packets. Similar to retransmission, in this case, too, ACKs have been held up at the receiver thus inflating RTTs.

In both cases, the system infers that the entire measurement range is now ambiguous. In response, therefore, the system *resets* the state: the measurement range is collapsed such that both its left and right edges are now equal to the highest byte transmitted (i.e., the previous right edge). This prevents tracking SEQ packets with expected ACKs same or less than the right edge—now deemed ambiguous. After the collapse, the measurement range updates same as normal operation, or the entry can be safely deleted or overwritten to make room for a new entry. Only the definition of the left edge is updated: it now reflects either the highest byte ACKed (old definition) or the highest byte that was *affected* by a retransmission or reordering ambiguity.

Processing ACKs for untracked SEQ packets: We might see an ACK packet with ACK number either (1) lesser than the left edge or (2) greater than the right edge. Type (1) indicates an ACK to a SEQ packet we have already deemed ambiguous and are not tracking anymore. Our system ignores such ACKs. Type (2) indicates an *optimistic ACK*—a form of early ACKing some receivers use to trick the sender into sending data more quickly [28]. `Dart` ignores these ACKs too, meaning it is not misled into collecting artificially deflated RTTs (see Section 7).

Maintaining a single measurement range: Consider a scenario where our system encounters SEQ packets with one or more packets missing in between; e.g., the sender sends P_1 through P_4 but the system only sees P_1, P_2, and P_4, either due to packet reordering or drop. Note that these packets pass the check illustrated in Figure 4a since they are over the right edge and in sequence. Now, if P_3 is dropped, the system will know only upon detecting a retransmission. If reordered, P_3 arrives after P_4, thus filling in the "hole". The most optimistic approach in this scenario is to assume reordering and to store the states for both byte-ranges, i.e., P_1-P_2, and P_3-P_4. As shown in Figure 4d, this requires storing two measurement windows instead of one. More generally, there could be n such holes, requiring $n+1$ measurement windows. We reason that such a strategy is too expensive in the data plane, and the system should use a constant space for this purpose. We, therefore, store the measurement range for only the highest byte-range ahead of any hole. In the current example, the measurement window would point to only P_4's starting and ending byte numbers.

Robust against congestion and SYN attacks: Maintaining a measurement range per flow with the Range Tracker (RT) table helps significantly with memory efficiency. The mechanism becomes even more crucial when network congestion happens, where more reordering and retransmission events occur. `Dart` is robust to such situations, making sure the space in both the RT and PT tables is

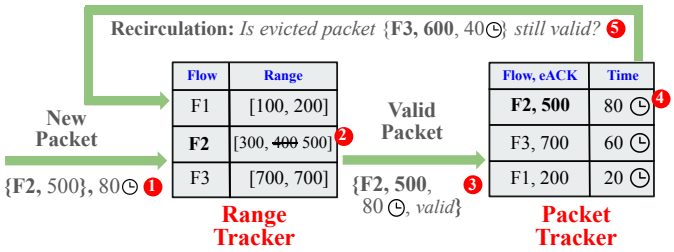


Figure 5: Working of Dart: The system ensures correctness by consulting and updating the Range Tracker (RT) table before inserting packet records into the Packet Tracker (PT) table. Memory efficiency is improved by having packet records consult the RT before re-insertion, a strategy that enables *lazy eviction*.

only used for producing valid, unambiguous samples. The measurement ranges collapse more often, allowing the entries with closed measurement ranges to be deleted or overwritten. This prevents the RT table and PT space from exploding. However, this also means that Dart may collect fewer RTT samples for some flows during heavy congestion. In order to mitigate this, Dart can be adjusted to report the frequency of measurement range collapses for a flow as an indicator of congestion. Dart can also be adjusted to aggregate RTT samples for flows going to the same subnets (e.g., /24 prefixes) before analyzing them. The aggregated RTTs will provide a more complete view of the congestion status of the target subnet.

Dart is also robust against harsh environments, where SYN flooding attacks are common. Dart does not create an entry in the RT table or entries in the PT table until *after* a TCP connection is established, i.e., after the three-way handshake is complete. That is, Dart completely ignores SYN and SYN-ACK packets. Therefore, the memory usage does not explode in either the RT or PT tables during SYN flooding attacks either. We discuss Dart's performance under other kinds of attacks in Section 7.

3.2 Lazy Eviction with a Second Chance

The Range Tracker (RT) table helps with memory efficiency, but the Packet Tracker (PT) table can still end up storing a SEQ packet that never matches a subsequent ACK. For example, the receiver might just cut off the TCP session, never sending an ACK. Sometimes, cumulative ACKs render certain packet entries unmatched since a higher byte number has already been ACKed by the receiver. These unmatched entries occupy expensive space in the PT table; we would rather reclaim this space for tracking new SEQ packets. The goal is clear: when a new packet comes and hash collision occurs, we want to keep an old entry if it is not stale, but otherwise replace it with the new packet. What is the most accurate and efficient way to achieve this?

Timeout is biased, garbage collection is expensive: One obvious way to deal with this issue is to associate a timeout with each entry [12] in the PT table. However, packets with naturally long RTTs can suffer from undersampling under this arrangement, on top of the challenge for finding the right timeout value. A more accurate alternative is to actively scan the PT table for stale entries when an ACK arrives. For example, there can be an active garbage collector that periodically polls the RT table and remove stale entries from the PT. This strategy, however, is far too expensive, especially if implemented in the data plane.

Lazy eviction: The goal is to create an efficient mechanism that replaces entries that are indeed stale and do this without bias. Ideally, this should also happen without additional control-plane interaction, as that would introduce extra overhead. Our first observation is that an entry does not need to be evicted until there is actually a hash collision with a new entry. We employ this *lazy eviction* strategy: this gives an entry, no matter how old it is, a chance to produce a valid sample as long as a new entry does not cause a hash collision.

Recirculation for a second chance: Now, when a new entry indeed causes a hash collision with an old entry, blindly replacing the old entry with a new one creates a bias towards samples with short round-trip times. This is because the old entry might have still produced a valid sample had it waited long enough. We indeed want to keep the old entry if it is still valid. The challenge is, however, that we do not know this *at the moment when the hash collision happens*. To address this, Dart recirculates the old entry, sending it back to the start of the ingress pipeline through the RT table again. This allows us to re-validate the entry against the measurement range in the RT table, thus checking for staleness. Meanwhile, we do not want to lose the new entry. Therefore, Dart *stashes* the new entry in the space that the old entry was occupying until the recirculated old entry comes back. To summarize, every time a new SEQ packet entry contends for space with an old entry due to a hash collision, (1) we allow the new entry to get inserted, (2) evict the old entry, and (3) recirculate the old packet to re-consult the Range Tracker (RT) table to determine its staleness. If stale, we allow the old entry to self-destruct; if not, we treat it as a new packet entry and repeat the above process.

This process is illustrated in Figure 5. The new packet for flow F2 arrives with expected ACK no. 500 at time t=80 (event 1, marked by the white literal in a red circle). It causes the measurement range to expand from [300, 400] to [300, 500] since it is valid (event 2). The corresponding packet record with key (F2, 500) and timestamp value 80 arrives at the PT table and collides with an existing entry (F3, 600, t=40) (event 3). The new entry gets stored (event 4) while the old entry is recirculated to the RT table for re-validation (event 5). The highlight of this mechanism is that it provides *another chance* to the packet that was evicted from the PT table instead of just discarding it right away. A TCP flow with naturally long RTTs, for example, will be preserved by such a mechanism. While recirculations are useful in this case, they add overhead since packet recirculation bandwidth is limited in the data plane. Later in the paper, we discuss a method to reduce recirculations by maintaining a *small cache of heavy flows* after the RT (Section 7).

Preventing infinite eviction loops: One might worry about an eviction loop: the old entry that was kicked out recirculates and gets re-inserted to the same index again, which kicks on the new entry, causing the old entry to get evicted again in the next iteration, and so on. To address this, we implement a method of detecting a "cycle", i.e., when the same entry that got inserted once ends up getting evicted and tries to re-insert itself. The method is to always

compare the new entry with the currently evicted entry before trying re-insertion. In Dart, we do this cycle detection before any recirculation. As another safeguard mechanism, we also set a limit the number of recirculations per SEQ packet.

3.3 Tracking Only *Useful* Samples

The analytics module is a component that can be customized based on the analytics the operator is interested in, using the RTT samples produced by the Range Tracker (RT) and Packet Tracker (PT) tables. In some cases, this analytics module can also help reduce the memory pressure on both tables and also reduce recirculations.

Example: RTT tracking with min-filtering: Consider a use case where an operator is interested in monitoring the propagation delay with hosts in a certain IP prefix. The high-level goal is to detect abnormal changes in the round-trip time, like an upward spike, when communicating with the IP prefix in real time. The operator, however, does not want to get an alert with outliers—only when there is an obvious, consistent hike. A good example is detecting nation-state BGP hijacking, which likely increases the end-to-end delay significantly. One effective way to implement this in the analytics module is to use *min-filtering*: instead of monitoring every single RTT sample, keep track of the minimum RTT value in a time window. If the minimum RTT value significantly increases from one time window to the next one, it is worth notifying the operator through an alert.

Preemptively discard *useless* samples: Now, since the analytics module is only interested in the minimum RTT value within every time window, the system can purge SEQ packets that will surely produce RTT samples that are longer than the currently maintained minimum in the given window. More precisely, when a SEQ packet is kicked out from the PT table, the saved timestamp of that SEQ packets can be compared to the current timestamp to see if it has a potential to produce a sample smaller than the current minimum. If not, there is no need to recirculate this packet entry: it will, at best, produce a *useless* sample anyway. This check can happen at the analytics module *before* the system decides to recirculate the packet back to the RT table. Thus, the analytics module can further limit the resource usage only to the packets that will indeed produce RTT samples that are *useful* to the analytics module.

4 HARDWARE SWITCH PROTOTYPE

In this section, we present our implementation of Dart on the Intel Tofino programmable switches. One version of the prototype runs in the ingress pipeline of a Tofino2 switch, and the other spans the ingress and egress pipelines of a Tofino1 switch. While the Tofino2 version is more efficient and leaves the egress free for network operators to deploy their own analytics and adaptation mechanisms, the Tofino1 version enables us to deploy Dart in our campus testbed. Our code is open-source.[2] The resource usage of the two prototypes is summarized in Table 1. We discuss our hardware implementation challenges and prototype features in this section, and then describe our experience deploying our prototype in the wild in Section 5.

[2]https://github.com/Princeton-Cabernet/dart

Resource Type	Tofino 1	Tofino 2
TCAM	4.9%	2.9%
SRAM	13.9%	1.4%
Hash Units	16.7%	35.8%
Logical Tables	47.9%	36.9%
Input Crossbars	15.4%	10.1%

Table 1: Data Plane Resource Usage in the Tofino (1 and 2)

Accessing memory sequentially: Our implementation requires that actions on the values of RT and PT table records happen sequentially. For example, a RT record is updated only when the flow signature matches with that of an incoming packet; thereafter, we first set the right edge to the maximum of the existing right edge and the incoming eACK, and then compare the eACK and sequence number with the existing right edge to decide the value of the left edge (Figure 4). Since memory once accessed cannot be revisited without recirculation in the data plane, we spread the RT and PT tables each across 3 component tables, and therefore 3 stages.

Constrained signature wordsize: In order to determine with complete accuracy whether a hashed location in the RT or PT contains the intended flow 4-tuple record, we need to store all 12 bytes (4 bytes x 2 for IPv4 addresses + 2 bytes x 2 for TCP port numbers) as part of the record key. However, the wordsize of a register key is constrained in the data plane; therefore, we use hash functions to reduce the flow signature to a fixed 4-byte hash. The downside is that hash collisions are possible (not significantly though, as our results suggest).

Computing the payload size: Computing the TCP payload size in the naive way, i.e., by subtracting the IP and TCP header lengths from the total IP packet length, is expensive in the data plane—it consumes multiple stages due to multiple arithmetic operations involving 32-bit integers. Instead, we pre-compute the TCP payload size for common values of the *IP header length* (5 bytes), the *total IP packet length* (40–1480 bytes), and the *TCP header length* (5–15 bytes) and store them as entries in a lookup table—this saves two Tofino stages. This is purely an optimization and can be easily reversed to support *any* values of the aforementioned header parameters.

TCP sequence number wraparound: TCP sequence numbers can wrap around, i.e., restart from zero. Our prototype detects and handles this case gracefully by resetting the RT left edge to zero. This foregoes the opportunity to collect valid RTT samples at the highest sequence numbers in favor of a simplified implementation.

Reordering among recirculated records: Since packets are processed in a streaming fashion in the data plane, there is a possibility that a more recent packet pertaining to a flow arrives and gets processed by Dart before a recirculated flow record for the same flow has had a chance to make a re-entry into the ingress pipeline. If not handled, this could render certain RTT samples inaccurate. We mitigate this by checking if a recirculated record matches the current flow entry and updating it if so, ensuring only one (the latest) record exists for a flow in the RT at a time.

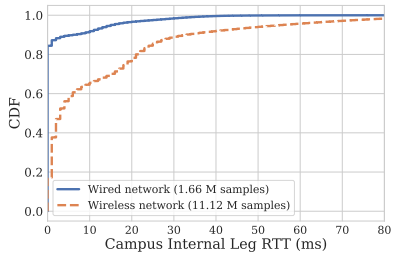


Figure 6: Difference in distribution of internal leg RTTs between wired and wireless subnets in the Princeton campus.

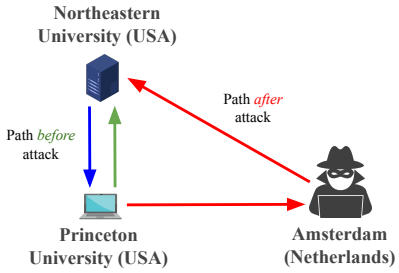


Figure 7: Traffic between Princeton (USA) and Northeastern (USA) intercepted by an attacker in Amsterdam (Netherlands).

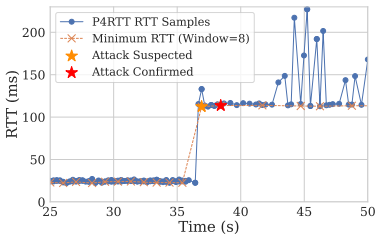


Figure 8: Interception attack is detected within 63 packets by observing change in minimum RTT over windows of samples.

Specifying target flows: Dart allows the operator to install rules to track any subset of flows directly from the control plane. Therefore, it is not necessary to recompile or redeploy Dart to change the set of flows it tracks. The flows can be specified by source and destination IP addresses or prefixes, and source and destination port numbers or port number ranges.

Monitoring the external and internal legs simultaneously: We describe in Section 2.1 how it is possible to compute RTTs on both the *external leg* (i.e., monitor to the Internet) and the *internal leg* (i.e., campus client to monitor). However, monitoring both legs simultaneously in the data plane is a challenge since the same packet has to be processed both as a SEQ and an ACK packet. We are able to do this by first processing the packet as a SEQ packet, but then recirculating it with a custom header that *remembers* the relevant ACK headers (i.e., 4-tuple and ACK number).

5 DART IN THE WILD

In this section, we describe our experience of installing Dart on a hardware switch and replaying real traffic on it.

5.1 RTT Monitoring on Campus Traffic

We run the Tofino1 prototype of Dart on a hardware switch in our lab. We replay an anonymized packet trace collected on the Princeton University campus on 7 April 2020 for 15 minutes (at 3 PM local time). This trace was collected during a period of heavy load with an average rate of 240,750 packets per second. The trace is replayed using tcpreplay from a server connected to the switch. The RTT reports are captured at another server using tcpdump.

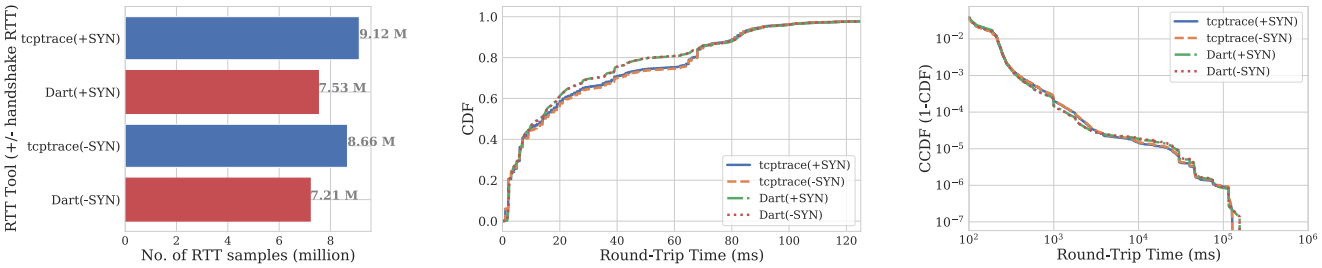
Wired network vs. Wireless network RTT: For this experiment, we replay the trace and monitor the RTTs of the *internal leg* (campus host to switch) for two campus subnets—one *wireless* and another *wired*. Dart collects 11.12M RTT samples for wireless traffic and 1.66M for wired traffic, as most on-campus users connect via mobile devices. Figure 6 shows the difference in the distribution of RTTs across the two subnets. RTTs for wireless connections are uniformly larger than those for the wired connections. More than 80% of internal RTTs for the wired subnet are less than 1 ms, whereas less than 40% of RTTs for the wireless subnet are so; in fact, for wireless traffic, the internal RTT exceeds 20 ms for more than 20% of the samples. Often, the *internal* latency for wireless users rivals the *wide-area* latency. For example, for wireless clients accessing YouTube, the 90th percentile end-to-end RTT is 14 ms, including 8 ms of intra-campus delay. In contrast, the wired clients have a 90th percentile RTT of 9 ms with just 2 ms of intra-campus delay.

5.2 Interception Attack Detection

Next, we demonstrate the ability of Dart to detect a long-distance interception attack in the wild.

Attack setup: First, we launch an ethical traffic-interception attack using the PEERING testbed [29] and following the technique described by Birge-Lee et al. [7]. PEERING runs geographically distributed ASes to enable researchers to make real BGP announcements for controlled and isolated traffic. We control one PEERING site each at Northeastern University and Princeton University in the USA via co-located Amazon AWS server instances (Figure 7). We announce our PEERING-provided /23 prefix from Northeastern, and establish a TCP connection from Princeton with an IP address from the announced prefix. We then use a site at Amsterdam (adversary) to announce a more specific /24 prefix with a number of carefully selected BGP community attributes, such that traffic to the IP at Northeastern (victim) is now rerouted through it. We capture the traffic trace of this attack at Princeton. Our threat model assumes that Dart is close to one of the end-hosts and can *see* both sides of the traffic, both pre-attack and post-attack. Therefore, our detection works irrespective of the direction of traffic (data or ACK) intercepted by the attacker.

Attack detection: Second, we deploy our Dart Tofino1 prototype on a real hardware switch in our campus testbed. We replay the trace of this interception attack through this switch. Dart collects raw RTT samples and sends them to a collection server, where a simple, threshold-based change-detection algorithm runs. The detection algorithm monitors propagation delay by computing the minimum RTT in a window of 8 consecutive raw RTT samples. An attack is *suspected* when the minimum RTT rises abruptly between consecutive windows, but *confirmed* only when the change sustains for another window. Figure 8 illustrates this mechanism: the attack takes effect at t=~36 seconds, as can be observed from the abrupt rise in collected raw RTT from ~25 ms to ~120 ms (blue line). Based on the minimum RTTs (orange line), an attack is *suspected* almost immediately (orange star) and *confirmed* in the next window (red



(a) RTT sample counts **(b) CDF of RTTs** **(c) CCDF of large RTTs**

Figure 9: `tcptrace` vs. `Dart` with infinite memory: `Dart` collects >82% RTTs as `tcptrace` and matches its RTT distribution closely.

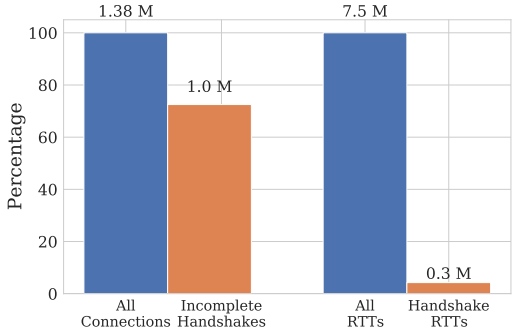


Figure 10: Skipping handshake packets: We can save memory for 72.5% of the connections while missing only 4.2% RTTs.

star). Only 63 packets are exchanged in 2.58 seconds between the interception attack taking effect and `Dart` confirming the attack.

6 EVALUATION

In this section, we evaluate `Dart` using a faithful simulator written in Python. We replay a campus trace collected on April 7, 2020, for this evaluation. The trace was captured using a packet broker service near the campus gateway router. The trace contains 1.38M TCP connections and 135.78M TCP packets over a 15-minute duration. As mentioned in Section 5, `Dart` can measure RTTs for either the *external leg* (monitor to Internet) or the internal leg (campus host to monitor), or both. In the following experiments, we limit our RTT measurement to the external leg only.

6.1 `Dart` Without Memory Constraints

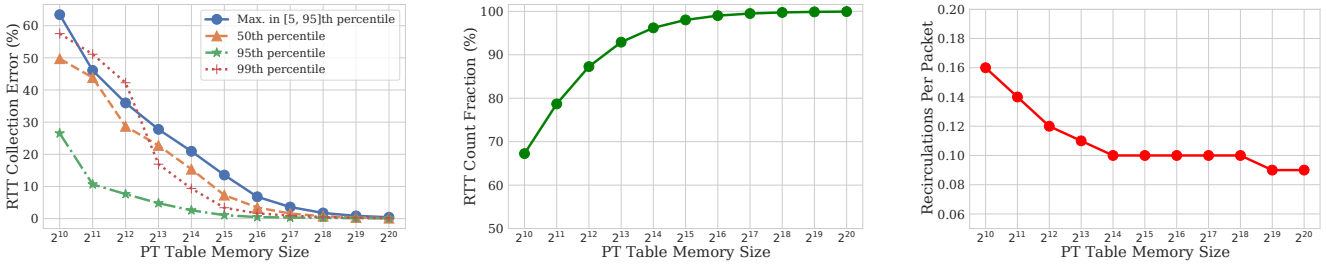
We compare `Dart` with our baseline `tcptrace`—a software tool used to compute RTTs on packet traces [27]. For this experiment, we assume that `Dart` is not bottlenecked by the memory available in RT and PT tables, and that the available memory is fully associative (i.e., records can be stored in any available memory location).

RTT sample count: Figure 9a illustrates the number of RTT samples captured by `Dart` vs. `tcptrace` (*SYN* in the figure is short-hand for packets where the SYN flag is set, i.e., SYN and SYN-ACK). For the case where we collect handshake RTTs, i.e., tcptrace(+SYN) and `Dart`(+SYN), `Dart` is able to capture 7.53M RTT samples vs. 9.12M samples collected by `tcptrace`. The difference is explained by the fact that `Dart` only keeps track of the latest contiguous unambiguous bytes when a *hole* appears in the sequence space, whereas `tcptrace` keeps track of all contiguous byte-ranges. Furthermore, unlike `Dart`, `tcptrace` collects all valid RTTs when sequence numbers wrap around; additionally, `tcptrace` sometimes breaks one RTT sample into two (causing an inaccuracy) due to a design flaw [3]. In the case where handshake RTTs are foregone by ignoring SYN and SYN-ACK packets, i.e., tcptrace(-SYN) and `Dart`(-SYN), `Dart` captures 7.21M RTT samples vs. 8.66M samples captured by `tcptrace`.

RTT distribution: Figures 9b and 9c show the distribution of the RTTs captured in the four settings discussed above (i.e., `tcptrace` vs. `Dart` and +SYN vs. -SYN). Figure 9b shows the distribution (CDF) of RTT samples with values between 0 and 125 ms. Figure 9c focuses on the tail and shows the complementary CDF (CCDF) of RTTs larger than 100 ms. The median RTTs for `tcptrace`(+SYN) vs. `Dart`(+SYN) (14 vs. 13 ms) and for `tcptrace`(-SYN) vs. `Dart`(-SYN) (15 vs. 13 ms) are comparable. There is a skew in the 95^{th} percentile, as can be observed from Figure 9b—`tcptrace`(+SYN) has it at 57 ms vs. 39 ms in `Dart`(+SYN) (similarly, 62 ms in `tcptrace`(-SYN) vs. 39 ms in `Dart`(-SYN)). The distributions converge at the tail, with 99^{th} percentile at 215 ms for both +SYN settings and 218 ms for both -SYN settings. We also observe that the large majority of RTT samples (96.3%) fall between 10 ms and 100 ms (Figure 9b). The tail, however, is long, and RTTs as large as 100 seconds or more are also seen (Figure 9c). Further analysis revealed that there are instances in the trace where a data packet does not receive an ACK for many seconds before a TCP keep-alive finally ACKs it. This may be due to the fact that our vantage point misses the original ACKs (with short RTTs) to these data packets, or a behavior of TCP where only a distant keep-alive acknowledges the last seen data packet. In any case, `Dart` collects the long RTTs that `tcptrace` also collects, demonstrating a lack of bias against large RTTs.

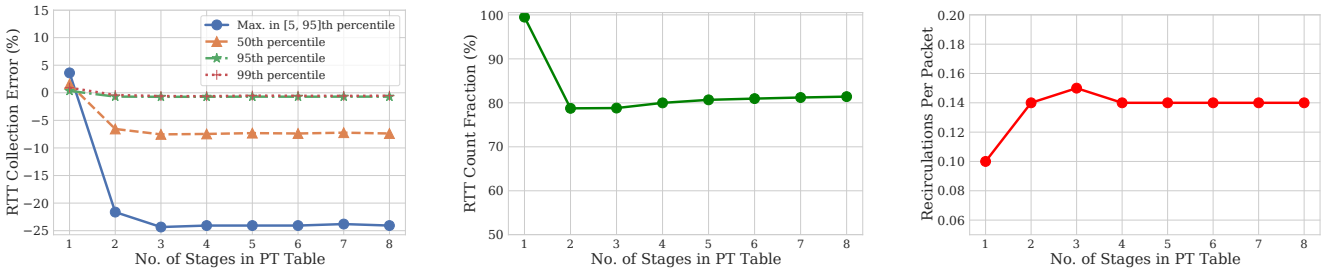
Handshake RTTs: In order to understand the impact of foregoing handshake RTTs (i.e., by ignoring packets with the SYN flag set), we compare the relative memory savings obtained vs. the number of

[3] `tcptrace` divides the sequence space (0 to $2^{31}-1$) into four quadrants. We observed that when an RTT sample is generated for a packet that spans two consecutive quadrants, `tcptrace` wrongly generates an extra RTT sample for the part of the packet that ends in the earlier quadrant.



(a) RTT collection error **(b) Fraction of RTT samples collected** **(c) Recirculations incurred per packet**

Figure 11: Performance of Dart with a large RT table and varying PT table-size.



(a) RTT collection error **(b) Fraction of RTT samples collected** **(c) Recirculations incurred per packet**

Figure 12: Performance of Dart with a large RT, fixed-size PT, and varying no. of PT stages.

samples foregone. Figure 10 illustrates that around 1M out of 1.38M TCP connections (72.5%) seen in the trace are due to incomplete TCP handshakes! Ignoring SYN and SYN-ACK packets, therefore, provides significant advantages in terms of memory in the RT table, since otherwise these flows would occupy much of the table. The relative loss in terms of number of RTTs collected is much less—Dart misses only 0.32M out of 7.53M samples (4.2%).

6.2 Impact of Table Configurations

In the ensuing experiments, we remove the assumptions that the tables are fully associative or of unlimited size. True to the prototype that can run on a hardware switch, the RT and PT tables are now one-way associative (i.e., only one memory location can be accessed per packet without recirculation). As a result, we grapple with contention for memory and recirculations (Section 3).

Baseline: Since Dart(-SYN) from the previous subsection operates with unlimited and a fully associative memory, it is the best Dart can do. The following experiments, therefore, treat Dart(-SYN)'s performance—in terms of the number of RTT samples collected and the resulting RTT distribution—as the baseline. We notice that Dart(-SYN) is actually set up similarly to tcptrace, except for the fact that the amount of state it can track for each flow is *constant*, e.g., one measurement range. For this reason, we view it as a variant of tcptrace with constant space—we refer to it as tcptrace_const.

Performance metrics: We evaluate Dart's accuracy along two dimensions: (1) The closeness of Dart's RTT distribution with tcptrace_const's RTT distribution—quantified by the *RTT collection error*, and (2) The number of RTT samples collected by Dart as compared to the number of RTT samples collected by tcptrace_const—quantified by the *fraction of RTT samples collected*. We define the *RTT collection error* as the *error at the* p^{th} *percentile* for $p = \{50, 95, 99\}$. For a given p, the *error at the* p^{th} *percentile* is computed as the difference between the p^{th} percentile RTT of tcptrace_const and Dart, normalized by the former. In this work, we report the 50^{th} and 95^{th} percentile RTTs since these are considered important latency characterization metrics, and the 99^{th} percentile RTT since it represents Dart's estimation of the largest RTTs collected. We also report the maximum error seen for any p between 5 and 95, which serves as a measure of the *worst-case accuracy*. We define the *fraction of RTT samples collected* as the ratio of the number of RTT samples collected by Dart to the number of RTT samples collected by tcptrace_const (expressed as a percentage). We contrast these two accuracy metrics with the recirculation overhead metric, *recirculations incurred per packet*. It is defined as the ratio of the total number of recirculations incurred to the total number of packets processed. In the following experiments, Dart's *performance* refers to the set of these three metrics.

Impact of the PT table size: In this experiment, we try to determine the *right* size for our one-way associative PT table. First, we set the RT table size to a number *large enough* (i.e., 2^{20} in this specific case) to accommodate all flows in our campus trace. While this number is large, we reason that an operator would only monitor a subset of flows at a time, and therefore, during actual usage, the RT

(a) RTT collection error **(b) Fraction of RTT samples collected** **(c) Recirculations incurred per packet**

Figure 13: Performance of `Dart` with a large RT, a PT with a fixed size and no. of stages, and varying no. of allowed recirculations.

memory requirement would be small enough to fit in one stage of the Tofino (if not, the RT table could be expanded to a multi-stage table). Second, we allow recirculations—we set the maximum allowable recirculations to 1 in the PT table. Now, we vary the PT table size between small ($2^{10} = 1,024$) and large ($2^{20} = 1,048,576$) and evaluate the impact on accuracy of RTT collection. Figure 11a shows the *RTT collection error*. We observe that, as expected, the error goes down as the PT table increases in size. The error at $p = 95$ is the least, followed by the error at $p = 99$, illustrating that `Dart` is not biased against large RTTs. Figure 11b shows that the *fraction of RTT samples collected* increases with increasing memory size, as expected. In fact, `Dart` collects more than 90% samples with a relatively small PT size (i.e., $2^{13} = 8,192$). We also observe from Figure 11c that 16 recirculations happen per 100 packets for a small PT table (i.e., size= 2^{10}), which goes down to 10 and lower as we increase the PT table size. For the next experiment, we fix the PT table size to $2^{17} = 131,072$: this is the lowest PT table size that results in less than 5% error for any value of p between 5 and 95 and collects more than 99% RTT samples.

Impact of the number of PT table stages: We explore the idea of implementing a multi-stage PT table, i.e., a k-way associative PT table. While a multi-stage PT table may be difficult to fit in Tofino1 or in the ingress pipeline of Tofino2, we can implement it in the egress pipeline of Tofino2. We fix the RT table size to *large enough* as before, the PT table size at 2^{17}, and maximum recirculations per packet at 1. We now try to determine the best value of k to divide the PT table into. Figure 12a shows the *RTT collection error* against k. We find that the error at $p = 95$ and $p = 99$ stay stable around zero, whereas the errors at $p = 50$ and the maximum error between $p = 5$ and $p = 95$ increase as soon as we divide the PT table memory into more than 1 stages. The error increases in the negative direction (i.e., `Dart` starts overestimating the median RTT). This can be attributed to the fact that some smaller RTT samples are missed when there are more stages, since there is just enough space for records with large RTTs to get inserted but never get evicted (since older records are preferred). We see a similar effect on the *fraction of RTT samples collected* (in Figure 12b) and the *recirculations incurred per packet* (in Figure 12c)—these metrics get significantly worse when the PT table is divided into 2 stages, and remain bad as the number of stages is gradually increase to 8. We determine from this experiment that simply dividing the PT table into multiple stages, without adding more memory, worsens performance. In the next experiment, we explore whether allowing more than 1 recirculation per packet might help a multi-stage PT.

Impact of the maximum number of allowed recirculations: In this experiment, we again ensure that the RT table is *large enough*, fix the PT table size to 2^{17}, and divide it across 8 stages. This time, we allow the maximum number of recirculations per packet to vary between 1 and 8. Our intuition is that allowing the hash-collided PT records to pass through the PT more times might help them find alternate locations to *live on* in the PT during memory pressure. Figure 13a shows that the RTT collection error rapidly improves as more recirculations are allowed; for a 8-stage PT, 4 recirculations bring down the error to nearly zero. We also see from Figure 13b that the *fraction of RTT samples collected* goes up to 99% and beyond when 4 or more recirculations are allowed. This improvement is achieved without the *number of recirculations incurred per packet* ever going up beyond 0.16, as shown in Figure 13c. We conclude that we can take advantage of spanning the PT across multiple stages of the Tofino—thereby increasing the total memory size beyond what 1 Tofino stage would allow—and still ensure good performance, if we also allow more recirculations per packet. This good performance is achieved without the consumption of significant recirculation bandwidth (16 recirculations per 100 packets in the worst-case).

7 DISCUSSION

Limitations of `Dart`: While `Dart` is designed for accuracy in the face of TCP ambiguities, the tool cannot handle all situations accurately. First, if `Dart` starts (or restarts after eviction) tracking a flow that is already in progress, it may not have enough information to infer retransmission or reordering. For example, what the RT table sees as a new SEQ packet may actually be a retransmitted copy of a SEQ packet sent before `Dart` started tracking the flow. Second, reordering may happen downstream from the monitoring device; in such cases, not just `Dart` but no other RTT measurement tool can measure RTT accurately, since the vantage point is never aware of such an event. Apart from inaccuracies, we may sometimes conservatively close the measurement range when it could have collected valid samples. When we see an ACK packet that acknowledges the left edge of the measurement range, we infer a duplicate ACK and collapse the range. However, since the left edge does not always indicate the highest byte ACKed, but can indicate the highest byte affected by a retransmission or reordering event,

the decision to close the measurement range conservatively may cost us some valid RTT samples. Similarly, we forego valid samples at the uppermost sequence numbers when a sequence number wraparound event happens; however, such events are rare—only 4 in our 15-minute campus trace. Sometimes, when the monitoring device does not see the last ACK in an exchange—either because of a packet drop at the device or a connection that closed abruptly—it may continue holding the RT and PT table records indefinitely, since we favor older entries over newer ones. Certain attacks can exploit this vulnerability by leaving a large number of data packets unacknowledged. A timeout mechanism for the RT table, with a very large timeout value (in seconds) could help.

Minimizing recirculations with approximation: Packet recirculation helps to produce valid samples in the face of memory pressure. Packet recirculation, however, is *additional overhead*. While we realize that some recirculations are unavoidable, those should be minimized. One idea is to maintain a copy of the original RT and put it *after* the PT table. This would allow the validity check to happen at the end of the pipeline instead of recirculating the packet back to the start of the pipeline. Yet, maintaining the consistency between the original RT and the copy is a challenge. Due to the sequential nature of the data plane's packet-processing pipeline, they might become out of sync even for a short amount of time. For example, a new packet might have updated the original RT table to a new state just before an evicted entry reads the second RT table, making the tables inconsistent. This subsequently implies that the copy of the RT is *approximate* in nature. Moreover, such an approach requires additional memory space. Thus, this approach trades recirculation overhead with memory space.

End-host delays: When a TCP receiver anticipates having some data of its own to send, it may wait until a certain number of bytes accumulate and then *piggyback* the ACK along with the data—a strategy known as delayed ACKing. The waiting time is reflected in the RTT samples collected by measurement tools like `Dart` and `tcptrace`. Some use cases may, however, require RTTs that do not include this end-host delay. Delayed ACKs are notoriously difficult to identify by a monitoring device in the end-to-end path. This is because there is no known way to infer whether an ACK has been intentionally delayed by the receiver. Delayed ACK implementation is known to vary widely across device types and operating systems. In fact, the QUIC protocol has proposed provisions to explicitly tag delayed ACKs to alleviate such measurement ambiguities [20]. Still, even without explicit detection of delayed ACKs, simple techniques such as "min. filtering" (e.g., computing the minimum RTT sample over a time window, as discussed in Section 3.3) can be quite effective at separating propagation delay along the network path from end-host delays such as delayed ACKs.

Extending `Dart` to QUIC and IPv6: The QUIC protocol does not expose sequence and acknowledgment numbers as TCP does. Therefore, it is not possible to measure RTTs for QUIC packets using the same mechanism as `Dart`. QUIC, however, uses a *spin bit* to indicate reversal of direction by a sender or receiver [20]. It is possible to track this bit and compute RTTs; however, this is fraught with challenges. First, the spin bit allows the measurement of a maximum of one valid RTT sample per congestion window. Second, inferring retransmissions or reordering is not possible using only the spin bit. Nevertheless, RTTs collected using the QUIC spin bit can augment RTTs collected for TCP traffic to the same destination. `Dart` can also be extended to work with IPv6 by adjusting how the payload size is computed. However, since the 4-tuple size is much larger in IPv6, and the RT flow signature size is fixed, `Dart` may encounter more hash collisions.

Identifying bufferbloat: In our campus trace, we observe some instances of high RTT variability on connections with remote clients. Further analysis reveals that some of these remote hosts were connected to a cellular provider—these hosts are presumably cellular devices connecting to servers located on our campus. The RTT patterns are indicative of *bufferbloat* at the remote end. We find these patterns interesting and posit that `Dart` can be used to detect bufferbloat events in real time, due to its ability to monitor RTTs on-path and continuously. Detecting bufferbloat (and networks prone to it) is useful for characterization from remote locations.

Deployment at multiple on-path vantage points: `Dart` can be deployed at multiple vantage points (VPs) on the route between two end-hosts. The advantage is that the total end-to-end RTT could then be divided into multiple legs (i.e., host 1 to first VP, first to second VP, and so on, leading up to host 2). One use case is capturing wide-area RTTs free from end-host delays like delayed ACKs. Another use case is identifying which part of the network is responsible for performance degradation.

Dealing with optimistic ACKs: Misbehaving TCP receivers can ACK data packets *before* they are received, to manipulate the sender into transmitting faster [28]. `Dart` is largely robust to such *optimistic* ACKs, since it ignores any ACK packet that arrives before the corresponding data packet. In fact, `Dart` can be easily extended to detect and report optimistic ACKs. However, if an optimistic ACK happens to reach `Dart` *after* the corresponding data packet has already arrived at `Dart` (but not at the receiver), `Dart` (and any other passive monitoring tool) would be misled into collecting an incorrect RTT sample. We reason that it is difficult for the receiver to reliably time optimistic ACKs in this way.

8 RELATED WORK

Passive RTT analysis: Previous work has proposed the idea of passive RTT monitoring [18] using SYN/SEQ and ACK packets. `tcptrace` [27] is a passive, open-source TCP traffic-analysis tool that can measure RTT per SEQ/ACK packet pair within a given packet trace. It runs in software on a general-purpose CPU. We use this tool as our ground truth for evaluating `Dart`'s correctness as `tcptrace` is free from time and memory constraints. That said, unlike `tcptrace`, `Dart` can generate RTTs on live traffic in real time. Also, while `tcptrace` rounds RTTs up to the nearest millisecond, the Tofino enables `Dart` to report RTTs down to a nanosecond granularity, which may be useful in extreme low-latency use cases.

Instead of matching data and ACK packets, the *TCP time-stamp* option available in the TCP header can also be used for passive RTT analysis. The `pping` tool [26] adopts this approach. However, TCP timestamps are often too coarse-grained (e.g., 10 or 100 ms granularity), and many services do not use them at all since the TCP timestamp is an optional field [14]. Also, an end-host puts in

its timestamp clock as the field value, but different TCP implementations increment their internal clocks at different rates—some by 100 every second, and others by 1000 [8]. This makes computing the absolute latency in milliseconds troublesome in a monitoring device that does not know the end-host's clock tick rate.

Passive RTT monitoring in the data plane: Dapper [16] is an early work that aims to measure TCP performance in the data plane. However, it can only track a single packet per congestion window when doing RTT measurement; it has to wait until the corresponding ACK arrives before starting to track another packet's RTT. In use cases where RTTs are large, or aggregate statistics (e.g., min., median, etc.) are being collected over time-windows, Dapper would report too few samples per unit time to be useful. Chen et al. [12] expanded Dapper's idea and proposed an algorithm and data structure that can track multiple SEQ packets simultaneously in the data plane. However, this previous work does not correctly deal with ambiguities in TCP, such as packet retransmission and ordering. It also uses the memory space inefficiently and produces RTT samples biased against long RTTs. We use this previous work as our strawman in Section 2. Zheng et al. [32] proposed a novel data structure called *fridges* to measure delay directly in the data plane with the focus on collecting RTT samples without bias against larger delays. This is done by applying a correction factor inversely proportional to the probability of producing an RTT sample. This previous work, however, does not address the accuracy and memory efficiency issues caused by TCP ambiguities.

Liu et al. [23] proposed algorithms that run in the data plane for four separate performance-monitoring tasks: round-trip latency, packet loss detection, out-of-order detection, and retransmission detection. This previous work focuses more on using memory sublinear in both the size of input data and the number of flows, and the four performance-monitoring algorithms are independent of each other. Also, the round-trip latency is *not* calculated in a per-packet matching scheme; rather, it computes the average RTT by subtracting the sum of all server-to-client (SEQ) packets' timestamps from the sum of all client-to-server (ACK) packets' timestamps and then computes the average, assuming no missing or duplicate SEQ or ACK packets. RouteScout [4] measures packet loss and delay in a programmable data plane to help select a better Internet path driven by performance in real time. However, RouteScout measures round-trip time only at TCP handshake using the first SYN and ACK exchange. Thus, RouteScout only produces one RTT sample per flow during connection establishiment. In contrast, `Dart` focuses on accurate, continuous round-trip time monitoring on a per-packet basis while taking into account the idiosyncrasies of TCP, including packet loss, reordering, and retransmission.

9 CONCLUSION

Round-trip time (RTT) is a critical metric in network performance monitoring, whether for tracking the QoE of latency-sensitive applications (e.g., in cloud gaming) or for detecting malicious attacks (e.g., nation-state traffic interception attacks). Performing RTT monitoring passively in real time in the data plane opens up new opportunities, especially because this allows the hardware switch to take immediate action on packets based on RTT values. At the same time, this means that RTTs should be computed with utmost accuracy even when faced with the idiosyncrasies of the TCP protocol. This is particularly difficult due to the memory and packet processing constraints in hardware switches. To this end, we present `Dart`, a novel algorithm and data structure implemented in P4 that fits and runs in a data plane with a Tofino1 or Tofino2 chipset. Even when running in a data plane with strict constraints, `Dart` still closely matches the accuracy of the widely used offline analysis tools that run on servers with abundant memory. We open source our implementation. We hope `Dart` will open up new opportunities for building applications that can benefit from `Dart`'s ability to compute accurate RTTs in real time directly in the data plane.

Acknowledgements: We thank our shepherd, Boon Thau Loo, and the anonymous paper reviewers for their insightful feedback. We thank the anonymous artifact reviewers for their feedback as well. We thank Xiaoqi (Danny) Chen whose earlier work on Tofino-based RTT monitoring inspired this work. We also thank Henry Birge-Lee and Daniel Jubas for helping us set up the PEERING experiments. We thank Princeton University's Office of Information Technology, Office of Institutional Research, and the Institutional Review Board for enabling us to evaluate our work with anonymized campus traffic. This work is supported by the NSF grant CNS-1704077 and the DARPA grant HR0011-20-C-0107.

REFERENCES

[1] 2021. NVIDIA Mellanox NIC's Performance Report with DPDK 21.05. *http://fast.dpdk.org/doc/perf/DPDK_21_05_Mellanox_NIC_performance_report.pdf* (2021).
[2] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino2: A 12.9 Tbps P4-Programmable Ethernet Switch. In *IEEE Hot Chips Symposium (HCS)*. IEEE Computer Society, 1–32.
[3] Aditya Akella, Jeffrey Pang, Bruce Maggs, Srinivasan Seshan, and Anees Shaikh. 2004. A comparison of overlay routing and multihoming route control. *ACM SIGCOMM Computer Communication Review* 34, 4 (2004), 93–106.
[4] Maria Apostolaki, Ankit Singla, and Laurent Vanbever. 2021. Performance-Driven Internet Path Selection. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*. 41–53.
[5] Axel Arnbak and Sharon Goldberg. 2014. Loopholes for circumventing the constitution: Unrestricted bulk surveillance on americans by collecting network traffic abroad. *Michigan Telecommunications and Technology Law Review* 21 (2014), 317.
[6] Debopam Bhattacherjee, Muhammad Tirmazi, and Ankit Singla. 2017. A cloud-based content gathering network. In *USENIX Workshop on Hot Topics in Cloud Computing*.
[7] Henry Birge-Lee, Liang Wang, Jennifer Rexford, and Prateek Mittal. 2019. Sico: Surgical interception attacks by manipulating BGP communities. In *ACM SIGSAC Conference on Computer and Communications Security*. 431–448.
[8] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. 2014. *TCP Extensions for High Performance*. RFC 7323. RFC Editor.
[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[10] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. 2019. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. In *ACM SIGMETRICS*. 1–25.
[11] Kuan-Ta Chen, Yu-Chun Chang, Po-Han Tseng, Chun-Ying Huang, and Chin-Laung Lei. 2011. Measuring the latency of cloud gaming systems. In *Proceedings of the 19th ACM international conference on Multimedia*. 1269–1272.
[12] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. 2020. Measuring TCP round-trip time in the data plane. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*. 35–41.
[13] Yunhua Deng, Yusen Li, Xueyan Tang, and Wentong Cai. 2016. Server allocation for multiplayer cloud gaming. In *Proceedings of the 24th ACM international conference on Multimedia*. 918–927.
[14] Hao Ding and Michael Rabinovich. 2015. TCP stretch acknowledgements and timestamps: Findings and implications for passive RTT measurement. *ACM SIGCOMM Computer Communication Review* 45, 3 (2015), 20–27.
[15] Jon Dugan, Seth Elliott, Bruce A Mah, Jeff Poskanzer, and Kaustubh Prabhu. 2014. iperf3, tool for active measurements of the maximum achievable bandwidth on

IP networks. (2014). https://github.com/esnet/iperf.
[16] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data plane performance diagnosis of TCP. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*. ACM, 61–74.
[17] Nicholas Hopper, Eugene Y Vasserman, and Eric Chan-Tin. 2010. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC)* 13, 2 (2010), 1–28.
[18] Hao Jiang and Constantinos Dovrolis. 2002. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review* 32, 3 (2002), 75–88.
[19] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*. 90–106.
[20] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The QUIC transport protocol: Design and internet-scale deployment. In *ACM SIGCOMM*. 183–196.
[21] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. 2015. Accurate latency-based congestion feedback for datacenters. In *USENIX Annual Technical Conference (ATC)*. 403–415.
[22] Sanghwan Lee, Zhi-Li Zhang, and Srihari Nelakuditi. 2004. Exploiting as hierarchy for scalable route selection in multi-homed stub networks. In *ACM Internet Measurement Conference*. 294–299.
[23] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. 2020. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems (APoCS)*. SIAM, 31–44.
[24] Giovane CM Moura, John Heidemann, Wes Hardaker, Pithayuth Charnsethikul, Jeroen Bulten, Joao Ceron, and Cristian Hesselman. 2022. Old but Gold: Prospecting TCP to Engineer and Real-time Monitor DNS Anycast. In *Passive and Active Measurement Conference*.
[25] RIPE NCC. 2021. RIPE Atlas. https://atlas.ripe.net/. (2021).
[26] Kathleen Nichols. 2017. pping (Pollere passive ping). https://github.com/pollere/pping. (2017).
[27] Shawn Ostermann. 2007. tcptrace Homepage. *http://www.tcptrace.org/* (2007).
[28] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. 1999. TCP congestion control with a misbehaving receiver. *ACM SIGCOMM Computer Communication Review* 29, 5 (1999), 71–78.
[29] Brandon Schlinker, Todd Arnold, Italo Cunha, and Ethan Katz-Bassett. 2019. PEERING: Virtualizing BGP at the Edge for Research. In *ACM SIGCOMM International Conference on Emerging Networking Experiments And Technologies*. 51–67.
[30] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2021. Fine-Grained RTT Monitoring Inside the Network. *Measuring Network Quality for End-Users* (2021).
[31] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. 2015. RAPTOR: Routing attacks on privacy in Tor. In *USENIX Security Symposium*. 271–286.
[32] Yufei Zheng, Xiaoqi Chen, Mark Braverman, and Jennifer Rexford. 2022. Unbiased Delay Measurement in the Data Plane. In *Symposium on Algorithmic Principles of Computer Systems (APoCS)*. SIAM, 15–30.