PINT: Probabilistic In-band Network Telemetry

Ran Ben Basat
Harvard University
ran@seas.harvard.edu

Sivaramakrishnan Ramanathan
University of Southern California
satyaman@usc.edu

Yuliang Li
Harvard University
yuliangli@g.harvard.edu

Gianni Antichi
Queen Mary University of London
g.antichi@qmul.ac.uk

Minlan Yu
Harvard University
minlanyu@seas.harvard.edu

Michael Mitzenmacher
Harvard University
michaelm@eecs.harvard.edu

ABSTRACT

Commodity network devices support adding in-band telemetry measurements into data packets, enabling a wide range of applications, including network troubleshooting, congestion control, and path tracing. However, including such information on packets adds significant overhead that impacts both flow completion times and application-level performance.

We introduce *PINT*, an in-band network telemetry framework that bounds the amount of information added to each packet. *PINT* encodes the requested data on multiple packets, allowing per-packet overhead limits that can be as low as one bit. We analyze *PINT* and prove performance bounds, including cases when multiple queries are running simultaneously. *PINT* is implemented in P4 and can be deployed on network devices.Using real topologies and traffic characteristics, we show that *PINT* concurrently enables applications such as congestion control, path tracing, and computing tail latencies, using only sixteen bits per packet, with performance comparable to the state of the art.

CCS CONCEPTS

• **Networks** → **Network protocols**; **Network algorithms**; **Network monitoring**;

KEYWORDS

Network Telemetry, Networking Algorithms, Networking Protocols

ACM Reference Format:
Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. 2020. *PINT*: Probabilistic In-band Network Telemetry. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3387514.3405894

1 INTRODUCTION

Network telemetry is the basis for a variety of network management applications such as network health monitoring [72], debugging [28], fault localization [6], resource accounting and planning [56], attack detection [27, 65], congestion control [46], load balancing [2, 41, 42], fast reroute [47], and path tracing [36]. A significant recent advance is provided by the In-band Network Telemetry (INT) [75]. INT allows switches to add information to each packet, such as switch ID, link utilization, or queue status, as it passes by. Such telemetry information is then collected at the network egress point upon the reception of the packet.

INT is readily available in programmable switches and network interface cards (NICs) [8, 14, 58, 85], enabling an unprecedented level of visibility into the data plane behavior and making this technology attractive for real-world deployments [16, 46]. A key drawback of INT is the overhead on packets. Since each switch adds information to the packet, the packet byte overhead grows linearly with the path length. Moreover, the more telemetry data needed per-switch, the higher the overhead is: on a generic data center topology with 5 hops, requesting two values per switch requires 48 Bytes of overhead, or 4.8% of a 1000 bytes packet (§2). When more bits used to store telemetry data, fewer bits can be used to carry the packet payload and stay within the maximum transmission unit (MTU). As a result, applications may have to split a message, e.g., an RPC call, onto multiple packets, making it harder to support the run-to-completion model that high-performance transport and NICs need [7]. Indeed, the overhead of INT can impact application performance, potentially leading in some cases to a 25% increase and 20% degradation of flow completion time and goodput, respectively (§2). Furthermore, it increases processing latency at switches and might impose additional challenges for collecting and processing the data (§2).

We would like the benefits of in-band network telemetry, but at smaller overhead cost; in particular, we wish to minimize the per-packet bit overhead. We design Probabilistic In-band Network Telemetry (*PINT*), a probabilistic variation of INT, that provides similar visibility as INT while bounding the per-packet overhead according to limits set by the user. *PINT* allows the overhead budget to be as low as one bit, and leverages approximation techniques to meet it. We argue that often an approximation of the telemetry data suffices for the consuming application. For example, telemetry-based congestion control schemes like HPCC [46] can be tuned to work with approximate telemetry, as we demonstrate in this paper. In some use cases, a single bit per packet suffices.

With *PINT*, a query is associated with a maximum overhead allowed on each packet. The requested information is probabilistically encoded onto several different packets so that a *collection* of a flow's packets provides the relevant data. In a nutshell, while with INT a query triggers every switch along the path to embed their own information, *PINT* spreads out the information over multiple packets to minimize the per-packet overhead. The insight behind this approach is that, for most applications, it is not required to know

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA

ACM ISBN 978-1-4503-7955-7/20/08.
https://doi.org/10.1145/3387514.3405894

Metadata value	Description
Switch ID	ID associated with the switch
Ingress Port ID	Packet input port
Ingress Timestamp	Time when packet is received
Egress Port ID	Packet output port
Hop Latency	Time spent within the device
Egress Port TX utilization	Current utilization of output port
Queue Occupancy	The observed queue build up
Queue Congestion Status	Percentage of queue being used

Table 1: Example metadata values.

all of the per-packet-per-hop information that INT collects. existing techniques incur high overheads due to requiring perfect telemetry information. For applications where some imperfection would be sufficient, these techniques may incur unnecessary overheads. *PINT* Is designed for precisely such applications For example, it is possible to check a flow's path conformance [30, 56, 72], by inferring its path from a collection of its packets. Alternatively, congestion control or load balancing algorithms that rely on latency measurements gathered by INT, e.g., HPCC [46], Clove [41] can work if packets convey information about the path's bottleneck, and do not require information about all hops.

We present the *PINT* framework (§3) and show that it can run several concurrent queries while bounding the per-packet bit overhead. To that end, *PINT* uses each packet for a query subset with cumulative overhead within the user-specified budget. We introduce the techniques we used to build this solution (§4) alongside its implementation on commercial programmable switches supporting P4 (§5). Finally, we evaluate (§6) our approach with three different use cases. The first traces a flow's path, the second uses data plane telemetry for congestion control, and the third estimates the experienced median/tail latency. Using real topologies and traffic characteristics, we show that *PINT* enables all of them concurrently, with only sixteen bits per packet and while providing comparable performance to the state of the art.

In summary, the main contributions of this paper are:

- We present *PINT*, a novel in-band network telemetry approach that provides fine-grained visibility while bounding the per-packet bit overhead to a user-defined value.
- We analyze *PINT* and rigorously prove performance bounds.
- We evaluate *PINT* in on path tracing, congestion control, and latency estimation, over multiple network topologies.
- We open source our code [1].

2 INT AND ITS PACKET OVERHEAD

INT is a framework designed to allow the collection and reporting of network data plane status at switches, without requiring any control plane intervention. In its architectural model, designated INT traffic sources, (e.g., the end-host networking stack, hypervisors, NICs, or ingress switches), add an *INT metadata header* to packets. The header encodes *telemetry instructions* that are followed by network devices on the packet's path. These instructions tell an INT-capable device what information to add to packets as they transit the network. Table 1 summarizes the supported *metadata values*. Finally, INT traffic sinks, e.g., egress switches or receiver hosts, retrieve the collected results before delivering the original packet to the application. The INT architectural model is intentionally generic, and hence can enable a number of high level applications, such as (1) Network troubleshooting and verification, i.e., microburst detection [36], packet history [30], path tracing [36], path latency computation [34]; (2) Rate-based congestion control, i.e., RCP [22], XCP [40], TIMELY [53]; (3) Advanced routing, i.e, utilization-aware load balancing [2, 42].

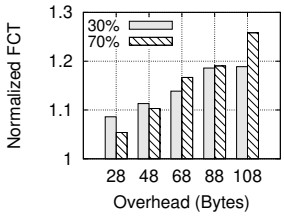


Figure 1: Normalized average Flow Completion Time varying the network load and increasing the per-packet overhead.

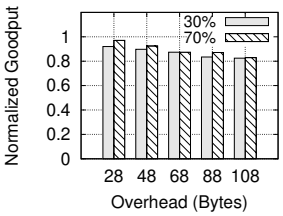


Figure 2: Normalized average goodput of long flows (>10MB) varying the network load and increasing per-packet overhead.

INT imposes a non insignificant overhead on packets though. The metadata header is defined as an 8B vector specifying the telemetry requests. Each value is encoded with a 4B number, as defined by the protocol [75]. As INT encodes per-hop information, the overall overhead grows linearly with both the number of metadata values and the number of hops. For a generic data center topology with 5 hops, the minimum space required on packet would be 28 bytes (only one metadata value per INT device), which is 2.8% of a 1000 byte packet (e.g., RDMA has a 1000B MTU). Some applications, such as Alibaba's High Precision Congestion Control [46] (HPCC), require three different INT telemetry values for each hop. Specifically, for HPCC, INT collects timestamp, egress port tx utilization, and queue occupancy, alongside some additional data that is not defined by the INT protocol. This would account for around 6.8% overhead using a standard INT on a 5-hop path.[1] This overhead poses several problems:

1. High packet overheads degrade application performance. The significant per-packet overheads from INT affect both flow completion time and application-level throughput, i.e., goodput. We ran an NS3 [76] experiment to demonstrate this. We created a 5-hop fat-tree data center topology with 64 hosts connected through 10Gbps links. Each host generates traffic to randomly chosen destinations with a flow size distribution that follows a web search workload [3]. We employed the standard ECMP routing with TCP Reno. We ran our experiments with a range of packet overheads from 28B to 108B. The selected overheads correspond to a 5-hop topology, with one to five different INT values collected at each hop. Figure 1 shows the effect of increasing overheads on the average flow completion time (FCT) for 30% (average) and 70% (high) network utilization. Figure 2, instead, focuses on the goodput for only the long flows, i.e., with flow size >10 MBytes. Both graphs are normalized to the case where no overhead is introduced on packets.

In the presence of 48 bytes overhead, which corresponds to 3.2% of a 1500B packet (e.g., Ethernet has a 1500B MTU), the average FCT increases by 10%, while the goodput for long flows degrades by 10% if network utilization is approximately 70%. Further increasing the overhead to 108B (7.2% of a 1500B packet) leads to a 25% increase and 20% degradation of flow completion time and

[1] HPCC reports a slightly lower (4.2%) overhead because they use customized INT. For example, they do not use the INT header as the telemetry instructions do not change over time.

Application	Description	Measurement Primitives
Per-packet aggregation		
Congestion Control [22, 29, 40, 46]	Congestion Control with in-network support	timestamp, port utilization, queue occupancy
Congestion Analysis [17, 38, 57]	Diagnosis of short-lived congestion events	queue occupancy
Network Tomography [26]	Determine network state, i.e., queues status	switchID, queue occupancy
Power Management [31]	Determine under-utilized network elements	switchID, port utilization
Real-Time Anomaly Detection [66, 86]	Detect sudden changes in network status	timestamp, port utilization, queue occupancy
Static per-flow aggregation		
Path Tracing [36, 56, 65, 72]	Detect the path taken by a flow or a subset	switchID
Routing Misconfiguration [45, 69, 72]	Identify unwanted path taken by a given flow	switchID
Path Conformance [45, 69, 73]	Checks for policy violations.	switchID
Dynamic per-flow aggregation		
Utilization-aware Routing [2, 41, 42]	Load balance traffic based on network status.	switchID, port utilization
Load Imbalance [45, 65, 73]	Determine links processing more traffic.	switchID, port utilization
Network Troubleshooting [36, 57, 73]	Determine flows experiencing high latency.	switchID, timestamp

Table 2: Use cases enabled by PINT, organized per aggregation mode.

goodput, respectively. This means that even a small amount of bandwidth headroom can provide a dramatic reduction in latency [4]. The long flows' average goodput is approximately proportional to the residual capacity of the network. That means, at a high network utilization, the residual capacity is low, so the extra bytes in the header cause larger goodput degradation than the byte overhead itself [4]. As in our example, the theoretical goodput degradation should be around $1 - \frac{100\%-70\%*1.072}{100\%-70\%*1.032} \approx 10.1\%$ when increasing the header overhead from 48B to 108B at around 70% network utilization. This closely matches the experiment result, and is much larger than the extra byte overhead (4%).

Although some data center networks employ jumbo frames to mitigate the problem[2], it is worth noting that (1) not every network can employ jumbo frames, especially the large number of enterprise and ISP networks; (2) some protocols might not entirely support jumbo frames; for example, RDMA over Converged Ethernet NICs provides an MTU of only 1KB [54].

2. Switch processing time. In addition to consuming bandwidth, the INT overhead also affects packet processing time at switches. Every time a packet arrives at and departs from a switch, the bits carried over the wire need to be converted from serial to parallel and vice versa, using the 64b/66b (or 66b/64b) encoding as defined by the IEEE Standard 802.3 [33]. For this reason, any additional bit added into a packet affects its processing time, delaying it at both input and output interfaces of every hop. For example, adding 48 bytes of INT data on a packet (INT header alongside two telemetry information) would cause a latency increase with respect to the original packet of almost 76*ns* and 6*ns* for 10G and 100G interfaces, respectively[3]. On a state-of-the-art switch with 10G interfaces, this can represent an approximately 3% increase in processing latency [60]. On larger topologies and when more telemetry data is needed, the overhead on the packet can cause an increase of latency in the order of microseconds, which can hurt the application performance [61].

3. Collection overheads. Telemetry systems such as INT generate large amounts of traffic that may overload the network. Additionally, INT produces reports of varying size (depending on the number of hops), while state-of-the-art end-host stack processing systems for telemetry data, such as Confluo [43], rely on *fixed-byte size* headers on packets to optimize the computation overheads.

[2] https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/network_mtu.html

[3] Consuming 48Bytes on a 10G interface requires 6 clock cycles each of them burning 6.4 ns [83]. On a 100G interface, it needs just one clock cycle of 3ns [84].

3 THE *PINT* FRAMEWORK

We now discuss the supported functionalities of our system, formalizing the model it works in.

Telemetry *Values*. In our work, we refer to the telemetry information as *values*. Specifically, whenever a packet p_j reaches a switch s, we assume that the switch observes a value $v(p_j, s)$. The value can be a function of the switch (e.g., port or switch ID), switch state (e.g., timestamp, latency, or queue occupancy), or any other quantity computable in the data plane. In particular, our definition supports the information types that INT [75] can collect.

3.1 Aggregation Operations

We design *PINT* with the understanding that collecting all (per-packet per-switch) values pose an excessive and unnecessary overhead. Instead, *PINT* supports several aggregation operations that allow efficient encoding of the aggregated data onto packets. For example, congestion control algorithms that rely on the bottleneck link experienced by packets (e.g., [46]) can use a per-packet aggregation. Alternatively, applications that require discovering the flow's path (e.g., path conformance) can use per-flow aggregation.

- **Per-packet aggregation** summarizes the data across the different values in the packet's path, according to an *aggregation function* (e.g., max/min/sum/product). For example, if the packet p_j traverses the switches $s_1, s_2, \ldots, s_k$ and we perform a max-aggregation, the target quantity is $\max \left\{v(p_j, s_i)\right\}_{i=1}^{k}$.
- ***Static* per-flow aggregation** targets summarizing values that may differ between flows or switches, but are fixed for a (flow, switch) pair. Denoting the packets of flow x by $p_1, \ldots, p_z$, the static property means that for any switch s on x's path we have $v(p_1, s) = \ldots = v(p_z, s)$; for convenience, we denote $v(x, s) \triangleq v(p_1, s)$. If the path taken by x is $s_1, \ldots, s_k$, the goal of this aggregation is then to compute all values on the path, i.e., $v(x, s_1), v(x, s_2), \ldots, v(x, s_k)$. As an example, if $v(x, s_i)$ is the ID of the switch s_i, then the aggregation corresponds to inferring the flow's path.
- ***Dynamic* per-flow aggregation** summarizes, for each switch on a flow's path, the stream of values observed by its packets. Denote by $p_1, \ldots, p_z$ the packets of x and by $s_1, \ldots, s_k$ its path, and let

sequence of values measured by s_i on x's packets be denoted as $S_{x,i} = \langle v(p_1, s_i), v(p_2, s_i), \ldots v(p_z, s_i)\rangle$. The goal is to compute a function of $S_{i,x}$ according to an aggregation function (e.g., median or number of values that equal a particular value v). For example, if $v(p_j, s_i)$ is the latency of the packet p_j on the switch s_i, using the median as an aggregation function equals computing the median latency of flow x on s_i.

3.2 Use Cases

PINT can be used for a wide variety of use cases (see Table 2). In this paper, we will mainly discuss three of them, chosen in such a way that we can demonstrate all the different *PINT* aggregations in action.

Per-packet aggregation: Congestion Control. State of the art congestion control solutions often use INT to collect utilization and queue occupancy statistics [46]. *PINT* shows that we can get similar or better performance while minimizing the overheads associated with collecting the statistics.

Static per-flow aggregation: Path Tracing. Discovering the path taken by a flow is essential for various applications like path conformance [45, 69, 73]. In *PINT*, we leverage multiple packets from the same flow to infer its path. For simplicity, we assume that each flow follows a single path.

Dynamic per-flow aggregation: Network Troubleshooting. For diagnosing network issues, it is useful to measure the latency quantiles from each hop [17, 34, 36, 57]. Tail quantiles are reported as the most effective way to summarize the delay in an ISP [19]. For example, we can detect network events in real-time by noticing a change in the hop latency [9]. To that end, we leverage *PINT* to collect the median and tail latency statistics of (switch, flow) pairs.

3.3 Query Language

Each query in *PINT* is defined as a tuple ⟨val_t, agg_t, bit-budget, ***optional***: space-budget, flow definition, frequency⟩ that specifies which values are used (e.g., switch IDs or latency), the aggregation type as in Section 3.1, and the *query bit-budget* (e.g., 8 bits per packet). The user may also specify a space-budget that determines how much per-flow storage is allowed, the flow-definition (e.g., 5-tuple, source IP, etc.) in the case of per-flow queries, and the query frequency (that determines which fraction of the packets should be allocated for the query).

PINT works with *static* bit-budgets to maximize its effectiveness while remaining transparent to the sender and receiver of a packet. Intuitively, when working with INT/PINT one needs to ensure that a packet's size will not exceed the MTU even after the telemetry information is added. For example, for a 1500B network MTU, if the telemetry overhead may add to X bytes, then the sender would be restricted to sending packets smaller than 1500−X. Thus, by fixing the budget, we allow the network flows to operate without being aware of the telemetry queries and path length.

3.4 Query Engine

PINT allows the operator to specify multiple queries that should run concurrently and a *global bit-budget*. For example, if the global bit-budget is 16 bits, we can run two 8-bit-budget queries on the same packet. In *PINT*, we add to packets a *digest* – a short bitstring whose length equals the global bit budget. This digest may compose of multiple *query digests* as in the above example.

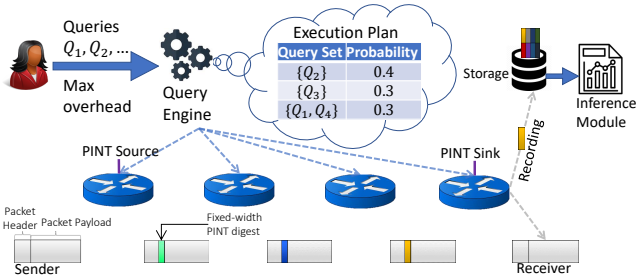


Figure 3: *PINT*'s architecture: The Query Engine decides on an execution plan that determines the probability of running each query set on packets and notifies the switches. The first hop, *PINT* Source, adds a digest whose size is determined by the user. Every switch along the path may modify the digest but does not add bits. The last hop, *PINT* Sink, removes the collected telemetry information and sends it to the Recording Module. On demand, the Inference Module is invoked to analyze the recorded data.

Each query instantiates an *Encoding Module*, a *Recording Module*, and an *Inference Module*. The Encoding runs on the switches and modifies the packet's *digest*. When a packet reaches a *PINT* Sink (the last hop on its path), the sink extracts (removes) the digest and sends the data packet to its destination. This way, *PINT* remains transparent to both the sender and receiver. The extracted digest is intercepted by the Recording Module, which processes and stores the digests. We emphasize that the per-flow data stored by the Recording Module sits in an offline storage and no per-flow state is stored on the switches. Another advantage of *PINT* is that, compared with INT, we send fewer bytes from the sink to be analyzed and thereby reduce the network overhead. The Inference Module runs on a commodity server that uses the stored data to answer queries. Fig. 3 illustrates *PINT*'s architecture.

Importantly, all switches must agree on which query set to run on a given packet, according to the distribution chosen by the Query Engine. We achieve coordination using a global hash function, as described in Section 4.1. Unlike INT, we do not add a telemetry header; in this way we minimize the bit overhead.[4] Instead, the *PINT* Query Engine compiles the queries to decide on the execution plan (which is a probability distribution on a query set, see Fig. 3) and notifies the switches.

3.5 Challenges

We now discuss several challenges we face when designing algorithms for *PINT*.

Bit constraints. In some applications, the size of values may be prohibitively large to a point where writing a single value on each packet poses an unacceptable overhead.

Switch Coordination. The switches must agree on which query set to use for each packet. While the switches can communicate by exchanging bits that are added to packets, this increases the bit-overhead of *PINT* and should be avoided.

Switch constraints. The hardware switches have constraints, including limited operations per packet, limited support for arithmetic operations (e.g., multiplication is not supported), inability

[4] We note that removing the header is minor compared to the overhead saving *PINT* obtains by avoiding logging all per-hop values.

to keep per-flow state, etc. See [12] for a discussion of the constraints. For *PINT*, these constraints mean that we must store minimal amount of state on switches and use simple encoding schemes that adhere to the programmability restrictions.

4 AGGREGATION TECHNIQUES

In this section, we present the techniques used by *PINT* to overcome the above challenges. We show how global hash functions allow efficient coordination between different switches and between switches and the Inference Module. We also show how distributed encoding schemes help reduce the number of packets needed to collect the telemetry information. Finally, we adopt compression techniques to reduce the number of bits required to represent numeric values (e.g., latency).

Our techniques reduce the bit-overhead on packets using probabilistic techniques. As a result, some of our algorithms (e.g., latency quantile estimation) are *approximate*, while others (e.g., path tracing) require *multiple packets* from the same flow to decode. Intuitively, oftentimes one mostly cares about tracing large (e.g., malicious) flows and does not require discovering the path of very short ones. Similarly, for network diagnostics it is OK to get approximated latency measurements as we usually care about large latencies or significant latency changes. We summarize which techniques apply for each of the use cases in Table 3.

Use Case	Global Hashes	Distributed Coding	Value Approximation
Congestion Control	✗	✗	✓
Path Tracing	✓	✓	✗
Latency Quantiles	✓	✗	✓

Table 3: A summary of which techniques are used for each use case.

4.1 Implicit Coordination via Global Hash Functions

In *PINT*, we extensively use global hash functions to determine probabilistic outcomes at the switches. As we show, this solves the switch coordination challenge, and also enables implicit coordination between switches and the Inference Module – a feature that allows us to develop efficient algorithms.

Coordination among switches. We use a global (i.e., that is known to all switches) hash function to determine which query set the current packet addresses. For example, suppose that we have three queries, each running with probability 1/3, and denote the query-selection hash, mapping packet IDs to the real interval[5] $[0, 1]$, by $\mathfrak{q}$. Then if $\mathfrak{q}(p_j) < 1/3$, all switches would run the first query, if $\mathfrak{q}(p_j) \in [1/3, 2/3]$ the second query, and otherwise the third. Since all switches compute the same $\mathfrak{q}(p_j)$, they agree on the executed query without communication. This approach requires the ability to derive unique packet identifiers to which the hashes are applied (e.g., IPID, IP flags, IP offset, TCP sequence and ACK numbers, etc.). For a discussion on how to obtain identifiers, see [21].

Coordination between switches and Inference Module. The Inference Module must know which switches modified an incoming packet's digest, but we don't want to spend bits on encoding switch IDs in the packet. Instead, we apply a global hash function g on a (packet ID, hop number)[6] pair to choose whether to act on a packet. This enables the *PINT* Recording Module to compute g's outcome for all hops on a packet's path and deduct where it was modified. This coordination plays a critical role in our per-flow algorithms as described below.

Example #1: Dynamic Per-flow aggregation. In this aggregation, we wish to collect statistics from values that vary across packets, e.g., the median latency of a (flow, switch) pair. We formulate the general problem as follows: Fix some flow x. Let $p_1, \ldots, p_z$ denote the packets of x and $s_1, \ldots, s_k$ denote its path. For each switch s_i, we need to collect enough information about the sequence $S_{i,x} = \langle v(p_1, s_i), v(p_2, s_i), \ldots v(p_z, s_i)\rangle$ while meeting the query's bit-budget. For simplicity of presentation, we assume that packets can store a single value.[7]

PINT's Encoding Module runs a distributed sampling process. The goal is to have each packet carry the value of a uniformly chosen hop on the path. That is, each packet p_j should carry each value from $\{v(p_j, s_1), \ldots, v(p_j, s_k)\}$ with probability $1/k$. This way, with probability $1 - e^{-\Omega(z/k)}$, each hop will get $z/k \cdot (1 \pm o(1))$ samples, i.e., almost an equal number.

To get a uniform sample, we use a combination of global hashing and the Reservoir Sampling algorithm [82]. Specifically, when the i'th hop on the path (denoted s_i) sees a packet p_j, it overwrites its digest with $v(p_j, s_i)$ *if* $g(p_j, i) \leq r_i$. Therefore, the packet will end up carrying the value $v(p_j, s_i)$ only if (i) $g(p_j, i) \leq r_i$, and (ii) $\forall_J \in \{i+1, \ldots, k\} : g(p_j, J) > r_J$. To get uniform sampling, we follow the Reservoir Sampling algorithm and set $r_i \triangleq 1/i$. Indeed, for each hop (i) and (ii) are simultaneously satisfied with probability $1/k$. Intuitively, while later hops have a lower chance of overriding the digest, they are also less likely to be replaced by the remaining switches along the path.

Intuitively, we can then use existing algorithms for constructing statistics from subsampled streams. That is, for each switch s_i, the collected data is a uniformly subsampled stream of $S_{i,x}$. One can then apply different aggregation functions. For instance, we can estimate quantiles and find frequently occurring values. As an example, we can estimate the median and tail latency of the (flow, switch) pair by finding the relevant quantile of the subsampled stream.

On the negative side, aggregation functions like the number of distinct values or the value-frequency distribution entropy are poorly approximable from subsampled streams [49].

PINT aims to minimize the decoding time and amount of per-flow storage. To that end, our Recording Module does not need to store all the incoming digests. Instead, we can use a sketching algorithm that suits the target aggregation (e.g., a quantile sketch [39]). That is, for each switch s_i through which flow x is routed, we apply a sketching algorithm to the sampled substream of $S_{i,x}$. If given a per-flow space budget (see §3.3) we split it between the k sketches evenly. This allows us to record a smaller amount of per-flow information and process queries faster. Further, we can use a sliding-window

[5] For simplicity, we consider hashing into real numbers. In practice, we hash into M bits (the range $\{0, \ldots, 2^M - 1\}$) for some integer M (e.g., $M = 64$). Checking if the real-valued hash is in $[a, b]$ corresponds to checking if the discrete hash is in the interval $\left[\lfloor (2^M - 1) \cdot a \rfloor, \lfloor (2^M - 1) \cdot b \rfloor\right]$.

[6] The hop number can be computed from the current TTL on the packet's header.

[7] If the global bit-budget does not allow encoding a value, we compress it at the cost of an additional error as discussed in Section 4.3. If the budget allows storing multiple values, we can run the algorithm independently multiple times and thereby collect more information to improve the accuracy.

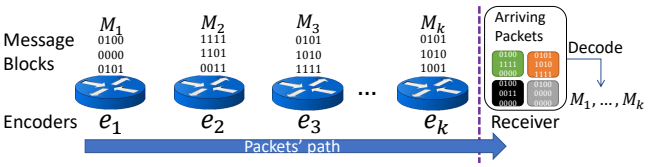


Figure 4: Multiple encoders send a distributed message.

sketch (e.g., [5, 11, 13]) to reflect only the most recent measurements. Finally, the Inference Module uses the sketch to provide estimates on the required flows.

The accuracy of *PINT* for dynamic aggregation depends on the aggregation function, the number of packets (z), the length of the path (k), and the per-flow space stored by the Recording Module (which sits off-switch in remote storage). We state results for two typical aggregation functions. The analysis is deferred to Appendix A.1.

THEOREM 1. *Fix an error target $\varepsilon \in (0,1)$ and a target quantile $\phi \in (0,1)$ (e.g., $\phi = 0.5$ is the median). After seeing $O(k\varepsilon^{-2})$ packets from a flow x, using $O(k\varepsilon^{-1})$ space, PINT produces a $(\phi \pm \varepsilon)$-quantile of $S_{x,i}$ for each hop i.*

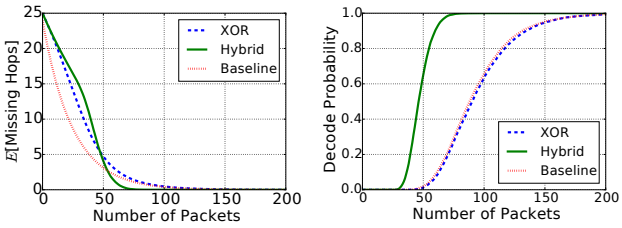
THEOREM 2. *Fix an error target $\varepsilon \in (0,1)$ and a target threshold $\theta \in (0,1)$. After seeing $O(k\varepsilon^{-2})$ packets from a flow x, using $O(k\varepsilon^{-1})$ space, PINT produces all values that appear in at least a θ-fraction of $S_{x,i}$, and no value that appears less than a $(\theta - \varepsilon)$-fraction, for each hop i.*

4.2 Distributed Coding Schemes

When the values are static for a given flow (i.e., do not change between packets), we can improve upon the dynamic aggregation approach using *distributed encoding*. Intuitively, in such a scenario, we can spread each value $v(x, s_i)$ over multiple packets. The challenge is that the information collected by *PINT* is not known to any single entity but is rather distributed between switches. This makes it challenging to use existing encoding schemes as we wish to avoid adding extra overhead for communication between switches. Further, we need a simple encoding scheme to adhere to the switch limitations, and we desire one that allows efficient decoding.

Traditional coding schemes assume that a single encoder owns all the data that needs encoding. However, in *PINT*, the data we wish to collect can be distributed among the network switches. That is, the message we need to transfer is partitioned between the different switches along the flow's path.

We present an encoding scheme that is fully distributed without any communication between encoders. Specifically, we define our scheme as follows: a *sequence* of k encoders hold a k-block message $M_1, \ldots, M_k$ such that encoder e_i has M_i for all $i \in \{1, \ldots, k\}$. The setting is illustrated in Fig. 4. Each packet carries a digest which has a number of bits that equals the block size and has a unique identifier which distinguishes it from other packets. Additionally, each encoder is aware of its hop number (e.g., by computing it from the TTL field in the packet header). The packet starts with a digest of $\bar{0}$ (a zero bitstring) and passes through $e_1, \ldots, e_k$. Each encoder can modify the packet's digest before passing it to the next encoder. After the packet visits e_k, it is passed to the Receiver, which tries to decode the message. We assume that the encoders are *stateless* to model the switches' inability to keep a per-flow state in networks.



(a) Algorithm Progress (b) Probability of Decoding

Figure 5: The XOR scheme (with prob. $1/d$) decodes fewer hops at first but is able to infer the entire path using a similar number of packets to Baseline. By interleaving both schemes (Hybrid), we get a better result as the first hops are mainly decoded by Baseline packets and the last hops by XOR packets that have XOR probability $\log\log d/\log d$ and are more likely to hit the missing hops. Plotted for $d = k = 25$ hops.

Our main result is a distributed encoding scheme that needs $k \cdot \log\log^* k \cdot (1 + o(1))$ packets for decoding the message with near-linear decoding time. We note that Network Coding [32] can also be adapted to this setting. However, we have found it rather inefficient, as we explain later on.

Baseline Encoding Scheme. A simple and intuitive idea for a distributed encoding scheme is to carry a uniformly sampled block on each packet. That is, the encoders can run the Reservoir Sampling algorithm using a global hash function to determine whether to write their block onto the packet. Similarly to our Dynamic Aggregation algorithm, the Receiver can determine the hop number of the sampling switch, by evaluating the hash function, and report the message.

The number of packets needed for decoding the message using this scheme follows the Coupon Collector Process (e.g., see [24]), where each block is a coupon and each packet carries a random sample. It is well-known that for k coupons, we would need $k \ln k(1 + o(1))$ samples on average to collect them all. For example, for $k = 25$, Coupon Collector has a median (i.e., probability of 50% to decode) of 89 packets and a 99'th percentile of 189 packets, as shown in Fig. 5.

The problem with the Baseline scheme is that while the first blocks are encoded swiftly, later ones require a higher number of packets. The reason is that after seeing most blocks, every consecutive packet is unlikely to carry a new block. This is because the encoders are unaware of which blocks were collected and the probability of carrying a new block is proportional to number of missing blocks. As a result, the Baseline scheme has a long "tail", meaning that completing the decoding requires many packets.

Distributed XOR Encoding. An alternative to the Baseline scheme is to use bitwise-xor while encoding. We avoid assuming that the encoders know k, but assume that they know a *typical length* d, such that $d = \Theta(k)$. Such an assumption is justified in most cases; for example, in data center topologies we often know a tight bound on the number of hops [72]. Alternatively, the median hop count in the Internet is estimated to be 12 [80], while only a few paths have more than 30 hops [15, 77]. The XOR encoding scheme has a parameter p, and each encoder on the path bitwise-xors its message onto the packet's digest with probability $p = 1/d$, according to the global hash function. That is, the i'th encoder changes the digest if $g(p_j, i) < p$. We note that this probability is uniform and that the decision of

whether to xor is independent for each encoder, allowing a distributed implementation without communication between the encoders.

When a packet reaches the Receiver, the digest is a bitwise-xor of multiple blocks $M_{i_1} \oplus \ldots \oplus M_{i_K}$, where K is a binomial random variable $K \sim Bin(k, p)$. The Receiver computes $g(p_j, 1), \ldots, g(p_j, k)$ to determine the values $i_1, \ldots, i_K$. If this set contains exactly one unknown message block, we can discover it by bitwise-xoring the other blocks. For example, if we have learned the values of M_1, M_3, M_4, M_6 and the current digest is $p_j.\text{dig} = M_1 \oplus M_5 \oplus M_6$, we can derive M_5 since $M_5 = p_j.\text{dig} \oplus M_1 \oplus M_6$.

On its own, the XOR encoding does not asymptotically improve over the Baseline. Its performance is optimized when $p = 1/d = \Theta(1/k)$, where it requires $O(k \log k)$ packets to decode, i.e., within a constant factor from the Baseline's performance. Interestingly, we show that the combination of the two approaches gives better results.

Interleaving the Encoding Schemes. Intuitively, the XOR and Baseline schemes behave differently. In the Baseline, the chance of learning the value of a message block with each additional packet *decreases* as we receive more blocks. In contrast, to recover data from an XOR packet, we need to know all xor-ed blocks but one. When p is much larger than $1/k$, many packet digests are modified by multiple encoders, which means that the probability to learn a message block value *increases* as we decode more blocks.

As an example for how the interleaved scheme helps, consider the case of $k = 2$ encoders. The Baseline scheme requires three packets to decode the message in expectation; the first packet always carries an unknown block, but each additional packet carries the missing block with probability only $1/2$. In contrast, suppose each packet chooses the Baseline scheme and the XOR scheme each with probability $1/2$, using $p = 1$. For the interleaved scheme to complete, we need either two Baseline packets that carry different blocks or one XOR packet and one Baseline packet. A simple calculation shows that this requires just $8/3$ packets in expectation.

For combining the schemes, we first choose whether to run the Baseline with probability τ, or the XOR otherwise. Once again, switches make the decision based on a global hash function applied to the packet identifier to achieve implicit agreement on the packet type. Intuitively, the Baseline scheme should reduce the number of undecoded blocks from k to k', and the XOR will decode the rest. To minimize the number of packets, we can set $\tau = 3/4$ and the XOR probability[8] to $\log\log d/\log d$ to reduce the required number of packets to $O(k \log\log k/\log\log\log k)$. In such setting, the Baseline decodes most hops, leaving $k' \approx k/\log k$ for the XOR layer. For example, when $k = 25$, we get a median of 41 packets and a 99'th percentile of 68 packets to decode the message. That is, not only does it improve the average case, the interleaving has sharper tail bounds. This improvement is illustrated in Fig. 5.

Multi-layer Encoding. So far, we used a single probability for xor-ing each packet, which was chosen inversely proportional to k' (the number of hops that were not decoded by the Baseline scheme). This way, we maximized the probability that a packet is xor-ed by exactly one of these k' blocks, and we xor any block from the $k - k'$ that are known already to remove them from the decoding. However, when most of the k' blocks left for XOR are decoded, it also "slows down" and requires more packets for decoding each additional block. Therefore, we propose to use multiple XOR *layers* that vary in their sampling probabilities. We call the Baseline scheme layer 0, and the XOR layers $1, \ldots, \mathcal{L}$. Each XOR layer $\ell \in \{1, \ldots, \mathcal{L}\}$ starts with k_ℓ undecoded blocks, xors with probability p_ℓ, and ends when $k_{\ell+1}$ blocks are undecoded.

Our analysis, given in Appendix A.2, shows that by optimizing the algorithm parameters τ, $\mathcal{L}$, $\{k_\ell\}_{\ell=1}^{\mathcal{L}}$ and $\{p_\ell\}_{\ell=1}^{\mathcal{L}}$, we obtain the following result. The value of $\mathcal{L}$ is a function of d, and we have that $\mathcal{L} = 1$ if $d \leq \lfloor e^e \rfloor = 15$ and $\mathcal{L} = 2$ if $16 \leq d \leq e^{e^e}$; i.e., in practice we need only one or two XOR layers.

THEOREM 3. *After seeing $k \log\log^* k(1+o(1))$ packets, the Multi-layer scheme can decode the message.*

We note that the $o(1)$ term hides an $O(k)$ packets additive term, where the constant depends on how well d approximates k. Namely, when $d = k$, our analysis indicates that $k(\log\log^* k+2+o(1))$ packets are enough. Finally, we note that if d is not representative of k at all, we still get that $k \ln k(1 + o(1))$ packets are enough, the same as in the Baseline scheme (up to lower order terms). The reason is that our choice of τ is close to 1, i.e., only a small fraction of the packets are used in the XOR layers.

Comparison with Linear Network Coding. Several algorithms can be adapted to work in the distributed encoding setting. For example, Linear Network Coding (LNC) [32] allows one to decode a message in a near-optimal number of packets by taking random linear combinations over the message blocks. That is, on every packet, each block is xor-ed into its digest with probability $1/2$. Using global hash functions to select which blocks to xor, one can determine the blocks that were xor-ed onto each digest. LNC requires just $\approx k + \log_2 k$ packets to decode the message. However, in some cases, LNC may be suboptimal and *PINT* can use alternative solutions. First, the LNC decoding algorithm requires matrix inversion which generally takes $O(k^3)$ time in practice (although theoretically faster algorithms are possible). If the number of blocks is large, we may opt for approaches with faster decoding. Second, LNC does not seem to work when using hashing to reduce the overhead. As a result, in such a setting, LNC could use fragmentation, but may require a larger number of packets than the XOR-based scheme using hashing.

Example #2: Static Per-flow Aggregation. We now discuss how to adapt our distributed encoding scheme for *PINT*'s static aggregation. Specifically, we present solutions that allow us to reduce the overhead on packets to meet the bit-budget in case a single value cannot be written on a packet. For example, for determining a flow's path, the values may be 32-bit switch IDs, while the bit-budget can be smaller (even a single bit per packet). We also present an implementation variant that allows to decode the collection of packets in near-linear time. This improves the quadratic time required for computing $\{g(p_j, i)\}$ for all packets p_j and hops i.

Reducing the Bit-overhead using Fragmentation. Consider a scenario where each value has $\mathtt{q}$ bits while we are allowed to have smaller $\mathtt{b}$-bits digests on packets. In such a case, we can break each value into $F \triangleq \lceil \mathtt{q}/\mathtt{b} \rceil$ *fragments* where each has $\leq \mathtt{b}$ bits. Using an additional global hash function, each packet p_j is associated with a *fragment number* in $\{1, \ldots, F\}$. We can then apply our distributed encoding scheme separately on each fragment number. While fragmentation reduces the bit overhead, it also increases the number of

[8] If $d \leq 15$ then $\log\log d < 1$; in this case we set the probability to $1/\log d$.

packets required for the aggregation, and the decode complexity, as if there were $k \cdot F$ hops.

Reducing the Bit-overhead using Hashing. The increase in the required number of packets and decoding time when using fragmentation may be prohibitive in some applications. We now propose an alternative that allows decoding with fewer packets, if the value-set is restricted. Suppose that we know in advance a small set of possible block values $\mathcal{V}$, such that any M_i is in $\mathcal{V}$. For example, when determining a flow's path, $\mathcal{V}$ can be the set of switch IDs in the network. Intuitively, the gain comes from the fact that the keys may be longer than $\log_2 |\mathcal{V}|$ bits (e.g., switch IDs are often 32-bit long, while networks have much fewer than 2^{32} switches). Instead of fragmenting the values to meet the q-bits query bit budget, we leverage hashing. Specifically, we use another global hash function h that maps (value, packet ID) pairs into q-bit bitstrings. When encoder e_i sees a packet p_j, if it needs to act it uses $h(M_i, p_j)$ to modify the digest. In the Baseline scheme e_i will write $h(M_i, p_j)$ on p_j, and in the XOR scheme it will xor $h(M_i, p_j)$ onto its current digest. As before, the Recording Module checks the hop numbers that modified the packet. The difference is in how the Inference Module works – for each hop number i, we wish to find a single value $v \in \mathcal{V}$ that agrees with all the Baseline packets from hop i. For example, if p_1 and p_2 were Baseline packets from hop i, M_i must be a value such that $h(M_i, p_1) = p_1.\text{dig}$ and $h(M_i, p_2) = p_2.\text{dig}$. If there is more than one such value, the inference for the hop is not complete and we require additional packets to determine it. Once a value of a block M_i is determined, from any digest p_j that was xor-ed by the i'th encoder, we xor $h(M_i, p_j)$ from $p_j.\text{dig}$. This way, the number of unknown blocks whose hashes xor-ed p_j decreases by one. If only one block remains, we can treat it similarly to a Baseline packet and use it to reduce the number of potential values for that block. Another advantage of the hashing technique is that it does not assume anything about the width of the values (e.g., switch IDs), as long as each is distinct.

Reducing the Decoding Complexity. Our description of the encoding and decoding process thus far requires processing is super-quadratic ($\omega(k^2)$) in k. That is because we need $\approx k \log\log^* k$ packets to decode the message, and we spend $O(k)$ time per packet in computing the g function to determine which encoders modified its digest. We now present a variant that reduces the processing time to nearly linear in k. Intuitively, since the probability of changing a packet is $\Omega(1/k)$, the number of random bits needed to determine which encoders modify it is $O(k \log k)$. Previously, each encoder used the global function g to get $O(\log k)$ pseudo-random bits and decide whether to change the packet. Instead, we can use g to create $O(\log 1/p) = O(\log k)$ pseudo-random k-bit vectors. Intuitively, each bit in the bitwise-and of these vectors will be set with probability p (as defined by the relevant XOR layer). The i'th encoder will modify the packet if the i'th bit is set in the bitwise-and of the vectors[9]. At the Recording Module, we can compute the set of encoders that modify a packet in time $O(\log k)$ by drawing the random bits and using their bitwise-and. Once we obtain the bitwise-and vector we can extract a list of set bits in time $O(\#\text{set bits})$ using bitwise operations. Since the average number of set bits is $O(1)$, the overall per-packet complexity remains $O(\log k)$ and the total decoding time becomes $O(k \log k \log\log^* k)$. We note that this improvement assumes that k fits in $O(1)$ machine words (e.g., $k \leq 256$) and that encoders can do $O(\log k)$ operations per packet.

Improving Performance via Multiple Instantiations. The number of packets *PINT* needs to decode the message depends on the query's bit-budget. However, increasing the number of bits in the hash may not be the best way to reduce the required number of packets. Instead, we can use multiple independent repetitions of the algorithm. For example, given an 8-bit query budget, we can use two independent 4-bit hashes.

4.3 Approximating Numeric Values

Encoding an exact numeric value on packet may require too many bits, imposing an undesirable overhead. For example, the 32-bit latency measurements that INT collects may exceed the bit-budget. We now discuss to compress the value, at the cost of introducing an error.

Multiplicative approximation. One approach to reducing the number of bits required to encode a value is to write on the packet's digest $\mathfrak{a}(p_j, s) \triangleq \left\lfloor \log_{(1+\varepsilon)^2} v(p_j, s) \right\rceil$ instead of $v(p_j, s)$. Here, the $[\cdot]$ operator rounds the quantity to the closest integer. At the Inference Module, we can derive a $(1+\varepsilon)$-approximation of the original value by computing $(1+\varepsilon)^{2 \cdot \mathfrak{a}(p_j, s)}$. For example, if we want to compress a 32-bit value into 16 bits, we can set $\varepsilon = 0.0025$.

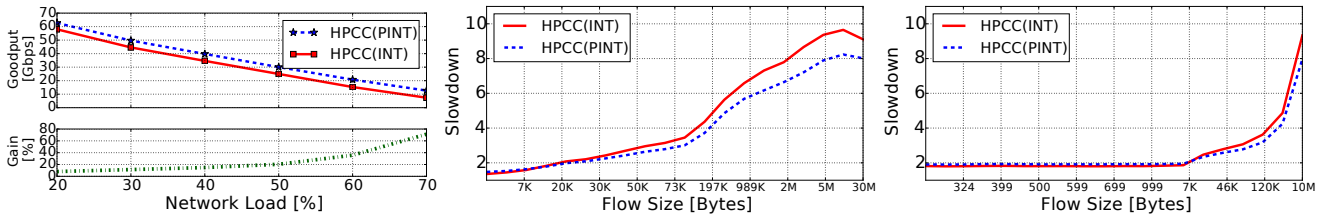
Additive approximation. If distinguishing small values is not as crucial as bounding the maximal error, we obtain better results by encoding the value with additive error instead of multiplicative error. For a given error target Δ (thereby reducing the overhead by $\lfloor \log_2 \Delta \rfloor$ bits), the Encoding Module writes $\mathfrak{a}(p_j, s) \triangleq \left\lfloor \frac{v(p_j, s)}{2\Delta} \right\rceil$, and the Inference Module computes $(2\Delta) \cdot \mathfrak{a}(p_j, s)$.

Randomized counting. For some aggregation functions, the aggregation result may require more bits than encoding a single value. For example, in a per-packet aggregation over a k-hop path with q-bit values, the sum may require $q + \log k$ bits to write explicitly while the product may take $q \cdot k$ bits. This problem is especially evident if q is small (e.g., a single bit specifying whether the latency is high). Instead, we can take a randomized approach to increase the value written on a packet probabilistically. For example, we can estimate the number of high-latency hops or the end-to-end latency to within a $(1+\varepsilon)$-multiplicative factor using $O(\log \varepsilon^{-1} + \log\log(2^q \cdot k \cdot \varepsilon^2)))$ bits [55].

Example #3: Per-packet aggregation. Here, we wish to summarize the data across the different values in the packet's path. For example, HPCC [46] collects per-switch information carried by INT data, and adjusts the rate at the end host according to the highest link utilization along the path. To support HPCC with *PINT*, we have two key insights: (1) we just need to keep the highest utilization (i.e., the bottleneck) in the packet header, instead of every hop; (2) we can use the multiplicative approximation to further reduce the number of bits for storing the utilization. Intuitively, *PINT* improves HPCC as it reduces the overheads added to packets, as explained in Section 2.

In each switch, we calculate the utilization as in HPCC, with slight tuning to be supported by switches (discussed later). The multiplication is calculated using log and exp based on lookup tables [67]. The result is encoded using multiplicative approximation. To further eliminate systematic error, we write $\mathfrak{a}(p_j, s) \triangleq \left\lfloor \log_{(1+\varepsilon)^2} v(p_j, s) \right\rfloor_R$,

[9] This assumes that the probability is a power of two, or provides a $\sqrt{2}$ approximation of it. By repeating the process we can get a better approximation.



(a) Web search workload (large flows) (b) Web search workload (c) Hadoop workload

Figure 7: Comparison of the 95th-percentile slowdown of the standard INT-based HPCC and the *PINT*-based HPCC. *PINT* improves the performance for the long flows due to its reduced overheads. In (b) and (c), the network load is 50% and the x-axis scale is chosen such that there are 10% of the flows between consecutive tick marks.

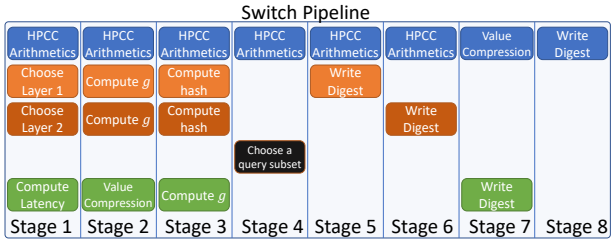


Figure 6: Layout illustration for two path tracing hashes, alongside a latency query, and a congestion control query.

the $[\cdot]_R$ *randomly* performs floor or ceiling, with a probability distribution that gives an expected value equals to $\log_{(1+\varepsilon)^2} v(p_j, s)$. This way, some packets will overestimate the utilization while others underestimate it, thus resulting in the correct value on average. In practice, we just need 8 bits to support $\varepsilon = 0.025$.

Tuning HPCC calculation for switch computation. We maintain the exponential weighted moving average (EWMA) of link utilization U of each link in the switch. U is updated on every packet with: $U = \frac{T-\tau}{T} \cdot U + \frac{\tau}{T} \cdot u$, where $u = \frac{qlen}{B \cdot T} + \frac{byte}{B \cdot \tau}$ is the new sample for updating U. Here, T is the base RTT and B is the link bandwidth (both are constants). Intuitively, the weight of the EWMA, $\frac{\tau}{T}$, corresponds to each new packet's time occupation τ. The calculation of u also corresponds to each new packet: *byte* is the packet's size, and *qlen* is the queue length when the packet is dequeued[10].

To calculate the multiplications, we first do the following transformation: $U = \frac{T-\tau}{T} \cdot U + \frac{qlen \cdot \tau}{B \cdot T^2} + \frac{byte}{B \cdot T}$. Then we calculate the multiplications using logarithm and exponentiation as detailed in Appendix B.

5 IMPLEMENTATION

PINT is implemented using the P4 language and can be deployed on commodity programmable switches. We explain how each of our use cases is executed.

For running the path tracing application (static per-flow aggregation), we require four pipeline stages. The first chooses a layer, another computes g, the third hashes the switch ID to meet the query's bit budget, and the last writes the digest. If we use more than one hash for the query, both can be executed in parallel as they are independent.

[10]This is slightly different from HPCC, where the calculation is done in the host, which can only see packets of its own flow. Therefore, the update is scaled for packets of the same flow (τ is time gap between packets of the same flow, and *byte* includes the bytes from other flows in between). Here, the update is performed on all packets on the same link. Since different flows may interleave on the link, our calculation is more fine-grained.

Computing the median/tail latency (dynamic per-flow aggregation) also requires four pipeline stages: one for computing the latency, one for compressing it to meet the bit budget; one to compute g; and one to overwrite the value if needed.

Our adaptation of the HPCC congestion control algorithm requires six pipeline stages to compute the link utilization, followed by a stage for approximating the value and another to write the digest. For completeness, we elaborate on how to implement in the data plane the different arithmetic operations needed by HPCC in Appendix C. We further note that running it may require that the switch would need to perform the update of U in a single stage. In other cases, we propose to store the last n values of U on separate stages and update them in a round-robin manner, for some integer n. This would mean that our algorithm would need to recirculate every n'th packet as the switch's pipeline is one-directional.

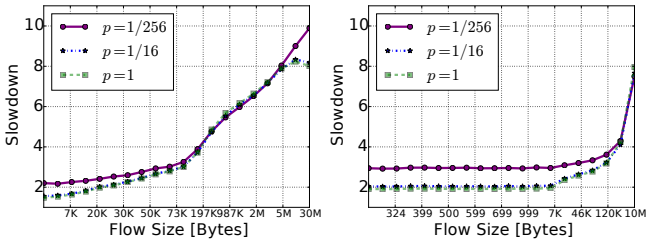
Since the switches have a limited number of pipeline stages, we parallelize the processing of queries as they are independent of each other. We illustrate this parallelism for a combination of the three use cases of *PINT*.We start by executing all queries simultaneously, writing their results on the packet vector. Since HPCC requires more stages than the other use cases, we concurrently compute which query subset to run according to the distribution selected by the Query Engine (see §3.4). We can then write the digests of all the selected queries without increasing the number of stages compared with running HPCC alone. The switch layout for such a combination is illustrated in Fig. 6.

6 EVALUATION

We evaluate on the three use cases discussed on §3.2.

6.1 Congestion Control

We evaluate how *PINT* affects the performance of HPCC [46] using the same simulation setting as in [46]. Our goal is not to propose a new congestion control scheme, but rather to present a low-overhead approach for collecting the information that HPCC utilizes. We use NS3 [76] and a FatTree topology with 16 Core switches, 20 Agg switches, 20 ToRs, and 320 servers (16 in each rack). Each server has a single 100Gbps NIC connected to a single ToR. The capacity of each link between Core and Agg switches, as well as Agg switches and ToRs, are all 400Gbps. All links have a 1μs propagation delay, which gives a 12μs maximum base RTT. The switch buffer size is 32MB. The traffic is generated following the flow size distribution in web search from Microsoft [3] and Hadoop from Facebook [62]. Each server generates new flows according to a Poisson process, destined to random servers. The average flow arrival time is set



(a) Web search workload (b) Hadoop workload

Figure 8: The 95th-percentile slowdown of running *PINT*-based HPCC (at 50% network load) on p-fraction of the packets. On both workloads, the performance of running it on $1/16$ of the packets produces similar results to running it on all.

so that the total network load is 50% (not including the header bytes). We use the recommended setting for HPCC: $W_{AI} = 80$ bytes, $maxStage = 0$, $\eta = 95\%$, and $T = 13\mu$s.

The results, depicted in Fig. 7(b) and Fig. 7(c), show that *PINT* has similar performance (in terms of slowdown) to HPCC, despite using just 8 bits per packet. Here, *slowdown* refers to the ratio between the completion time of the flow in the presence of other flows and alone. Specifically, *PINT* has better performance on long flows while slightly worse performance on short ones. The better performance on long flows is due to *PINT*'s bandwidth saving. Fig. 7(a) shows the relative goodput improvement, averaged over all flows over 10MB, of using *PINT* at different network load. At higher load, the byte saving of *PINT* brings more significant improvement. For example, at 70% load, using *PINT* improves the goodput by 71%. This trend aligns with our observation in §2.

To evaluate how the congestion control algorithm would perform alongside other queries, we experiment in a setting where only a $p = 1, 1/16, 1/256$ fraction of the packets carry the query's digest. As shown in Fig. 8(a) and Fig. 8(b), the performance only slightly degrades for $p = 1/16$. This is expected, because the bandwidth-delay product (BDP) is 150 packets, so there are still 9.4 ($\approx$150/16) packets per RTT carrying feedback. Thus the rate is adjusted on average once per 1/9.4 RTT (as compared to 1/150 RTT with per-packet feedback), which is still very frequent. With $p = 1/256$, the performance of short flows degrades significantly, because it takes longer than an RTT to get feedback. The implication is that congestion caused by long flows is resolved slowly, so the queue lasts longer, resulting in higher latency for short flows. The very long flows (e.g., > 5MB) also have worse performance. The reason is that they are long enough to collide with many shorter flows, so when the competing shorter flows finish, the long flows have to converge back to the full line rate. With $p = 1/256$, it takes much longer time to converge than with smaller p.

In principle, the lower feedback frequency p only affects the convergence speed as discussed above, but not the stability and fairness. Stability is guaranteed by no overreaction, and HPCC's design of reference window (constant over an RTT) provides this regardless of p. Fairness is guaranteed by additive-increase-multiplicative-decrease (AIMD), which is preserved regardless of p.

6.2 Latency Measurements

Using the same topology and workloads as in our congestion control experiments, we evaluate *PINT*'s performance on estimating latency

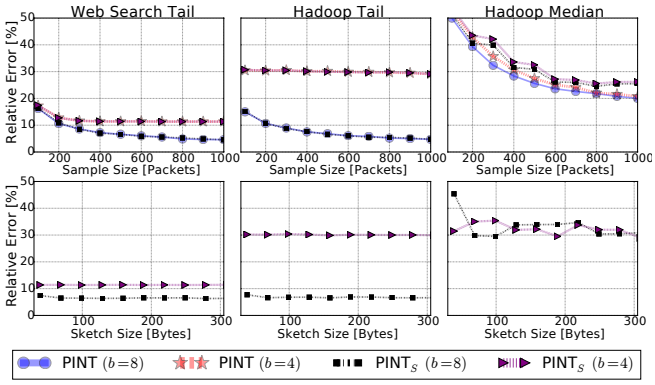


Figure 9: *PINT* error on estimating latency quantiles with a sketch (PINT_S) and without. In the first row, the sketch has 100 digests; in the second, the sample has 500 packets.

quantiles. We consider *PINT* in four scenarios, using $b = 4$ and $b = 8$ bit-budgets, with sketches (denoted $PINT_S$), and without. In our experiment, we have used the, state of the art, KLL sketch [39]. The results, appearing in Fig. 9, show that when getting enough packets, the error of the aggregation becomes stable and converges to the error arising from compressing the values. As shown, by compressing the incoming samples using a sketch (e.g., that keeps 100 identifiers regardless of the number of samples), *PINT* accuracy degrades only a little even for small 100B sketches. We conclude that such sketches offer an attractive space to accuracy tradeoff.

6.3 Path Tracing

We conduct these experiments on Mininet [52] using two large-diameter (denoted D) ISP topologies (Kentucky Datalink and US Carrier) from Topology Zoo [44] and a ($K = 8$) Fat Tree topology. The Kentucky Datalink topology consisted of 753 switches with a diameter of 59 and the US carrier topology consisted of 157 switches with a diameter of 36. For each topology and every path, we estimate the average and 99'th percentile number of packets needed for decoding over 10K runs. We consider three variants of *PINT*– using 1-bit, 4-bit, and two independent 8-bit hash functions (denoted by $2 \times (b = 8)$). We compare *PINT* to two state-of-the-art IP Traceback solutions PPM [65] and AMS2 [70] with $m = 5$ and $m = 6$. When configured with $m = 6$, AMS2 requires more packets to infer the path but also has a lower chance of false positives (multiple possible paths) compared with $m = 5$. We implement an improved version of both algorithms using Reservoir Sampling, as proposed in [63]. *PINT* is configured with $d = 10$ on the ISP topologies and $d = 5$ (as this is the diameter) on the fat tree topology. In both cases, this means a single XOR layer in addition to a Baseline layer.

The results (Fig. 10) show that *PINT* significantly outperforms previous works, even with a bit-budget of a single bit (PPM and AMS both have an overhead of 16 bits per packet). As shown, the required number of packets for *PINT* grows near-linearly with the path length, validating our theoretical analysis. For the Kentucky Datalink topology ($D = 59$), *PINT* with $2 \times (b = 8)$ on average uses 25–36 times fewer packets when compared to competing approaches. Even when using *PINT* with $b = 1$, *PINT* needs 7–10 times fewer packets than competing approaches. For the largest number of hops we evaluated (59, in the Kentucky Datalink topology), *PINT* requires only 42 packets on average and 94 for the 99'th percentile, while

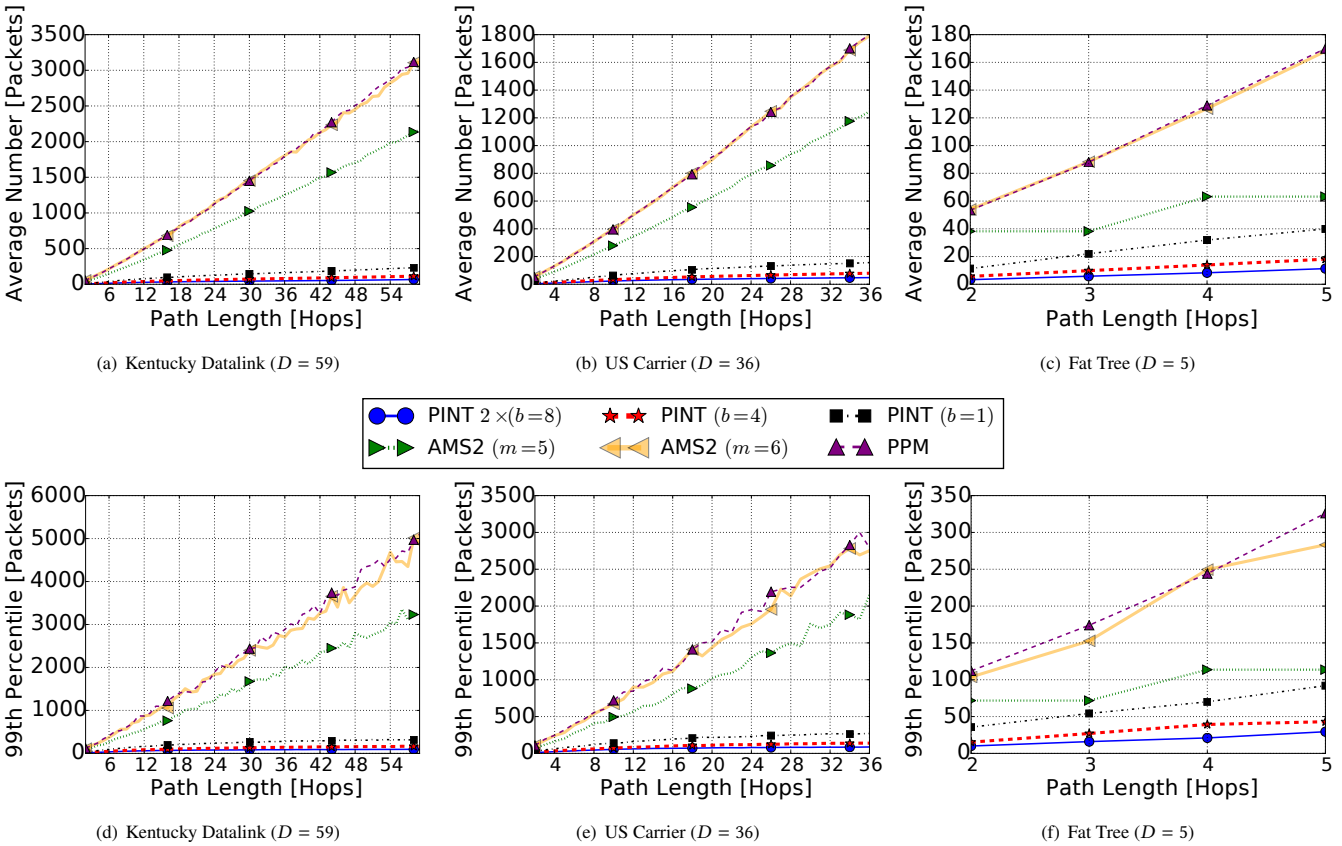


Figure 10: Comparison of the number of packets required (lower is better) for path decoding of different algorithms, including *PINT* with varying bit-budget.

alternative approaches need at least 1–1.5K on average and 3.3–5K for 99'th percentile, respectively.

6.4 Combined Experiment

We test the performance of *PINT* when running all three use cases concurrently. Based on the previous experiments, we tune *PINT* to run each query using a bit budget of 8 bits and a global budget of 16 bits. Our goal is to compare how *PINT* performs in such setting, compared with running each application alone using 16 bits per packet (i.e., with an effective budget of 3×16 bits). That is, each packet can carry digests of two of the three concurrent queries.

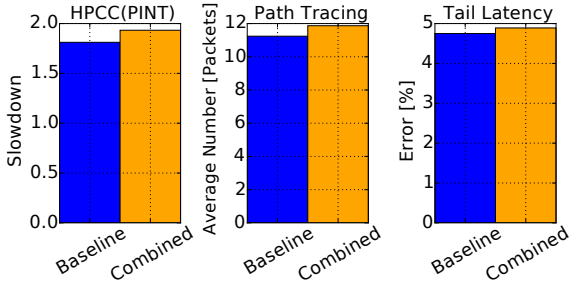


Figure 11: The performance of each query in a concurrent execution (FatTree topology + Hadoop workload) compared to running it alone.

As we observe that the congestion control application has good performance when running in $p = 1/16$ of the packets, and the path tracing requires more packets than the latency estimation, we choose the following configuration. We run the path algorithm on all packets, alongside the latency algorithm in $15/16$ of the packets, and alongside HPCC in $1/16$ of the packets. As Fig. 11 shows, the performance of *PINT* is close to a Baseline of running each query separately. For estimating median latency, the relative error increases by only 0.7% from the Baseline to the combined case. In case of HPCC, we that observe short flows become 6.6% slower while the performance of long flows does not degrade. As for path tracing, the number of packets increases by 0.5% compared with using two 8 bit hashes as in Fig. 10. We conclude that, with a tailored execution plan, our system can support these multiple concurrent telemetry queries using an overhead of just two bytes per packet.

7 LIMITATIONS

In this section, we discuss the limitations associated with our probabilistic approach. The main aspect to take into consideration is the required per-packet bit-budget and the network diameter. The bigger overhead allowed and the smaller the network, the more resilient *PINT* will be in providing results in different scenarios.

Tracing short flows. *PINT* leverages multiple packets from the same flow to infer its path. In our evaluation (§6), we show that our solution needs significantly fewer packets when compared to competing

approaches. However, in data center networks, small flows can consist of just a single packet [3]. In this case, *PINT* is not effective and a different solution, such as INT, would provide the required information.

Data plane complexity. Today's programmable switches have a limited number of pipeline stages. Although we show that it is possible to parallelize the processing of independent queries (§5), thus saving resources, the *PINT* requirements might restrict the amount of additional use cases to be implemented in the data plane, e.g., fast reroute [18] or in-network caching [37] and load balancing [2, 42].

Tracing flows with multipath routing. The routing of a flow may change over time (e.g., when using flowlet load balancing [2, 42]) or multiple paths can be taken simultaneously when appropriate transport protocols such as Multipath TCP are used [25]. In those cases, the values (i.e, switch IDs) for some hops will be different. Here, *PINT* can detect routing changes when observing a digest that is not consistent with the part of the path inferred so far. For example, if we know that the sixth switch on a path is M_6, and a Baseline packet p_j comes with a digest from this hop that is different than $h(M_6, p_j)$, then we can conclude that the path has changed. The number of packets needed to identify a path change depends on the fraction of the path that has been discovered. If path changes are infrequent, and *PINT* knows the entire path before the change, a Baseline packet will not be consistent with the known path (and thus signify a path change) with probability $1 - 2^{-q}$. Overall, in the presence of flowlet routing, *PINT* can still trace the path of each flowlet, provided enough packets for each flowlet-path are received at the sink. *PINT* can also profile all paths simultaneously at the cost of additional overhead (e.g., by adding a path checksum to packets we can associate each with the path it followed).

Current implementation. At the time of writing, the *PINT* execution plan is manually selected. We envision that an end to end system that implements *PINT* would include a Query Engine that automatically decides how to split the bit budget.

8 RELATED WORK

Many previous works aim at improving data plane visibility. Some focus on specific flows selected by operators [56, 78, 87] or only on randomly selected sampled flows [10, 21]. Such approaches are insufficient for applications that need global visibility on all flows, such as path tracing. Furthermore, the flows of interest may not be known in advance, if we wish to debug high-latency or malicious flows.

Other works can be classified into three main approaches: (1) keep information out-of-band; (2) keep flow state at switches; or (3) keep information on packets. The first approach applies when the data plane status is recovered by using packet mirroring at switches or by employing specially-crafted probe packets. Mirroring every packet creates scalability concerns for both trace collection and analysis. The traffic in a large-scale data center network with hundreds of thousands of servers can quickly introduce terabits of mirrored traffic [28, 62]. Assuming a CPU core can process tracing traffic at 10 Gbps, thousands of cores would be required for trace analysis [87], which is prohibitively expensive. Moreover, with mirroring it is not possible to retrieve information related to switch status, such as port utilization or queue occupancy, that are of paramount importance for applications such as congestion control or network troubleshooting. While such information can be retrieved with specially-crafted probes [74], the feedback loop may be too slow for applications like high precision congestion control [46]. We can also store flow information at switches and periodically export it to a collector [45, 69]. However, keeping state for a large number of active flows (e.g., up to 100K [62]), in the case of path tracing, is challenging for limited switch space (e.g., 100 MB [51]). This is because operators need the memory for essential control functions such as ACL rules, customized forwarding [68], and other network functions and applications [37, 51]. Another challenge is that we may need to export data plane status frequently (e.g., every 10 ms) to the collector, if we want to enable applications such as congestion control. This creates significant bandwidth and processing overheads [45].

Proposals that keep information on packets closely relate to this work [36, 72, 75], with INT being considered the state-of-the-art solution. Some of the approaches, e.g., Path Dump [72], show how to leverage properties of the topology to encode only part of each path (e.g., every other link). Nonetheless, this still imposes an overhead that is linear in the path length, while *PINT* keeps it constant. Alternative approaches add small digests to packets for tracing paths [64, 65, 70]. However, they attempt to trace back to potential attackers (e.g., they do not assume unique packet IDs or reliable TTL values as these can be forged) and require significantly more packets for identification, as we show in Section 6. In a recent effort to reduce overheads on packets, similarly to this work, Taffet et al. [71] propose having switches use Reservoir Sampling to collect information about a packet's path and congestion that the packet encounters as it passes through the network. *PINT* takes the process several steps further, including approximations and coding (XOR-based or network coding) to reduce the cost of adding information to packets as much as possible. Additionally, our work rigorously proves performance bounds on the number of packets required to recover the data plane status as well as proposes trade-offs between data size and time to recover.

9 CONCLUSION

We have presented *PINT*, a probabilistic framework to in-band telemetry that provides similar visibility to INT while bounding the per-packet overhead to a user-specified value. This is important because overheads imposed on packets translate to inferior flow completion time and application-level goodput. We have proven performance bounds (deferred to Appendix A due to lack of space) for *PINT* and have implemented it in P4 to ensure it can be readily deployed on commodity switches. *PINT* goes beyond optimizing INT by removing the header and using succinct switch IDs by restricting the bit-overhead to a constant that is independent of the path length. We have discussed the generality of *PINT* and demonstrated its performance on three specific use cases: path tracing, data plane telemetry for congestion control and estimation of experienced median/tail latency. Using real topologies and traffic characteristics, we have shown that *PINT* enables the use cases, while drastically decreasing the required overheads on packets with respect to INT.

Acknowledgements. We thank the anonymous reviewers, Jiaqi Gao, Muhammad Tirmazi, and our shepherd, Rachit Agarwal, for helpful comments and feedback. This work is partially sponsored by EPSRC project EP/P025374/1, by NSF grants #1829349, #1563710, and #1535795, and by the Zuckerman Foundation.

This work does not raise any ethical issues.

REFERENCES

[1] 2020. PINT open source code: https://github.com/ProbabilisticINT. (2020). https://github.com/ProbabilisticINT

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM*.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *ACM SIGCOMM*.

[4] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. 2012. Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *USENIX NSDI*.

[5] Arvind Arasu and Gurmeet Singh Manku. 2004. Approximate Counts and Quantiles over Sliding Windows. In *ACM PODS*.

[6] 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 *USENIX NSDI*.

[7] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast Userspace Packet Processing. In *IEEE/ACM ANCS*.

[8] Barefoot. [n. d.]. Barefoot Deep Insight. https://barefootnetworks.com/products/brief-deep-insight/. ([n. d.]).

[9] Barefoot Networks. 2018. Barefoot Deep Insight. https://www.barefootnetworks.com/static/app/pdf/DI-UG42-003ea-ProdBrief.pdf. (2018).

[10] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. 2020. Routing Oblivious Measurement Analytics. In *IFIP Networking*.

[11] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. 2018. Memento: Making Sliding Windows Efficient for Heavy Hitters. In *ACM CoNEXT*.

[12] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches using Probabilistic Recirculation. In *IEEE ICNP*.

[13] Ran Ben-Basat, Gil Einziger, and Roy Friedman. 2018. Fast flow volume estimation. *Pervasive Mob. Comput.* (2018).

[14] Broadcom. [n. d.]. Broadcom BCM56870 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series. ([n. d.]).

[15] Robert L Carter and Mark E Crovella. 1997. Server selection using dynamic path characterization in wide-area networks. In *IEEE INFOCOM*.

[16] SDX Central. [n. d.]. AT&T Runs Open Source White Box. https://www.sdxcentral.com/articles/news/att-runs-open-source-white-box-switch-live-network/2017/04/. ([n. d.]).

[17] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-Grained Queue Measurement in the Data Plane. In *ACM CoNEXT*.

[18] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiundefinedski, Georgios Nikolaidis, and Stefan Schmid. 2019. PURR: A Primitive for Reconfigurable Fast Reroute. In *ACM CoNEXT*.

[19] Baek-Young Choi, Sue Moon, Rene Cruz, Zhi-Li Zhang, and Christophe Diot. 2007. Quantile Sampling for Practical Delay Monitoring in Internet Backbone Networks. *Computer Networks*.

[20] Damu Ding, Marco Savi, and Domenico Siracusa. 2020. Estimating Logarithmic and Exponential Functions to Track Network Traffic Entropy in P4. In *IEEE/IFIP NOMS*.

[21] N. G. Duffield and Matthias Grossglauser. 2001. Trajectory Sampling for Direct Traffic Observation. In *IEEE/ACM ToN*.

[22] Nandita Dukkipati and Nick McKeown. 2006. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM CCR* (2006).

[23] David Felber and Rafail Ostrovsky. 2017. A Randomized Online Quantile Summary in $O((1/\varepsilon)\log(1/\varepsilon))$ Words. In *Theory of Computing*.

[24] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. 1992. Birthday Paradox, Coupon Collectors, Caching Algorithms and Self-organizing Search. *Discrete Applied Mathematics* (1992).

[25] Alan Ford, Costin Raiciu, Mark J. Handley, and Olivier Bonaventure. 2013. TCP Extensions for Multipath Operation with Multiple Addresses. (2013).

[26] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2019. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *USENIX NSDI*.

[27] Dimitrios Gkounis, Vasileios Kotronis, Christos Liaskos, and Xenofontas Dimitropoulos. 2016. On the Interplay of Link-Flooding Attacks and Traffic Engineering. *ACM SIGCOMM CCR* (2016).

[28] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM*.

[29] Dongsu Han, Robert Grandl, Aditya Akella, and Srinivasan Seshan. 2013. FCP: A Flexible Transport Framework for Accommodating Diversity. *ACM SIGCOMM CCR* (2013).

[30] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX NSDI*.

[31] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. 2010. ElasticTree: Saving Energy in Data Center Networks. In *USENIX NSDI*.

[32] T Ho, R Koetter, M Medard, DR Karger, and M Effros. 2003. The Benefits of Coding over Routing in a Randomized Setting. In *IEEE ISIT*.

[33] IEEE. [n. d.]. Standard 802.3. https://standards.ieee.org/standard/802_3-2015.html. ([n. d.]).

[34] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. QPipe: Quantiles Sketch Fully in the Data Plane. In *ACM CoNEXT*.

[35] Svante Janson. 2018. Tail Bounds for Sums of Geometric and Exponential Variables. *Statistics & Probability Letters* (2018).

[36] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *ACM SIGCOMM*.

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*.

[38] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical Real-Time Microburst Monitoring for Datacenter Networks. In *ACM APSys*.

[39] Zohar S. Karnin, Kevin J. Lang, and Edo Liberty. 2016. Optimal Quantile Approximation in Streams. In *IEEE FOCS*.

[40] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion Control for High Bandwidth-Delay Product Networks. In *ACM SIGCOMM*.

[41] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. 2017. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *ACM CoNEXT*.

[42] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM SOSR*.

[43] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. 2019. Confluo: Distributed Monitoring and Diagnosis Stack for High-Speed Networks. In *USENIX NSDI*.

[44] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The Internet Topology Zoo. *IEEE JSAC* (2011).

[45] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*.

[46] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*.

[47] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring Connectivity via Data Plane Mechanisms. In *USENIX NSDI*.

[48] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. 1998. Approximate Medians and Other Quantiles in One Pass and with Limited Memory. In *ACM SIGMOD*.

[49] Andrew Mcgregor, A. Pavan, Srikanta Tirthapura, and David P. Woodruff. 2016. Space-Efficient Estimation of Statistics Over Sub-Sampled Streams. *Algorithmica* (2016).

[50] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*.

[51] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM*.

[52] Mininet. [n. d.]. Mininet: An Instant Virtual Network on your Laptop. http://mininet.org/. ([n. d.]).

[53] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-Based Congestion Control for the Datacenter. In *ACM SIGCOMM*.

[54] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *ACM SIGCOMM*.

[55] Robert Morris. 1978. Counting Large Numbers of Events in Small Registers. In *Communications of ACM*. ACM.

[56] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In *USENIX NSDI*.

[57] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*.

[58] Netronome. [n. d.]. Netronome Agilio CX SmartNIC. https://www.netronome.com/blog/in-band-network-telemetry-its-not-rocket-science/. ([n. d.]).

[59] Donald J Newman. 1960. The Double Dixie Cup Problem. *The American Mathematical Monthly* (1960).

[60] Remi Oudin, Gianni Antichi, Charalampos Rotsos, Andrew W. Moore, and Steve Uhlig. 2019. OFLOPS-SUME and the art of switch characterization. *IEEE JSAC* (2019).
[61] Diana Popescu, Noa Zilberman, and Andrew W. Moore. 2017. Characterizing the Impact of Network Latency on Cloud-based Applications' Performance. In *Technical Report, Number 914, UCAM-CL-TR-914*. University of Cambridge.
[62] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*.
[63] Pegah Sattari. 2007. Revisiting IP Traceback as a Coupon Collector's Problem. In *PhD Dissertation*. University of California, Irvine.
[64] Pegah Sattari, Minas Gjoka, and Athina Markopoulou. 2010. A network coding approach to IP traceback. In *IEEE Symposium on Network Coding (NetCod)*.
[65] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. 2000. Practical Network Support for IP Traceback. In *ACM SIGCOMM*.
[66] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. 2004. Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams. In *ACM IMC*.
[67] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *USENIX NSDI*.
[68] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. 2015. DC.P4: Programming the Forwarding Plane of a Datacenter Switch. In *ACM SOSR*.
[69] Alex C. Snoeren, Craig Partridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Stephen T. Kent, and W. Timothy Strayer. 2001. Hash-based IP Traceback. In *ACM SIGCOMM*.
[70] Dawn Xiaodong Song and Adrian Perrig. 2001. Advanced and Authenticated Marking Schemes for IP Traceback. In *IEEE INFOCOM*.
[71] Philip Taffet and John Mellor-Crummey. 2019. Understanding Congestion in High Performance Interconnection Networks Using Sampling. In *ACM SC*.
[72] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with Pathdump. In *USENIX OSDI*.
[73] Praveen Tammana, Rachit Agarwal, and Mjungjin Lee. 2018. Distributed Network Monitoring and Debugging with SwitchPointer. In *USENIX NSDI*.
[74] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. 2019. Netbouncer: Active Device and Link Failure Localization in Data Center Networks. In *USENIX NSDI*.
[75] The P4.org Applications Working Group. [n. d.]. In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report.pdf. ([n. d.]).
[76] The University of Washington NS-3 Consortium. [n. d.]. NS3 official website. https://www.nsnam.org/. ([n. d.]).
[77] Wolfgang Theilmann and Kurt Rothermel. 2000. Dynamic distance maps of the Internet. In *IEEE INFOCOM*.
[78] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. 2018. Stroboscope: Declarative Network Monitoring on a Budget. In *USENIX NSDI*.
[79] Muhammad Tirmazi Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. 2020. Cheetah: Accelerating Database Queries with Switch Pruning. In *ACM SIGMOD*.
[80] P Van Mieghem, Gerard Hooghiemstra, and Remco Hofstad. 2001. A scaling law for the hopcount in Internet. In *PAM*.
[81] Vladimir N Vapnik and A Ya Chervonenkis. 2015. On the Uniform Convergence of Relative Frequencies of Events to their Probabilities. *Measures of Complexity* (2015).
[82] Jeffrey S Vitter. 1985. Random Sampling with a Reservoir. *Transactions on Mathematical Software* (1985).
[83] Xilinx. [n. d.]. 10G/25G Ethernet Subsystem. https://www.xilinx.com/products/intellectual-property/ef-di-25gemac.html. ([n. d.]).
[84] Xilinx. [n. d.]. UltraScale Integrated 100G Ethernet Subsystem. https://www.xilinx.com/products/intellectual-property/cmac.html. ([n. d.]).
[85] Xilinx. [n. d.]. Xilinx to Showcase Unprecedented Programmability and Visibility. https://www.xilinx.com/news/press/2018/barefoot-networks-and-xilinx.html. ([n. d.]).
[86] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *USENIX NSDI*.
[87] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM*.

A ANALYSIS

A.1 Dynamic per-flow Aggregation

We now survey known results that Theorem 1 and Theorem 2 are based on.

Quantiles. Classical streaming results show that by analyzing a uniformly selected subset of $O(\varepsilon_s^{-2}\log\varepsilon_s^{-1})$ elements, one can estimate all possible quantiles [48, 81] to within an additive error of ε_s. It is also known that if one is interested in a specific quantile (e.g., median), a subset size of $O(\varepsilon_s^{-2})$ is enough.

In our case, for each switch s_i through which flow x is routed, we get a sampled substream of $S_{i,x}$ where each packet carries a value from it with probability $1/k$. This is not a fixed-size subset, but a *Bernouli sample*. Nevertheless, Felber and Ostrovsky show that a Bernouli sample with the same *expected size* is enough [23]. Therefore, for a specific quantile (e.g., median) we need to get $O(\varepsilon_s^{-2})$ samples for each of the k switches on the path. Using a standard Chernoff bound argument, we have that if $z = O(k\varepsilon_s^{-2})$ packets reach the *PINT* sink, *all* hops on the path will get at least $O(\varepsilon_s^{-2})$ samples with probability $1 - e^{-\Omega(z/k)} = 1 - e^{-\Omega(\varepsilon_s^{-2})}$.

To compress the amount of per-flow storage needed for computing the quantiles, we can use a $\widetilde{O}(\varepsilon_a^{-1})$ space sketch such as KLL [39]. We run separate sketch for each of the k hops, thus needing $\widetilde{O}(k\varepsilon_a^{-1})$ per-flow storage in total. The resulting error would be $\varepsilon = \varepsilon_s + \varepsilon_a$, as the sampling adds an additive error of ε_s and the sketching an additive error of ε_a.

Frequent Values. Using a standard Chernoff bound argument, one can use a $O(\varepsilon_s^{-2})$-sized substream of $S_{i,x}$, one can estimate the fraction in which each specific value appears, up to an additive error of ε_s. We can then use a heavy hitters algorithm like Space Saving [50] to estimate the frequency of values in the sampled substream to within an additive error of ε_a, using $O(\varepsilon_a^{-1})$ space. As before, to get the correct estimations for all hops, we need a factor k multiplicative overhead to both the number of packets and space.

A.2 Static per-flow Aggregation

Before we can analyze the algorithm (§A.2.2), we start with some auxiliary results.

A.2.1 Auxiliary Results.

The first lemma gives a bound on how many independent coins with probability p we need to flip until we get k successes.

LEMMA 4. *Let $k \in \mathbb{N}$ and $p, \delta \in (0,1)$.*
Denote $N = \frac{k+2\ln\delta^{-1}+\sqrt{2k\ln\delta^{-1}}}{p}$ and let $X \sim Bin(N,p)$. Then

$$\Pr[X \le k] \le \delta.$$

PROOF. Using the Chernoff bound we have that for any $\gamma > 0$:

$$\Pr[X < \mathbb{E}[X](1-\gamma)] \le e^{-\gamma^2\mathbb{E}[X]/2}.$$

We set $\gamma = \sqrt{\frac{2\ln\delta^{-1}}{Np}}$, which means that $\gamma^2\mathbb{E}[X]/2 = \ln\delta$ and therefore $\Pr[X < \mathbb{E}[X](1-\gamma)] \le \delta$.

Finally,

$$\mathbb{E}[X](1-\gamma) = Np(1-\gamma) = Np - \sqrt{2Np\ln\delta^{-1}}.$$

Denote $x = \sqrt{Np}$, then we want to show that

$$x^2 - x\sqrt{2\ln\delta^{-1}} - k \ge 0,$$

which holds for

$$x > \frac{\sqrt{2\ln\delta^{-1}} + \sqrt{2\ln\delta^{-1}+4k}}{2} = \frac{\sqrt{\ln\delta^{-1}} + \sqrt{\ln\delta^{-1}+2k}}{\sqrt{2}}.$$

This gives

$$N = x^2/p \ge \frac{\left(\sqrt{\ln\delta^{-1}} + \sqrt{\ln\delta^{-1}+2k}\right)^2}{2p} = \frac{k + 2\ln\delta^{-1} + \sqrt{2k\ln\delta^{-1}}}{p}.$$

□

The next theorem provide a high-probability bound on the Double Dixie Cup problem [59]. Specifically, consider trying to collect at least Z copies from each of k coupons, where at each stage you get a random coupon. The following bounds the number of samples you need.

THEOREM 5. *After seeing*

$$N = k\cdot\left(Z - 1 + \ln(k/\delta) + \sqrt{(Z-1+\ln(k/\delta))^2 - (Z-1)^2/4}\right)$$

samples, our algorithm has at least Z copies of each of the k coupons.

PROOF. Therefore, the number of copies of the i'th coupon isa binomial random variable we denote by $Y_i \sim Bin(N, 1/k)$. Our goal is to show that getting $Y_i < Z$ is unlikely; to that end, we use the Chernoff inequality that states that $\Pr[Y_i \le \mathbb{E}[Y_i](1-\gamma)] \le e^{-\mathbb{E}[Y_i]\gamma^2/2}$ for any $\gamma \in (0,1]$. We set $\gamma = 1 - k(Z-1)/N$ to get

$$\begin{aligned}\Pr[Y_i < Z] &= \Pr[Y_i \le Z-1] = \Pr[Y_i \le \mathbb{E}[Y_i](1-\gamma)] \\ &\le e^{-\mathbb{E}[Y_i]\gamma^2/2} = e^{-N/2k\cdot(1-k(Z-1)/N)^2} \\ &= e^{-(N/2k-(Z-1)+k(Z-1)^2/2N)} = e^{-(x-(Z-1)+(Z-1)^2/4x)},\end{aligned}$$

where $x = N/(2k)$. We want $\Pr[X_i < Z] \le \delta/k$, which according to the above follows from

$$x - (Z-1) + (Z-1)^2/4x \ge \ln k/\delta \iff$$
$$x \ge 0.5\cdot\left(Z - 1 + \ln(k/\delta) + \sqrt{(Z-1+\ln(k/\delta))^2 - (Z-1)^2/4}\right).$$

The last inequality follows directly from our choice of N. Finally, we use the union bound to get that after N samples all coupons get at least Z copies except with probability $k\cdot\Pr[Y_i < Z] \le \delta$. □

We proceed with a tail bound on the Partial Coupon Collector problem, in which we wish to get N out of r possible coupons, where at each timestamp we get a random coupon. Our proofs relies on the following result for a sharp bound on the sum of geometric random variables:

THEOREM 6. *([35]) Let $\{A_1, \ldots A_N\}$ be independent geometric random variables such that $A_i \sim Geo(p_i)$ and $p_1 \ge \ldots \ge p_N$. Then the sum $A = \sum_{i=1}^{N} A_i$ satisfies:*

$$\Pr\left[A > \mathbb{E}[A]\cdot\lambda\right] \le e^{-p_N\mathbb{E}[A](\lambda-1-\ln\lambda)}.$$

Additionally, we will use the following fact.

FACT 7. *For any positive real number* $\varepsilon \in \mathbb{R}^+$,

$$1 + \varepsilon + \sqrt{2\varepsilon} - \ln(1 + \varepsilon + \sqrt{2\varepsilon}) \geq 1 + \varepsilon.$$

We now prove our result.

THEOREM 8. *Let* $\mathbb{E}[A] = r(H_r - H_{r-N})$ *denote the* expected *number of samples required for seeing N distinct coupons. With probability* $1 - \delta$*, the number of samples required for seeing at least N distinct coupons is at most*

$$\mathbb{E}[A] + \frac{r \ln \delta^{-1}}{(r-N)} + \sqrt{\frac{2r\mathbb{E}[A] \ln \delta^{-1}}{(r-N)}}.$$

PROOF. We wish to use Theorem 6; notice that we need $\lambda - \ln \lambda \geq 1 + \frac{\ln \delta^{-1}}{p_N \mathbb{E}[A]}$ which implies

$$e^{-p_N \mathbb{E}[A](\lambda - 1 - \ln \lambda)} \leq \delta.$$

According to Fact 7, for $\varepsilon = \frac{\ln \delta^{-1}}{p_N \mathbb{E}[A]}$, it is enough to set

$$\lambda = 1 + \frac{\ln \delta^{-1}}{p_N \mathbb{E}[A]} + \sqrt{\frac{2 \ln \delta^{-1}}{p_N \mathbb{E}[A]}}.$$

Plugging in $p_N = (1 - (N-1)/r) > (r-N)/r$ we have that the required number of required packets is at most

$$\lambda \cdot \mathbb{E}[A] = \left(\mathbb{E}[A] + \frac{\ln \delta^{-1}}{p_N} + \sqrt{\frac{2\mathbb{E}[A] \ln \delta^{-1}}{p_N}}\right) \leq \left(\mathbb{E}[A] + \frac{r \ln \delta^{-1}}{(r-N)} + \sqrt{\frac{2r\mathbb{E}[A] \ln \delta^{-1}}{(r-N)}}\right).$$

For example, if $r = 2N$, we have $\mathbb{E}[A] \approx 1.39N$ and the number of packets is

$$\left(\mathbb{E}[A] + 2\ln \delta^{-1} + \sqrt{4\mathbb{E}[A] \ln \delta^{-1}}\right) \approx \left(1.39N + 2\ln \delta^{-1} + 2.35\sqrt{N \ln \delta^{-1}}\right). \quad \square$$

Next, we show a bound on the number of samples needed to collect $\mathcal{K}(1-\psi)$ in a Coupon Collector process [24] on $\mathcal{K}$ coupons.

LEMMA 9. *Let* $\mathcal{K} \in \mathbb{N}^+$ *and* $\psi \in (0, 1/2]$*. The number of samples required for collecting all but* $\psi\mathcal{K}$ *coupons is at most*

$$\mathcal{K} \ln \psi^{-1} + \psi^{-1} \ln \delta^{-1} + \sqrt{2\mathcal{K}\psi^{-1} \ln \psi^{-1} \ln \delta^{-1}} = O(\mathcal{K} \ln \psi^{-1} + \psi^{-1} \ln \delta^{-1}).$$

PROOF. For $i = 1, \ldots, \mathcal{K}(1-\psi)$, let $A_i \sim Geo(1 - (i-1)/\mathcal{K})$ denote the number of samples we need for getting the i'th distinct coupon, and let $A = \sum_{i=1}^{K(1-\psi)} A_i$. We have that

$$\mathbb{E}[A] = \sum_{i=1}^{\mathcal{K}(1-\psi)} \frac{\mathcal{K}}{\mathcal{K} - (i-1)} = \mathcal{K}\left(H_{\mathcal{K}} - H_{\mathcal{K}\psi}\right) = \mathcal{K} \ln \psi^{-1}.$$

According to Theorem 8, it is enough to obtain the following number of samples

$$\mathbb{E}[A] + \frac{\mathcal{K} \ln \delta^{-1}}{\mathcal{K}(1-(1-\psi))} + \sqrt{\frac{2\mathcal{K}\mathbb{E}[A] \ln \delta^{-1}}{\mathcal{K}(1-(1-\psi))}} = \mathcal{K} \ln \psi^{-1} + \psi^{-1} \ln \delta^{-1} + \sqrt{2\mathcal{K}\psi^{-1} \ln \psi^{-1} \ln \delta^{-1}}.$$

Finally, we note that $\sqrt{\mathcal{K}\psi^{-1} \ln \psi^{-1} \ln \delta^{-1}}$ is the geometric mean of $\mathcal{K} \ln \psi^{-1}$ and $\psi^{-1} \ln \delta^{-1}$ and thus:

$$\mathcal{K} \ln \psi^{-1} + \psi^{-1} \ln \delta^{-1} + \sqrt{2\mathcal{K}\psi^{-1} \ln \psi^{-1} \ln \delta^{-1}} \leq \left(\mathcal{K} \ln \psi^{-1} + \psi^{-1} \ln \delta^{-1}\right)(1 + 1/\sqrt{2}) = O\left(\mathcal{K} \ln \psi^{-1} + \psi^{-1} \ln \delta^{-1}\right).$$

$\square$

A.2.2 Analysis of the algorithm. We denote by $\mathfrak{d} \triangleq \frac{d}{\log^* d}$ the number of hops we aim to decode using the XOR layers. Our algorithm has $\lceil \log^* \mathfrak{d} \rceil + 1$ *layers*, where layer 0 runs the Baseline scheme and the remaining $\mathcal{L} \triangleq \lceil \log^* \mathfrak{d} \rceil$ layers use XOR. We denote by $\uparrow\uparrow$ Knuth's iterated exponentiation arrow notation, i.e., $x \uparrow\uparrow 0 = 1$ and

$$x \uparrow\uparrow y = x^{x^{\cdot^{\cdot^{x}}}} \Big\}\, y\text{-times}.$$

The sampling probability in layer ℓ is then set to

$$p_\ell = \frac{e \uparrow\uparrow (\ell - 1)}{\mathfrak{d}}.$$

Each packet it hashed to choose a layer, such that layer 0 is chosen with probability $\tau = \left(1 - \frac{1}{1 + \log \log^* \mathfrak{d}}\right) = 1 - o(1)$ and otherwise one of layers $1, \ldots, \mathcal{L}$ is chosen uniformly. The pseudo code for the final solution is given in Algorithm 1.

Algorithm 1 *PINT* Processing Procedure at Switch s

Input: A packet p_j with $\mathfrak{b}$-bits digest p_j.dig.
Output: Updated digest p_j.dig.
Initialization:
$\tau = \frac{\log \log^* \mathfrak{d}}{1 + \log \log^* \mathfrak{d}}$, $\forall \ell \in \{1, \ldots, \mathcal{L}\} : p_\ell = \frac{e \uparrow\uparrow (\ell - 1)}{\mathfrak{d}}$.

Let i such that the current switch is the $i'th$ so far
$\mathfrak{H} \leftarrow \mathcal{H}(p_j)$ ▷ Distributed uniformly on $[0, 1]$
if $\mathfrak{H} < \tau$ **then** ▷ Update layer 0
  if $g(p_j, i) < 1/i$ **then**
    p_j.dig $\leftarrow h(s, p_j)$ ▷ Sample with probability $1/i$
else
  $\ell \leftarrow \left\lceil \mathcal{L} \cdot \frac{\mathfrak{H} - \tau}{1 - \tau} \right\rceil$ ▷ Choose the layer
  if $g(p_j, i) < p_\ell$ **then**
    p_j.dig $\leftarrow p_j$.dig $\oplus\, h(s, p_j)$ ▷ Xor w.p. p_ℓ

For simplicity, we hereafter assume in the analysis that a packet can encode an entire identifier. This assumption is not required in practice and only serves for the purpose of the analysis. We note that even under this assumption the existing approaches require $O(k \log k)$ packets. In contrast, we show that except with probability

$\delta = e^{-O(k^{0.99})}$ the number of packets required for decoding a k-hops path in our algorithm is just

$$\mathcal{X} = k \log\log^* k \cdot (1 + o(1)).^{11}$$

Note that $\log\log^* k$ is a function that grows extremely slowly, e.g., $\log\log^* P < 2$ where P is the number of atoms in the universe. Our assumption on the error probability δ allows us to simplify the expressions and analysis but we can also show an

$$O\left(k \log\log^* k + \log^* k \log \delta^{-1}\right)$$

bound on the required number of packets thus the dependency on δ is minor.

For our proof, we define the quantities

$$\mathcal{Q} \triangleq k_1 + \ln\left(\frac{4\log^* k_1}{\delta}\right) + \sqrt{2k_1 \ln\left(\frac{4\log^* k_1}{\delta}\right)} = O\left(\frac{k}{\log^* k} + \log\delta^{-1}\right) = O\left(\frac{k}{\log^* k}\right)$$

and

$$\mathcal{S} \triangleq \frac{\mathcal{Q} + 2\ln\left(\frac{4\mathcal{L}}{\delta}\right) + \sqrt{2\mathcal{Q}\ln\left(\frac{4\mathcal{L}}{\delta}\right)}}{c \cdot e^{-c}} = O\left(\frac{k}{\log^* k} + \log\delta^{-1}\right) = O\left(\frac{k}{\log^* k}\right).$$

Note that $\mathcal{Q}$ and $\mathcal{S}$ are not known to our algorithm (which is only aware of d) and they are used strictly for the analysis. Our proof follows the next roadmap:

(1) When a flow has at least $\mathcal{X} \triangleq k\log\log^* k \cdot (1 + o(1))$ packets, Baseline (layer 0) gets at least $\mathcal{X} \cdot (1 - o(1)) = k\log\log^* k \cdot (1 + o(1))$ digests and XOR (layers 1 and above) gets at least $\Omega(\mathcal{X}/\log\log^* k) = \Omega(k)$ digests with probability $1 - \delta/6$.
(2) When Baseline (layer 0) gets at least $\mathcal{X} \cdot (1 - o(1))$ digests, it decodes all hops but $k_1 \triangleq \frac{k}{\log^* k}$ with probability $1 - \delta/6$.
(3) When at least $\Omega(k)$ packets reach XOR (layers 1 and above), with probability $1 - \delta/6$ each layer gets at least $\mathcal{S}$ digests.
(4) When a layer $\ell \in \{1, \ldots, \mathcal{L}\}$ gets $\mathcal{S}$ digests, with probability $1 - \delta/6\mathcal{L}$, at least $\mathcal{Q}$ of the digests contain exactly one of the k_ℓ undecoded switches.
(5) When a layer $\ell \in \{1, \ldots, \mathcal{L} - 1\}$ gets $\mathcal{Q}$ of digests that contain exactly one of the $k_\ell \triangleq k_1/(e \uparrow\uparrow (\ell - 1))$ undecoded switches, it decodes all hops but at most $k_{\ell+1}$ with probability $1 - \delta/6\mathcal{L}$.
(6) When the last layer $\mathcal{L}$ gets $\mathcal{Q}$ of digests that contain exactly one of the k_ℓ undecoded switches, it decoded all the remaining hops with probability $1 - \delta/6\mathcal{L}$.

We then use the union bound over all bad events to conclude that the algorithm succeeds with probability at least $1 - \delta$.

[11] The $o(1)$ part hides an additive $O(k)$ term, which we upper bound as $\frac{k}{c \cdot e^{-c}}$ up to lower order terms. Specifically, if $d = k$ (thus, $c = 1$), the required number of packets is at most $k\log\log^* k + e \cdot k + o(k)$.

A.2.3 Proof of Part (1). The first step is to observe that by a straightforward application of the Chernoff bound, since layer 0 is chosen with probability $1/2$, the number of packets that it receives is with probability $1 - \delta/6$:

$$\mathcal{X}_0 = \tau \cdot \mathcal{X} \pm O\left(\sqrt{\tau \cdot \mathcal{X} \cdot \log\delta^{-1}}\right).$$

Since $\mathcal{X} = \omega(\log\delta^{-1})$, we have that

$$\mathcal{X}_0 \geq \mathcal{X}(\tau - o(1)) = k\log\log^* k \cdot (1 + o(1)).$$

A.2.4 Proof of Part (2). Applying Lemma 9 for $\psi = \frac{1}{\log^* k}$, we get that after

$$k\ln\log^* k + \log^* k \ln\delta^{-1} + \sqrt{2k\log^* k \ln\log^* k \ln\delta^{-1}} = k\log\log^* k \cdot (1 + o(1))$$

packets from layer 0 the number of hops that are not decoded is at most $k_1 \triangleq k \cdot \psi = \frac{k}{\log^* k}$ with probability $1 - \delta/6$. That is, we use k_1 to denote the number of undecoded hops that are left for layers 1 and above.

A.2.5 Proof of Part (3). When at least $\Omega(k)$ reach XOR, the number of digests that the levels get is a balls and bins processes with the levels being the bins. According to Theorem 5:

After seeing

$$\mathcal{L} \cdot \left(\mathcal{S} - 1 + \ln(6\mathcal{L}/\delta) + \sqrt{(\mathcal{S} - 1 + \ln(6\mathcal{L}/\delta))^2 - (\mathcal{S} - 1)^2/4}\right) = O\left(\mathcal{L} \cdot (\mathcal{S} + \log(\mathcal{L}/\delta))\right) = O\left(\log^* k \cdot \left(\frac{k}{\log^* k} + \log\delta^{-1} + \log(\delta^{-1}\log^* k)\right)\right) = O(k)$$

packets, with probability $1 - \delta/6$ our algorithm has at least $\mathcal{Q}$ samples in each layer.

A.2.6 Proof of Part (4). Follows from Lemma 4 for $p = c \cdot e^{-c}$, $k = \mathcal{Q}$ and $\delta' = \frac{\delta}{6\mathcal{L}}$.

A.2.7 Proof of Part (5). Follows from Lemma 9 with $\mathcal{K} = k_\ell$ and $\psi = \frac{k_{\ell+1}}{k_\ell}$.

A.2.8 Proof of Part (6). The last layer is samples needs to decode

$$k_{\mathcal{L}} \leq \frac{k_1}{e \uparrow\uparrow (\mathcal{L} - 1)} = \frac{k_1}{\log\mathfrak{d}} = O\left(\frac{k_1}{\log k_1}\right)$$

and samples with probability

$$p_{\mathcal{L}} = \frac{e \uparrow\uparrow (\mathcal{L} - 1)}{\mathfrak{d}} = \frac{\log\mathfrak{d}}{\mathfrak{d}} = \Theta\left(\frac{\log k_1}{k_1}\right).$$

Therefore, with a constant probability, a digest would be xor-ed by exactly one of the $k_{\mathcal{L}}$ undecoded hops, and the number of such packets needed to decode the remainder of the path is $O\left(k_{\mathcal{L}}\log k_{\mathcal{L}}\right) = O(k_1)$.

A.3 Revised Algorithm to Improve the Lower Order Term's Constant

Consider changing the algorithm to sample layer 0 with probability

$$\tau' \triangleq \frac{1 + \log\log^* d}{2 + \log\log^* d} = 1 - \frac{1}{2 + \log\log^* d}.$$

Then when getting $\mathcal{X}' = k \cdot \left(\log\log^* k + 1 + \frac{1}{ce^{1-c}} + o(1)\right)$, we will have

$$k \cdot (\log\log^* k + 1 + o(1))$$

packets that reach layer 0, which would leave only

$$k_1' \triangleq \frac{k}{e \cdot \log^* k}$$

undecoded hops to layers 1 and above. As above, the number of packets required for the upper layers to decode the missing hops is

$$\frac{k_1' \log^* k_1'}{ce^{-c}} \leq \frac{k}{ce^{1-c}}.$$

Since $ce^{-c} \leq 1/e$ for any $c > 0$, we get that this is a strict improvement in the number of packets that are required for the path decoding. For example, if $d = k$ (i.e., $c = 1$), we reduce the required number of packets from $k(\log\log^* k + e + o(1))$ to $k(\log\log^* k + 2 + o(1))$.

A.4 An Extension – Detecting Routing Loops

Real-time detection of routing loops is challenging, as switches need to recognize looping packets without storing them. Interestingly, we can leverage *PINT* to detect loops on the fly. To do so, we check whether the current switch's hash matches the one on the packet. Specifically, before choosing whether to sample or not, the switch checks whether $p_j.\text{dig} = h(s, p_j)$. If there is a loop and s was the last switch to write the digest, it will be detected. Unfortunately, such an approach may result in a significant number of false positives. For example, if we use $\mathtt{b} = 16$-bit hashes, the chance of reporting a false loop over a path of length 32 would be roughly 0.05%, which means several false positives per second on a reasonable network.

To mitigate false positives, we propose requiring multiple matches, corresponding to multiple passes through the loop. We use an additional counter c to track the number of times a switch hash matched the digest. When $c = 0$, the switches follow the same sampling protocol as before. However, if $c > 0$ then the digest is no longer changed, and if c exceeds a value of T then we report a loop. This changes the loop detection time, but the switch that first incremented c may report the loop after at most T cycles over the loop. This approach adds $\lceil \log_2(T+1) \rceil$ bits of overhead, but drastically reduces the probability of false positives. For example, if $T = 1$ and $\mathtt{b} = 15$, we still have an overhead of sixteen bits per packet, but the chance of reporting false loops decreases to less than $5 \cdot 10^{-7}$. If we use $T = 3$ and $\mathtt{b} = 14$, the false reporting rate further decreases to $5 \cdot 10^{-13}$, which allows the system to operate without false alarms in practice.

Algorithm 2 *PINT* Processing at s with Loop Detection

Input: A packet p_j with $\mathtt{b}$-bits digest $p_j.\text{dig}$ and a counter $p_j.c$.
Output: Updated digest $p_j.\text{dig}$ or LOOP message.

1: **if** $p_j.\text{dig} = h(s, p_j)$ **then**
2: **if** $p_j.c = T$ **then return** LOOP
3: $p_j.c \leftarrow p_j.c + 1$

Let i such that the current switch is the $i'th$ so far

4: **if** $p_j.c = 0$ and $g(p_j, i) < 1/i$ **then**
5: $p_j.\text{dig} \leftarrow h(s, p_j)$ ▷ Sample with probability $1/i$

B COMPTING HPCC'S UTILIZATION

We first calculate the logarithm:

$$U\_term = \log(\frac{T-\tau}{T} \cdot U) = \log(T-\tau) - \log T + \log U$$

$$qlen\_term = \log(\frac{qlen \cdot \tau}{B \cdot T^2}) = \log qlen + \log \tau - \log B - 2\log T$$

$$byte\_term = \log(\frac{byte}{B \cdot T}) = \log byte - \log B - \log T$$

Then calculate U using exponentiation:

$$U = 2^{U\_term} + 2^{qlen\_term} + 2^{byte\_term}$$

C ARITHMETIC OPERATIONS IN THE DATA PLANE

Some of our algorithms require operations like multiplication and division that may not be natively supported on the data plane of current programmable switches. Nonetheless, we now discuss how to approximate these operations through fixed-point representations, logarithms, and exponentiation. We note that similar techniques have appeared, for example, in [67], [79] and [20].

Fixed-point representation: Modern switches may not directly support representation of fractional values. Instead, when requiring a *real-valued* variable in the range $[0, R]$, we can use m bits to represent it so that the integer representation $r \in \{0, 1, \ldots, 2^m - 1\}$ stands for $R \cdot r \cdot 2^{-m}$. R is called *scaling factor* and is often a power of two for simplicity. For example, if our range is $[0, 2]$, and we use $m = 16$ bits, then the encoding value 39131 represents $2 \cdot 39131 \cdot 2^{-16} \approx 1.19$.

Conveniently, this representation immediately allows using integer operations (e.g., addition or multiplication) to manipulate the variables. For example, if x and y are variables with scale factor R that are represented using $r(x), r(y)$, then their sum is represented using $r(x) + r(y)$ (assuming no overflow, this keeps the scaling factor intact) and their product is $r(x) \cdot r(y)$ with a scaling factor of R^2. As a result, we hereafter consider operating on integer values.

Computing logarithms and exponentiating: Consider needing to approximate $\log_2(x)$ for some integer x (and storing the result using a fixed-point representation). If the domain of x is small (e.g., it is an 8-bit value), we can immediately get the value using a lookup table. Conversely, say that x is an m-bit value for a large m (e.g., $m = 64$). In this case, we can use the switch's TCAM to find the most significant set bit in x, denoted ℓ. That is, we have that $x = 2^\ell \cdot \alpha$ for some $\alpha \in [1, 2)$. Next, consider the next q bits of x, denoted by x_q, where q is such that it is feasible to store a 2^q-sized lookup table on the switch (e.g., $q = 8$). [12] Then we have that $x = x_q \cdot 2^{\ell-q}(1+\varepsilon)$ for a small relative error $\varepsilon < 2^{-q}$. Therefore, we write

$$\log_2(x) = \log_2(x_q \cdot 2^{\ell-q}(1+\varepsilon)) = (\ell - q) + \log_2(x_q) + \log_2(1+\varepsilon).$$

Applying the lookup table to x_q, we can compute $\widetilde{y} \triangleq (\ell - q) + \log_2(x_q)$ on the data plane and get that $\widetilde{y} \in [\log_2 x - \log_2(1+\varepsilon), \log_2 x]$. [13] We can further simplify the error expression as $\log_2(1+$

[12] If $q < \ell$ we can simply look up the exact value as before.
[13] In addition to the potential error that arises from the lookup table.

$\varepsilon) \leq \varepsilon/\ln 2 \approx 1.44 \cdot 2^{-q}$. We also note that computing logarithms with other bases can be done similarly as $\log_y x = \log_2 x/\log_2 y$.

For exponentiation, we can use a similar trick. Assume that we wish to compute 2^x for some real-valued x that has a fixed-point representation r. Consider using a lookup table of 2^q entries for a suitable value of q, and using the TCAM to find the most significant set bit in r. Then we can compute 2^x up to a multiplicative factor of $2^{x\varepsilon}$ for some $\varepsilon \leq 2^{-q}$. Assuming that x is bounded by $R \leq 2^q$, this further simplifies to $2^{x\varepsilon} \leq 2^{x2^{-q}} \leq 1+R\cdot 2^{-q}$. For example, if x is in the range $[0, 2]$ and we are using $q = 8$ then logarithms are computed to within a $(1 + 2^{-7})$-multiplicative factor (less than 1% error).

Multiplying and dividing: We overcome the lack of support for arithmetic operations such as multiplication and division using approximations, via logarithms and exponentiation. Intuitively, we have that $x \cdot y = 2^{\log_2 x+\log_2 y}$ and $x/y = 2^{\log_2 x-\log_2 y}$. We have already discussed how to approximate logarithms and exponentiation, while addition and subtraction are currently supported. We note that the errors of the different approximations compound and thus it is crucial to maintain sufficient accuracy at each step to produce a meaningful approximation for the multiplication and division operations.

An alternative approach is to directly use a lookup table that takes the $q/2$ most significant bits, starting with the first set bit, of x and y and return their product/quotient (as before, this would require a 2^q-sized table). However, going through logarithms may give a more accurate result as the same lookup table can be used for both x and y, and its keys are a single value, which allows considering q bits for the same memory usage.