CubeTrace: Microscopic Network Tracing for Heterogeneous Cloud Gateways

Yunming Xiao[1] Yinchao Yang[2] Jiaqi Zheng[3] Xuqian Li[2] Dongbo Gu[2] Jun Zhang[2]
Miantao Wan[2] Chao Pei[2] Chen Tian[3] Mingwei Xu[4] Ang Chen[5] Congcong Miao[6*]

[1] The Chinese University of Hong Kong, Shenzhen [2] Tencent [3] Nanjing University
[4] Tsinghua University [5] University of Michigan [6] National University of Singapore

ABSTRACT

Modern cloud gateways have evolved to include diverse network functions and heterogeneous hardware, such as programmable switches and FPGAs, to handle increasing workloads and minimize forwarding latency. Existing network tracing tools, however, operate primarily at device granularity and cannot pinpoint which function on which hardware component causes packet losses or latency spikes. To bridge this gap, we present CubeTrace, a unified, function-level flow tracing system that enables microscopic tracing inside heterogeneous cloud gateways. CubeTrace standardizes tracing units as *cubes* across different hardware platforms, regardless of their varied underlying implementations, and operates at flow granularity for reliability reasons. This introduces a new tracing abstraction for heterogeneous gateways while maintaining low overhead. Moreover, the collected flow-cube data by CubeTrace can be decoded into packet-level representations and integrated with well-established distributed tracing frameworks, enabling the use of off-the-shelf analysis tools. Our evaluations demonstrate that CubeTrace introduces minimal overhead, consuming less than 1% of memory resources and adding less than 1% to forwarding latency. Having been deployed in a large-scale cloud gateway, CubeTrace has significantly improved problem localization, reducing resolution times from hours or even days to just minutes.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Maintainability and maintenance**; • **Networks** → **Network reliability**; **In-network processing**; **Middle boxes / network appliances**.

KEYWORDS

Network Tracing; Cloud Gateways; Performance Diagnosis

ACM Reference Format:
Yunming Xiao, Yinchao Yang, Jiaqi Zheng, Xuqian Li, Dongbo Gu, Jun Zhang, Miantao Wan, Chao Pei, Chen Tian, Mingwei Xu, Ang Chen, and Congcong Miao. 2026. CubeTrace: Microscopic Network Tracing for Heterogeneous Cloud Gateways. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3789240.3829165

*Congcong Miao is the corresponding author.

SIGCOMM '26, August 17–21, 2026, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/26/08
https://doi.org/10.1145/3789240.3829165

1 INTRODUCTION

The cloud gateway is a crucial component of data center network infrastructure, integrating a wide range of stateless and stateful network functions (e.g., routing, load balancing, firewalls, and SNAT) [74, 85, 98, 100]. The diversifying functions coupled with rising performance requirements have pushed modern gateways to transition away from using simple commodity routers. For performance and flexibility, modern gateways, including those in our production networks, comprise a *heterogeneous* platform that combines (*i*) a software server for slow-path processing, (*ii*) a programmable switch for fast-path forwarding, and (*iii*) FPGAs that augment the memory of programmable switch [74, 98].

While the heterogeneous architecture is here to stay, in-network complexity also introduces reliability challenges – left unaddressed, they will significantly offset the benefits of modern gateways. For example, when a flow experiences persistent packet losses, is it due to the routing module, load balancer, or firewall? Or, is a latency spike caused by a misconfigured DPDK function, a bottleneck in the programmable switch pipeline, or inefficiency in the FPGA module? Traditionally, network-level problems are tracked and diagnosed using tracing tools [54, 105–107], which operate akin to 'traceroute' that gathers hop-by-hop information from each device. While this has worked well for traditional network devices, which are opaque entities with relatively simple forwarding logic, cloud gateways are far more complex, general-purpose, and enclosing diverse software and hardware components. None of the above problems can be captured with coarse-grained, device-level trajectories.

We call this need *microscopic network tracing*, which recognizes the complexity of modern gateways and inspects the internals of these multi-function, heterogeneous components using probes. Microscopic traces afford a granular view into the gateways, able to locate a wider range of root causes whether for packet drops or performance degradation.

We observe that elsewhere, the need for granular tracing has given rise to modern tools such as uprobe [10], which can trace function-level details for a single device and generate dynamic call graphs, and Xtrace [38] and Dapper [84], which track granular information across devices. However, these tools cannot be applied to heterogeneous gateways, as they face two major challenges.

The first challenge is the need to accommodate diverse hardware platforms. Since gateways are composed of general-purpose CPUs, programmable switches, and FPGAs, each comes with distinct observability constraints. Function-level tracing (e.g., uprobes [10] or DPDK statistics [90]) assumes flexible logging/probing support on the host. Programmable switches and FPGAs, by contrast, lack per-packet, per-function logging and offer only restricted counters

or event export. As a result, CPU-style, instruction/function-level tracing cannot be uniformly applied across devices.

The second challenge is consistent and reliable tracing. Distributed tracing frameworks like Dapper [84] rely on in-band context propagation, *i.e.,* embedding tracing information in the original requests/packets, across different components/devices; however, it operates above transport-layer protocols like TCP and hence ignores reliability issues like packet losses. XTrace [38] instead propagates context in-band and diagnoses loss via out-of-band reports. However, single-packet granularity in modern gateways would imply per-packet reporting, which incurs infeasible overhead.

To address heterogeneity, we introduce a unified tracing model where the smallest tracing unit, termed a *cube*, can represent a function on a CPU, a forwarding stage on a programmable switch, or a module on an FPGA, serving as the "narrow waist" of microscopic tracing. Cubes are interconnected with directed edges that represent their causal relationships (across hardware boundaries) and record flow-level information as packets traverse them. We refer to the tracing system on this tracing abstraction as CubeTrace.

Moreover, instead of propagating context in-band, CubeTrace relies on out-of-band reports emitted by cubes at flow granularity. Each cube exports aggregated metrics – such as packet/byte counts and latency distributions – keyed by (flow, cube). This shifts the subject granularity from per-packet to per-flow, eliminating the need for per-packet context while retaining enough detail for fine-grained diagnosis within hardware limits.

After having the new abstraction, a third challenge then arises in decoding the new tracing abstraction. Although a cube graph superficially resembles a dynamic call graph or trace tree, the subject granularity differs, *i.e.,* from per-packet to per-flow. Aggregating packets into flows introduces information loss, so exact inversion may be infeasible. More specifically, we find that a CubeTrace graph may be *ambiguous*, meaning that it may correspond to multiple distinct dynamic call graphs or trace trees. To solve this, we come up with a new algorithm to remove such ambiguity.

Overall, we present CubeTrace, a unified, function-level flow tracing system that enables *microscopic tracing* inside heterogeneous cloud gateways. Its design is inspired by distributed tracing, but addresses unique challenges in heterogeneous cloud gateways. And despite the difference in granularity during data collection, we show that flow-cube traces can be decoded into dynamic call graphs or trace trees under specific conditions. This capability minimizes the need for developing new analysis algorithms for CubeTrace, allowing us to leverage the robust features of existing distributed tracing frameworks for data analysis.

We have implemented CubeTrace within our cloud gateway with more than 300 tracing cubes. Our evaluations show that CubeTrace introduces minimal overhead. Specifically, tracing on the programmable data plane (switch and FPGA) adds no additional latency and consumes less than 1% of the total memory resources. In DPDK, CubeTrace uses a negligible amount of memory and, in the most demanding scenario – where packet sizes are large and all packets require tracing – consumes up to 8% of CPU cycles. Even under such conditions, the increase in forwarding latency remains under 1%, which is highly acceptable given the significant benefits CubeTrace provides. Moreover, CubeTrace's flow-cube granularity for data collection reduces communication costs by up to six orders of magnitude compared to finer-grained approaches.

Finally, we show that CubeTrace dramatically enhances problem localization in production, reducing resolution time from hours to minutes. It also enables effective monitoring of gateway forwarding performance through intuitive data visualization and analysis.

This work **does not** raise any ethical issues.

2 BACKGROUND

2.1 Cloud Gateway and Its Evolution

Cloud gateway is a key component of data center infrastructure, serving as the pivotal link that facilitates seamless and secure data movement through various network segments, e.g., virtual machines, private clouds, cross-region data centers, and the Internet. To address a broad spectrum of data transmission needs, cloud gateways integrate numerous critical network functionalities, including routing and virtual routing, load balancing, SNAT, VPN, direct path routing, firewalls, and more [37, 59, 67–69, 74, 82, 85, 93, 95, 98, 100].

Driven by the need to handle expanding workloads [13, 32, 74] and to reduce forwarding delays for time-sensitive flows [39, 49, 73], the architecture of cloud gateways has experienced substantial evolution. Initially, cloud gateways are supported by a series of commodity routers. Later, they are virtualized with multiple network functions consolidated into nested chains for reduced latency, and efficient network stacks like DPDK [1] are adopted. More recently, programmable switches [6] have gained attention for their superior performance. However, their limited memory capacity [14] restricts their ability to support memory-intensive network functions. To address this limitation, various approaches have been proposed to extend their memory capabilities, including integration with commodity servers [20, 52, 55, 74] or FPGAs [22, 98], enabling these devices to support more comprehensive network services.

Our cloud gateway has adopted an extended programmable hardware architecture, as shown in Figure 1. It is composed of a programmable switch enhanced with two FPGAs to increase memory capacity, complemented by a server tasked with processing the initial packets of a flow before offloading subsequent packets to the programmable switch, as well as handling traffic necessitating complex processing logic. Notably, this architecture reduces the cost per Gbps of traffic by 76%, while improving tail latency by 95% at the 99th percentile and 90% at the 99.9th percentile.

2.2 Cloud Gateway Tracing Demands

While integrating diverse network functions and specialized hardware significantly improves cloud gateway's capability and cost efficiency, it also increases architectural complexity, introducing new failure modes that can compromise reliability and violate SLAs. As a result, microscopic network tracing capability is indispensable. Below we list problems that the tracing system needs to address:

- Packet loss or flow failure at cloud gateway. Modern gateways comprise multiple stateless and stateful functions with complex processing logic. Even a single component, such as the DPDK load balancer in our gateway, exhibits 15 well-defined packet drop scenarios (Table 1), not mentioning the undefined drops caused by software bugs. Effective tracing must precisely localize

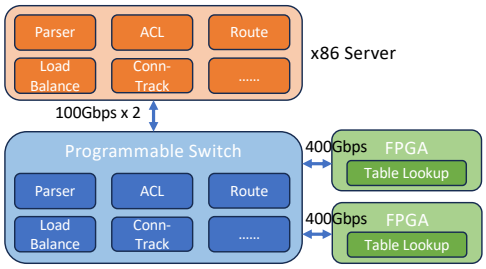


Figure 1: Architecture of our cloud gateway node.

packet loss to a specific DPDK function, switch stage, or FPGA module, and distinguish among different drop causes.

- Packet loss or flow failure outside cloud gateway. Tracing must also support failures that manifest beyond the gateway itself, such as packets rejected by the receiver due to incorrect gateway processing. To enable application-level debugging, the tracing system should reveal the sequence of functions traversed within the gateway and the packet counts at each stage, supporting diagnosis of retransmissions and end-to-end failures.
- Root cause of performance degradation. Beyond correctness, tracing must expose performance anomalies. In our production gateways, tail packet processing delays can be two orders of magnitude higher than expected (e.g., ~10 ns → ~1 ms). Identifying such bottlenecks requires tracking per-function processing latency to pinpoint slow stages and guide optimization.
- Problem localization across gateway nodes. Flows may be processed by multiple gateway nodes. For instance, the "to" and "from" traffic of a flow is often handled by different nodes, as their five-tuples differ due to the source and destination IP addresses and ports being swapped. Additionally, debugging may be required for applications that generate multiple flows simultaneously. Therefore, flow tracing must support analysis across multiple gateway nodes to ensure comprehensive visibility.

Existing network tracing systems fall short of meeting these demands. Many rely on treating each network hop as a black box, using minimal consensus protocols like ICMP (as in traceroute) to infer high-level data at each hop. More recently, endpoint packet inspection and in-band network telemetry (INT) [54] leveraged the programmability of modern network devices, collecting more fine-grained metrics such as switch queuing lengths and buffer sizes, for comprehensive cloud-based analysis [19, 45, 58, 96, 104–107]. However, they still offer limited spatial granularity, operating at device level, and fails to meet our microscopic tracing demands.

Instead, we draw inspiration from elsewhere. First, tools for program debugging utilities like Linux's uprobe library [10] supports function-level tracing by dynamically attaching probes to functions. Similarly, the Linux Trace Toolkit Next Generation (LTTng) enables detailed instrumentation of DPDK libraries [23], while DPDK's native tracing libraries also provide efficient, low-overhead function-level tracing [2]. Additionally, eBPF [3] offers a flexible solution for tracing kernel and user-space functions. It is noteworthy that, as our cloud gateway nodes feature heterogeneous hardware platforms communicating via network traffic, tracing requirements extend beyond individual "servers." Tools like Dapper [84], which propagate trace context across distributed systems, better align with these broader demands.

Table 1: Pre-defined packet drop flags for the DPDK load balancing handler in our cloud gateway.

No.	Packet Drop Flag	No.	Packet Drop Flag	No.	Packet Drop Flag
1	exceed_cc_max_sess	6	quota_svc_conn	11	get_v2v_bip_error
2	cluster_id_failed	7	get_svc_failed	12	get_to_vpc_error
3	quota_bps_pps_failed	8	quota_cc_conn	13	do_reset_failed
4	private_link_cascade	9	get_rs_error	14	do_nat_failed
5	exceed_svc_max_sess	10	get_rbip_error	15	sg_deny

However, none of these solutions meet the demands of tracing in heterogeneous cloud gateway, as they are designed for a single hardware platform, primarily general-purpose CPUs, and do not support programmable switches or FPGAs. This highlights the need for a unified, fine-grained flow tracing system tailored to heterogeneous cloud gateways.

2.3 Goals and Challenges

Goal. We aim to develop a network tracing system that enables *microscopic tracing* inside modern cloud gateways characterized by their heterogeneous hardware components. However, this goal presents a series of challenges.

Challenge 1: Accommodating diverse hardware platforms. The most critical challenge of developing a unified tracing and analysis system comes from the need to accommodate diverse hardware platforms. This requires advancing beyond conventional tracing tools to accommodate diverse programming paradigms: from C in DPDK, and P4 in programmable switches, to Verilog for FPGAs.

More importantly, the system must navigate the unique constraints inherent to each platform. For example, P4 and FPGA compilers often lack the flexible logging capabilities available in DPDK and thus cannot track traffic at the per-packet granularity. Also, programmable switches cannot generate timestamps during execution, yet performance metrics are essential. These differences in languages and hardware limitations present significant challenges in deriving a unified tracing abstraction across all platforms.

Challenge 2: Consistent and reliable tracing. Ensuring consistent flow tracing across heterogeneous hardware platforms and multiple gateway nodes is crucial, as inconsistent data undermines effective issue diagnosis. A simple approach of setting start and end times for tracing is insufficient because: (*i*) packets still being processed at the end time may be missed, causing inconsistencies; and (*ii*) system clock variations across hardware platforms can lead to timing discrepancies, resulting in inaccurate analysis.

Reliability is equally crucial, yet cloud gateways lack key mechanisms leveraged by existing white-box solutions. For example, program debugging tools utilize runtime context to obtain extra program information, while distributed tracing frameworks usually operate above transport-layer protocols like TCP and therefore disregard packet loss. In contrast, flow tracing in heterogeneous cloud gateways lacks these benefits. Programmable hardware and device interconnections do not provide execution context or debugging metadata. Moreover, packet-level tracing occurs below the transport layer, where packet loss is unavoidable – and detecting such losses is a primary tracing objective (§ 2.2). Overall, this presents a unique challenge to realize our envisioned tracing system.

Challenge 3: Decoding the new tracing abstraction. The heterogeneity of cloud gateways, along with strict consistency and

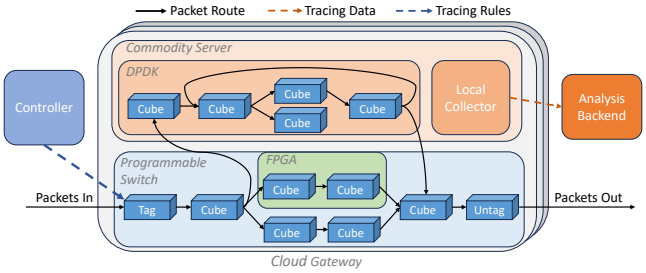


Figure 2: High-level overview of CubeTrace. The controller installs tracing rules and assigns trace IDs. Matching packets are tagged at gateway ingress, processed by cubes across different hardware platforms, and untagged before leaving the gateway. The local collector collects tracing data from cubes and forwards it to the analysis backend.

reliability requirements, may necessitate a new tracing abstraction that diverges from existing models like call graphs or trace trees. This raises the need for new algorithms to decode information from this abstraction. A promising direction is to determine whether the new abstraction can be mapped to existing structures, such as call graphs or trace trees. This allows reusing existing features, significantly reducing development costs and improving accessibility.

3 SYSTEM DESIGN

CubeTrace introduces a new tracing abstraction named "cube" that accommodates three diverse hardware environments: commodity servers, programmable switches, and FPGAs. This section begins with a high-level overview of CubeTrace (§ 3.1), followed by a detailed discussion of its core designs, including the tracing abstraction and reliability (§ 3.2), consistency (§ 3.3), and how to decode cube traces to be compatible with existing distributed tracing framework (§ 3.4). Finally, we clarify limitations in § 3.5.

3.1 High-Level Overview

Tracing abstraction: cubes. CubeTrace introduces a unified tracing abstraction, called a *cube*, to enable fine-grained tracing inside heterogeneous cloud gateways composed of commodity servers, programmable switches, and FPGAs. A cube represents the smallest observable processing unit on a given platform: a software function in DPDK, a match-action table (or pipeline stage) in a programmable switch, or a hardware module in an FPGA.

Crucially, CubeTrace does not trace individual packets. Instead, each cube aggregates packet statistics that match a given tracing rule while traversing that processing unit. A tracing rule defines the *subject of aggregation* (e.g., a 5-tuple flow, a destination IP prefix, or any operator-defined filter). As a result, a cube logically corresponds to a <processing unit, tracing rule> pair and maintains flow-level aggregated statistics, rather than per-packet records.

Each cube exports two classes of metrics: (i) packet counters, which record how many packets matching the tracing rule traverse the cube; and (ii) performance statistics, such as the minimum, average, and maximum processing latency at the cube. These metrics are sufficient to localize packet losses, infer forwarding paths, and diagnose performance bottlenecks, while remaining feasible under the hardware constraints of programmable switches and FPGAs.

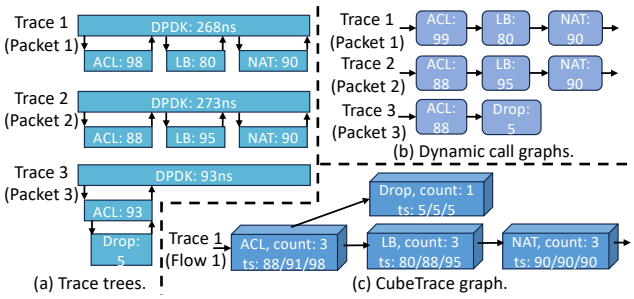


Figure 3: Tracing abstraction comparison. In heterogeneous cloud gateways, per-packet tracing ((a) and (b)) is infeasible; we instead introduce the per-flow *cube* abstraction (c).

Figure 3 demonstrates this concept with an example where a flow of three packets is processed by DPDK, where only three functions are present for the sake of simplicity. In both dynamic call graphs and trace trees, each packet generates its own trace. In contrast, CubeTrace produces a single trace that aggregates all three packets. Figure 2 illustrates the cube abstraction in our production gateway, which consists of more than 300 cubes across three hardware platforms.

Tracing granularity and design rationale. CubeTrace deliberately operates at flow-level (or rule-level) granularity, rather than packet-level granularity. While packet-level tracing provides maximal fidelity, it is infeasible in heterogeneous gateways: programmable switches and FPGAs cannot maintain per-packet state or emit per-packet logs, and in-band context propagation is unreliable below the transport layer due to packet loss.

By aggregating packets into cubes keyed by tracing rules, CubeTrace avoids per-packet state while preserving the information necessary for operational debugging. Packet counters allow CubeTrace to localize packet drops between cubes, while aggregated latency statistics reveal performance anomalies and bottlenecks along processing paths. § 3.2 further justifies this design choice and analyzes the scalability implications of finer-grained alternatives.

End-to-end tracing workflow. Figure 2 shows the end-to-end workflow of CubeTrace, which consists of four components: tracing cubes, a local collector, a global controller, and an analysis backend. It operates as follows:

- An operator specifies tracing rules that define the traffic of interest (e.g., a 5-tuple flow or all traffic to a destination IP). Each rule is assigned a unique trace ID and distributed by the global controller to all gateway nodes.
- When a packet enters the gateway, it is tagged with the trace ID if it matches a tracing rule. As the packet traverses the gateway, each cube recognizes the tag and updates its local aggregated statistics. The tag ensures consistency across heterogeneous hardware platforms and across gateway nodes (§ 3.3).
- After tracing completes, the local collector retrieves cube statistics from software and hardware components, omits inactive cubes, and uploads the results to the analysis backend.
- Although CubeTrace collects aggregated flow-cube data, it can decode resulting graph into packet-level trace trees or dynamic call graphs when the cube graph is ambiguity-free (§ 3.4). This allows CubeTrace to interoperate with existing distributed tracing frameworks and reuse their analysis and visualization tools.

3.2 In Search of a Tracing Abstraction

A key challenge in designing a tracing system for heterogeneous cloud gateways is identifying a tracing abstraction that is simultaneously expressive, reliable, and implementable across diverse hardware platforms. This section explains how CubeTrace arrives at its flow-level cube abstraction by examining the primitives shared by commodity servers, programmable switches, and FPGAs, and by ruling out finer-grained alternatives.

Despite their differences in programming models and capabilities, we identify the three hardware platforms used in modern cloud gateways share a limited set of tracing-relevant primitives during packet processing: (*i*) Packet header modification: All platforms can read and modify packet headers as packets traverse the data path. (*ii*) Packet counting: Each platform can increment counters associated with specific processing units (e.g., functions, tables, or modules). (*iii*) Limited performance statistics: While commodity CPUs can record timestamps precisely, programmable switches and FPGAs cannot do so at runtime. At best, they support coarse or profiled latency information.

CubeTrace must be built using only these primitives, as relying on richer mechanisms available on CPUs alone would break portability and consistency across platforms.

Packet header modification alone is insufficient. Packet header modification naturally suggests in-band packet tracing, where each packet carries its tracing context as it traverses the gateway. Indeed, CubeTrace uses header modification to carry a trace ID inside the gateway, but only as a stable aggregation key (§ 3.3). The trace ID does not encode packet history or per-module tracing results. Full in-band packet tracing by header modification is fundamentally unsuitable for heterogeneous cloud gateways for two reasons.

First, there is a reliability issue under packet loss. Tracing in cloud gateways occurs below the transport layer, where packet loss is unavoidable and often the very failure being diagnosed. Programmable switches and FPGAs cannot snapshot packet headers at intermediate stages due to memory constraints; tracing data can only be extracted at the end of the pipeline. Consequently, if a packet is dropped before reaching the final stage, all in-band tracing context carried by that packet is irretrievably lost. This makes header-only packet tracing unreliable precisely in the scenarios where tracing is most needed.

Second, there are scalability constraints. Even if reliability were not an issue, embedding per-packet tracing context in packet headers and exporting it out-of-band would incur prohibitive overhead. Per-packet tracing scales with traffic volume and quickly overwhelms data-plane resources and analysis pipelines. Appendix A quantifies these costs.

Thus, we avoid packet-level tracing with this primitive.

Flow-level aggregation as the narrow waist. Given these constraints, CubeTrace adopts flow-level (or rule-level) aggregation as the narrow waist of its tracing abstraction. Instead of tracing individual packets, CubeTrace aggregates packets that match the same tracing rule as they traverse each processing unit. For each cube, CubeTrace maintains: (*i*) a packet counter, representing how many packets with a given trace ID traverse the cube; and (*ii*) aggregated performance statistics, including minimum, average, and maximum processing latency.

Table 2: Summary of tracing granularity.

Subject \ Spatial	Function	Device
Packet	Ideal but not feasible for programmable switch/FPGA	E.g., traceroute/INT; No visibility into internal gateway functions
Flow	CubeTrace; satisfying all our tracing demands	E.g., NetFlow; suitable for simple switch/router

This aggregation sacrifices per-packet ordering and per-packet causality, but preserves the information required for the operational debugging tasks described in § 2.2:

- Packet loss localization: By comparing packet counts across adjacent cubes, CubeTrace can identify packet drops at specific functions, between processing stages, or at hardware boundaries.
- Forwarding path inference: The cube graph encodes causal relationships between processing units. Under specific structural conditions (§ 3.4), aggregated cube data can be decoded to reconstruct packet-level forwarding paths.
- Performance diagnosis: Aggregated latency statistics reveal bottlenecks, even when fine-grained timestamps are unavailable.

Importantly, the number of cubes scales with the number of processing units, not with the number of packets/flows.

Scope and limitations. Aggregation in CubeTrace is a deliberate design choice. CubeTrace does not preserve per-packet ordering or per-packet latency distributions and is therefore not intended for fine-grained protocol debugging (e.g., TCP handshake analysis). Instead, it targets operational debugging tasks inside cloud gateways, where identifying where packets are dropped or which functions dominate latency is the primary goal. Table 2 summarizes different granularities and their applicabilities.

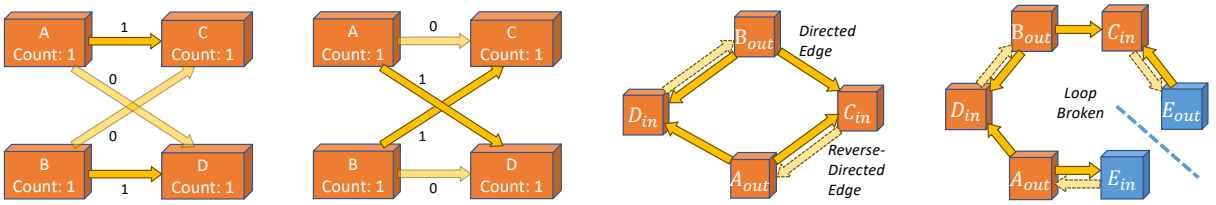
3.3 Consistent Flow Tracing

Having established flow-level cube aggregation as the tracing abstraction, the remaining challenge is to ensure consistency: all packets belonging to the same tracing rule must be aggregated coherently across heterogeneous hardware platforms and across gateway nodes. CubeTrace achieves this through lightweight trace-ID tagging via packet header modification.

Trace IDs and tracing rules. A tracing rule specifies the traffic subset to be traced. A rule may match a standard 5-tuple flow or a broader traffic class defined by an arbitrary combination of packet header fields (e.g., destination IP prefix, protocol, or post-NAT address). Each tracing rule is assigned a unique, fixed-size trace ID by the global controller.

The trace ID serves as a stable identifier for aggregation, not as an execution context. It does not encode packet history, timestamps, or per-packet state. All packets matching the same tracing rule carry the same trace ID and are aggregated together by cubes. The global controller distributes the set of active tracing rules and their associated trace IDs to all gateway nodes, ensuring that trace ID semantics are consistent across the deployment.

Packet tagging and propagation. CubeTrace enforces a single logical entry point for tracing at each gateway node. When a packet arrives at the gateway, it is first processed by an entry-stage cube that matches the packet against the active tracing rules. If a match is found, the packet is tagged with the corresponding trace ID by overwriting a designated packet header field.



(a) Example of graph ambiguity. (b) Example of graph ambiguity. (c) A loop in G' indicates presence of ambiguity in the original graph G. (d) General solution to resolve an ambiguity by adding a vertex to the loop.

Figure 4: Illustrations of ambiguity in a directed graph and the resolution.

The trace ID tag is then propagated unchanged as the packet traverses the gateway across all processing stages and hardware platforms. Each cube uses the trace ID to index its local aggregation state and update packet counters and performance statistics.

At the final stage of gateway processing, packets are untagged by restoring the modified header field to its originally expected value: the gateway's source MAC address (see below). As a result, trace IDs are visible only inside the gateway and are completely transparent to the rest of the network.

In our deployment, we use the source MAC address field (`src_mac`) for tagging, as it is irrelevant beyond layer-2 forwarding within the gateway. The choice of header field does not affect CubeTrace's abstraction; it is an implementation detail constrained by the deployment environment.

Consistency under rule updates and concurrency. A key advantage of packet tagging is that it decouples data-plane consistency from control-plane updates. Tracing rules are installed and removed only at the entry-stage cube. Once a packet is tagged, its trace ID remains unchanged throughout its lifetime in the gateway, regardless of subsequent rule updates. As a result, packets already in flight are always aggregated consistently, and enabling or disabling tracing rules does not affect partial traces.

CubeTrace can support multiple tracing rules simultaneously. Each active rule has a distinct trace ID, and cubes maintain a separate aggregation state per trace ID. In our implementation, the programmable switch supports over 1K concurrent tracing rules, sufficient for operational use.

Consistency guarantees. This tagging-based design provides strong consistency guarantees at multiple levels. First, it ensures per-packet consistency: every packet matching a tracing rule is deterministically associated with exactly one trace ID throughout its entire processing pipeline. Second, it achieves cross-platform consistency, as the same trace ID is interpreted uniformly by cubes implemented in software, programmable switches, and FPGAs. Third, it provides cross-node consistency: since tracing rules and trace IDs are distributed globally, packets belonging to the same logical flow are aggregated coherently even when they are processed by different gateway nodes. CubeTrace does not require synchronized clocks across hardware platforms and does not rely on start or end time windows for tracing; instead, all aggregation is driven solely by trace IDs and packet counts.

3.4 Decoding CubeTrace Data

After data collection, the next step is to analyze the traces for failures and performance anomalies. Although the new abstraction could require entirely new analysis tools, we instead seek to decode CubeTrace traces into dynamic call graphs or trace trees, enabling direct reuse of existing, well-established analysis frameworks. At first glance, this decoding appears straightforward, since CubeTrace data aggregates such traces. However, aggregation introduces a fundamental challenge that complicates decoding. We describe this challenge and our solution below.

Problem statement. *A CubeTrace graph may correspond to multiple dynamic call graphs or trace trees.*

This problem arises due to information loss during the aggregation process for generating CubeTrace data, making exact reversal impossible. For example, while (c) in Figure 3 can be derived from (a) and (b), reversing the process loses specific details, such as the latency at each node or span for each individual packet. Additionally, in some cases, multiple dynamic call graphs or trace trees may produce identical CubeTrace data, leading to ambiguity in the reversal process.

Figures 4(a) and 4(b) depicts one such scenario, where identical vertex visitation counts (packet counts at cubes) lead to two distinct solutions for edge visitation counts (packet counts between cubes). The number of distinct solutions for edge visitation counts grows as packet volume increases. This complexity impedes our ability to precisely locate packet losses, as it becomes unclear which path discrepancies are responsible for packet drops if there are any. For instance, in Figure 4(a), a packet loss at cube C – where the count becomes zero instead of one – leaves us uncertain whether the loss occurred along the edge <A, C> or <B, C>.

Assessing the tracing graph ambiguity. We observe that the potential for multiple decoding outcomes is a property of the tracing graph, rather than of any individual runtime trace. We refer to this property as graph ambiguity. Below, we devise an algorithm to tackle this problem.

The algorithm unfolds as follows: For a given directed graph $G = (V, E)$, we create a new bipartite directed graph $G' = (V', E')$ where $V' = \{v_{in}, v_{out} | \forall v \in V\}$ and $E' = \{(s_{out}, t_{in}) | \forall (s, t) \in E)\}$. This means that for every vertex $v \in V$, a corresponding pair of vertices v_{in} and v_{out} are created where v_{in} assumes incoming edges originally directed towards v, and v_{out} assumes all of the outgoing edges of v. No edge is present between v_{out} and v_{in} unless there is a self-loop at v in the original graph G.

Next, we execute a depth-first search (DFS) on the newly formed graph G'. During this search, we traverse the graph in a specific pattern: alternating between a directed edge and a reverse-directed edge that is not the immediate reversal of the previously traversed edge. This entails progressing along a directed edge, followed by

moving through a reverse-directed edge — specifically avoiding the reverse of the edge just crossed — and maintaining this alternating sequence throughout the search. If we encounter an even-length loop during this exploration[1], then the original graph G is deemed ambiguous; if no such loop is found, the graph is considered unambiguous. An example of such loop can be found in Figure 4(a) with the following sequence: (B_{out}, C_{in}), (C_{in}, A_{out}), (A_{out}, D_{in}), (D_{in}, B_{out}), as illustrated in Figure 4(c). The time complexity is $O(|V| + |E|)$.

Intuitively, traversing a directed edge (u_{out}, v_{in}) in G' simulates the action of sending a packet from u_{out} to v_{in} (*i.e.,* from u to v in the original graph G). Conversely, moving along a reverse-directed edge (v_{in}, w_{out}) in G' can be interpreted as retracting a packet that was sent from w_{out} to v_{in} (also from w to v in G), ensuring that the visitation count for vertex w_{out} remains unchanged (so is w in G). If a loop is encountered during the DFS, it suggests the presence of an alternate pathway for edge visitation that does not alter the vertex visitation count. In other words, a single set of vertex visitation counts corresponds to multiple edge visitation count solutions. Thus, the original graph is ambiguous. Interested readers can find more details on how to remove the ambiguity in Appendix B. § 5.1 shows how this refinement is applied to our production graph. Once the ambiguity is resolved, decoding the CubeTrace data becomes straightforward. Detailed explanations are omitted here due to space limitations.

3.5 Diagnostic Scope and System Limitations

Supported diagnostic tasks. CubeTrace is designed for flow-level diagnosis across heterogeneous gateway components. Given an ambiguity-free tracing graph, CubeTrace reconstructs flow traces from distributed cube statistics and supports the operational debugging tasks described in § 2.2: forwarding path reconstruction, packet-loss localization, bottleneck identification, and cross-node diagnosis. In particular, cube counters reveal where packet counts decrease along the processing pipeline, while performance statistics identify cubes or paths that dominate forwarding latency.

Information lost through aggregation. The above capabilities come from aggregating packets according to tracing rules and cubes. This aggregation is intentional, but it also loses packet-level information. CubeTrace does not generally preserve per-packet ordering, per-packet causality, or full per-packet latency distributions within a traced aggregate. As a result, CubeTrace is not intended to diagnose fine-grained protocol behaviors such as intermittent packet reordering, nor to reconstruct complete packet-level execution histories for all packets in a flow. Such tasks require complementary packet-level tracing or endpoint debugging tools.

Packet-loss diagnosis. Although CubeTrace does not generally identify the exact packet instance dropped within a large aggregate, packet-level properties may still be inferred in low-cardinality cases. For example, when only one packet traverses a distinguishable path, its loss or latency can often be inferred directly from cube statistics. More generally, CubeTrace can distinguish different loss patterns by comparing traces across granularities: losses recurring across many independently traced flows, or under a coarser tracing rule, indicate uniform loss at a function; losses confined to flows matching a source, destination, or header subset localize the issue to that traffic class. Thus, CubeTrace sacrifices certain packet-level details in exchange for a deployable, consistent tracing abstraction across software, programmable switches, and FPGAs.

[1]Note that detecting an even-length loop within a general directed graph is considered as an NP-Hard problem [25, 30, 89]. However, the graph G' we construct is bipartite, as there are no edges between any two "in" vertices or any two "out" vertices. This guarantees that all loops within G' are of even length. Consequently, the problem is reduced to finding a simple cycle where a polynomial-time algorithmic solution exists.

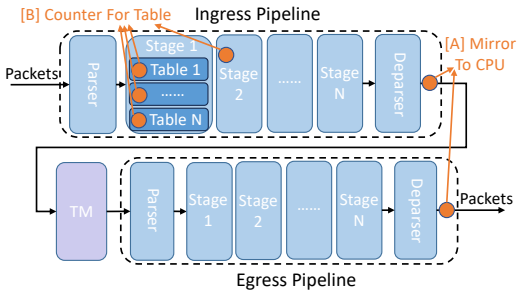


Figure 5: Architecture of programmable switch and two tracing options. Tagging is ignored for simplicity.

4 IMPLEMENTATIONS

In this section, we detail our implementations of logical tracing cubes on different platforms. (§ 3.1). In total, CubeTrace in our cloud gateway consists of 300+ cubes across three hardware platforms.

4.1 Programmable Switch

The programmable switch is tasked with two primary functions: (i) tagging/untagging packets, and (ii) logging cube statistics. § 3.3 has detailed the first task; below, we focus on the second task.

Packet counter. Before going into the details, we briefly introduce the concepts in programmable switches. In the Reconfigurable Match Tables (RMT) switch programming model [26] (Figure 5), a programmable switch is structured with an ingress and an egress `pipelines`, each comprising a series of sequential `stages`. Each stage can host several match-action `tables` arranged in parallel. In CubeTrace, we treat each of these tables as a distinct cube.

To log cube statistics, one approach is to implement an extra custom header that is designed to encapsulate all tracing information. As the packet progresses through the programmable switch, the statistics per cube will be recorded in the header. When the packet reaches the end of each pipeline within the programmable switch, it will be mirrored, where the mirrored header is transmitted to the local collector at the server CPU ([A] in Figure 5). Nevertheless, this does not work because of reliability and scalability issues (§ 3.2).

We thus opt for an alternative approach that involves instrumenting the programmable switch code to include a `counter` array within each stage, serving as the packet counter. Each index within this array corresponds to a specific match-action table, *i.e.,* a cube in CubeTrace ([B] in Figure 5). As packets traverse through a table, the relevant counter is incremented. In cases where a packet is dropped, all previously traversed cubes are recorded, allowing us to infer the drop location. Once tracing for a given rule concludes, the local collector retrieves data from these tracing counters, and reset them for future tracing jobs. In general, this approach enables granular monitoring of packet drops across individual cubes, allowing for precise identification of any unexpected packet losses.

Table 3: Profiled latency of programmable-switch pipelines.

Pipe	Latency	# cycles in stages 0–11
0-Ing.	198.4 ns	22 | 22 | 26 | 22 | 30 | 30 | 22 | 20 | 22 | 20 | 1 | 1
0-Egr.	141.0 ns	22 | 22 | 20 | 20 | 20 | 20 | 20 | 20 | 1 | 1 | 1 | 1
1-Ing.	138.6 ns	22 | 22 | 30 | 22 | 22 | 20 | 20 | 2 | 2 | 1 | 1 | 1
1-Egr.	142.7 ns	22 | 22 | 22 | 20 | 20 | 20 | 20 | 20 | 1 | 1 | 1 | 1
2-Ing.	221.4 ns	22 | 2 | 26 | 26 | 22 | 22 | 30 | 22 | 22 | 30 | 22 | 20
2-Egr.	166.5 ns	22 | 22 | 2 | 26 | 2 | 2 | 30 | 22 | 22 | 26 | 22 | 1
3-Ing.	198.4 ns	22 | 22 | 26 | 22 | 30 | 30 | 22 | 20 | 22 | 20 | 1 | 1
3-Egr.	141.0 ns	22 | 22 | 20 | 20 | 20 | 20 | 20 | 20 | 1 | 1 | 1 | 1

Table 4: End-to-end programmable-switch fast-path latency under different offered loads. Latency variations are primarily due to load-induced queueing delay.

Traffic Load	P99 Latency	P99.9 Latency
1 Gbps	~ 5 μs	~ 5 μs
100 Gbps	~ 5 μs	~ 10 μs
300 Gbps	~ 10 μs	~ 20 μs
700 Gbps	~ 10 μs	~ 20 μs

Performance statistics. CubeTrace also aims to enable performance tracing. However, programmable switches lack real-time timestamping capabilities due to hardware limitations, making straightforward logging infeasible. To overcome this, we adopt an indirect approach: by profiling the processing delays for each cube in advance and combining this with per-cube packet counters, we can infer the internal processing paths of packets and conduct detailed performance analysis.

This approach is feasible because packet processing inside the programmable switch follows a fixed line-rate match-action pipeline. Once the P4 program is compiled, each stage's contribution to packet-processing latency is deterministic. Table 3 reports the profiled latency of each ingress/egress pipeline in our deployment. Each row corresponds to one physical pipe and one gress direction; a packet traverses the ingress and egress pipelines on its selected pipe, rather than all physical pipes. The per-stage values show the fixed cycle contribution of each stage to the pipeline latency.

At runtime, packets may additionally experience load-dependent queueing delay due to buffering and scheduling within the switch datapath. To account for this effect, CubeTrace correlates traced paths with concurrent gateway workload measurements. Table 4 reports the measured end-to-end programmable-switch latency under different loads. Thus, CubeTrace separates the deterministic per-cube processing latency from the workload-dependent queueing component, allowing operators to reason about both path-specific processing costs and load-induced latency inflation.

4.2 FPGA

Packet counter. We place a trace counter within each critical module, which is considered as a cube in CubeTrace. When a target packet arrives at a module, this counter increments by one. The trace counter operates concurrently with the packet forwarding logic, ensuring that packet processing is not disrupted. The local collector periodically reads and resets these counters upon the completion of a tracing task.

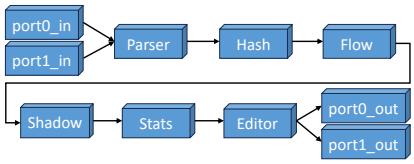


Figure 6: Tracing cubes of FPGA.

Table 5: FPGA's lookup-pipeline latency under different offered loads, excluding Parser and Editor modules. 55 Mpps is the FPGA's no-drop limit.

Load (PPS)	Min	Max	Avg	Relative
0.01M	344 ns	344 ns	344 ns	1.0×
24.0M	536 ns	600 ns	579 ns	1.7×
49.9M	1,144 ns	1,896 ns	1,533 ns	4.5×
56.1M	1,672 ns	1,896 ns	1,795 ns	5.2×

Performance statistics. Like the programmable switch, the FPGA does not provide a practical way to record precise start and end timestamps at every cube for every packet. We therefore profile FPGA module delays offline and use the profiled values as inputs to CubeTrace's performance analysis.

In our gateway, the FPGA is not the dominant contributor to end-to-end forwarding latency. Its role is intentionally narrow: it serves primarily as an extended memory resource for table lookups, rather than performing complex packet processing or frequent packet editing. This design keeps the FPGA pipeline simple and limits latency variation. As shown in Figure 6, the FPGA contains only 10 cubes, making it the simplest component in CubeTrace.

The parser and editor modules use cut-through processing and exhibit little variation. The main latency variation comes from the four middle lookup cubes, where requests may contend for external memory. To keep the presentation concise, Table 5 reports their aggregate latency rather than the per-cube breakdown. Under light load, the lookup latency stays close to the profiled baseline; under high load, DDR contention and backpressure from small internal buffers increase latency. At runtime, CubeTrace interprets FPGA performance by correlating the traced cube path with concurrent workload. It is noteworthy that even near the FPGA's no-drop processing limit, the average lookup latency remains below 2 μs, so FPGA latency is load-dependent but not the main contributor to end-to-end fast-path latency.

4.3 Commodity Server

The server executes two functions: (*i*) packet forwarding and tracing in DPDK, and (*ii*) a local collector that retrieves and aggregates the data for this cloud gateway node.

DPDK tracing. Tracing in DPDK is simpler than in programmable hardware, with various implementations proposed [23, 90, 101, 102]. The native DPDK tracing library [2] offers the best performance among them but generates excessive data and lacks compatibility with our cube abstraction. Therefore, we implement custom DPDK tracing by instrumenting key functions' entry and exit points. As shown in Figure 7, we minimize instrumented code logic to maintain efficiency, using pre-initialized shared memory to avoid unnecessary system calls. Performance metrics are limited to recording min/average/max durations for each cube.

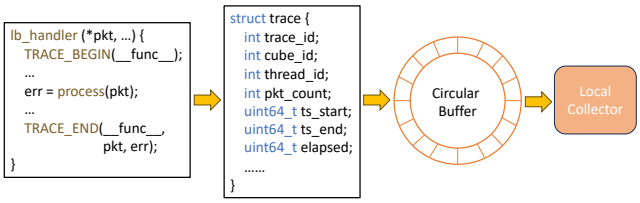


Figure 7: Tracing instrumentation in server DPDK.

Local collector. In parallel to the data plane DPDK process, we run a local collector as an independent process. This collector reads cube data from the shared memory on the CPU, retrieves tracing data from designated counters within the programmable hardware, and collects workload pressure. After collecting the data, the collector will refine it by eliminating cubes that have a zero visitation count, *i.e.,* no packets traversed through these cubes. Then, it transmits the data to the analysis backend using Opentelemetry SDK [4].

5 EVALUATION

5.1 Tracing Graph Ambiguity

As shown in § 3.4, ambiguities in tracing graphs can hinder the analysis due to the aggregated nature of our tracing data. Consequently, after instrumenting the cloud gateway production code, we export the tracing graph – comprising tracing cubes and their connections – and feed it into our ambiguity assessment algorithm (§ 3.4). This evaluation is conducted every time there are changes to the tracing program. If the tracing graph exhibits ambiguity, we resolve it based on the outputs of our detection algorithm. Production tracing is enabled only after the exported cube graph is ambiguity-free.

Below, we demonstrate this process with the assessment of the first draft of our tracing graph, where 303 ambiguity cycles were identified, followed by measures we took to address them.

The detected number of ambiguity cycles may appear alarming at first glance. Fortunately, a detailed analysis reveals that most of them originate from one "hot spot" of ambiguity. Specifically, this arises from the load balance handler in the programmable switch. The handler was divided into 7 parallel tables – and consequently, 7 cubes – due to varying packet origins. Following the handler, packets proceed to a routing logic, which is also divided into 4 tables (and thus 4 cubes) based on the packet's destination type. Consequently, the interaction between these two stages creates a fully connected layer structure, as depicted in Figure 8(a), creating a total of $C(7,2) \cdot C(4,2) + C(7,3) \cdot C(4,3) + C(7,4) \cdot C(4,4) = 301$ ambiguity cycles. Addressing it merely requires inserting a single cube between these two stages, as demonstrated in Figure 8(b). This effectively reduces the ambiguity cycle count to just two.

Figure 8(c) shows one of the remaining ambiguity cycles that occurred post-ACL in the programmable switch, where packets will be forwarded to DPDK for additional processing. Prior to the ACL stage, packets are routed through two paths for data retrieval from two distinct FPGAs, resulting in two parallel ACL cubes based on the FPGA interaction. Meanwhile, there are two ports linking the programmable switch to the DPDK. Since packets from either ACL cube could be sent to DPDK through either port, ambiguity was introduced. To resolve this, we insert additional cubes between the ACL and the programmable switch-DPDK interface (Figure 8(d)). Note that we adopt different ambiguity removal strategies here given the availability of different physical resources. The ambiguity in another location is addressed similarly.

Overall, we discovered that the primary sources of ambiguity were the interconnections between the programmable switch and DPDK, as well as in the forwarding logic of the programmable switch, largely due to our initial efforts to minimize resource usage, *i.e.,* reducing the number of cubes. No ambiguities were detected in the FPGA-related code, and, contrary to our initial concerns, the circulation logic within DPDK was also free from ambiguity. It is still noteworthy that the ambiguity assessment algorithm is designed for general directed graphs, making it robust and adaptable to future complications in cloud gateway logistics.

With the above adjustments being applied, the revised tracing graph is found to be ambiguity-free. We proceed to adopt this updated tracing graph for further evaluation.

Table 6: CubeTrace overhead summary.

Hardware Platform	Forwarding Latency	Resource Consumption
Programmable Switch	+0%	1.25% SRAM Per Stage 25% SALU Per Stage
FPGA	+0%	0.02% Register
Server DPDK	+1%	2% to 8% CPU Cycles Up to 10KB Memory

5.2 Tracing Overhead

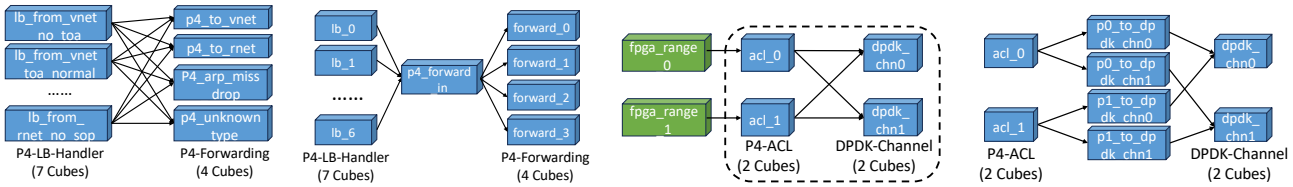
Programmable switch. In the programmable switch, the tracing instrumentation of CubeTrace involves incrementing a counter at each stage. Because the execution time for each stage is predetermined and fixed, the inclusion of tracing operations does not affect the packet forwarding latency.

In terms of resource consumption, CubeTrace needs to use counter resources, which occupy SRAM for storage and SALU for computation. In our programmable switch configuration, CubeTrace uses the minimum allocatable amount of SRAM – 1.25% – and one out of the four SALUs available per stage. In our experience, this allocation leaves sufficient resources for normal network operations, especially since the FPGAs extend the memory space of the programmable switch and handle the most of table lookups (§ 4.2).

FPGA. The tracing counters in FPGAs operate in parallel to other packet processing functions within each module. The tracing counter is clocked at 250 MHz, where each clock cycle can support an increment of the counter. Consequently, this setup enables tracing support for up to 250Mpps, which is five times the typical packet handling capacity of a cloud gateway node. Therefore, the tracing implementation is not the performance bottleneck and does not introduce any additional latency to packet forwarding.

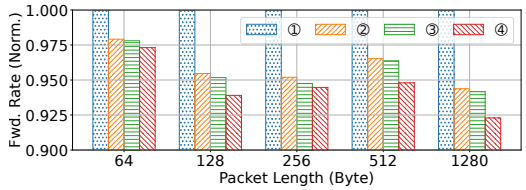
Regarding resource consumption, our implementation of tracing in FPGA consumes 10 tracing counters, each 32 bits in size, totaling 320 register units. Given that the overall number of register units in the FPGA exceeds 1,500,000, the tracing functionality utilizes approximately 0.02% of the available register resources, indicating a minimal impact on the FPGA's resource capacity. Table 6 lists the forwarding latency and resource consumption of CubeTrace on all hardware platforms.

Server DPDK. Finally, we turn our attention to server-based DPDK. In each set of experiments, we send packets with a fixed size –

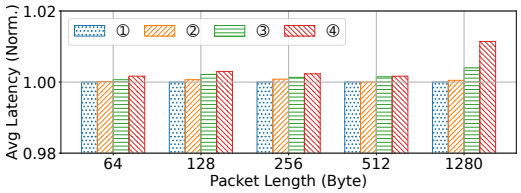


(a) Part of our initial tracing graph. (b) Updated tracing graph. (c) Part of our initial tracing graph. (d) Updated tracing graph.

Figure 8: Ambiguities found in our first draft of tracing graph and solutions to address them.



(a) Packet forwarding rate.



(b) Packet forwarding latency.

Figure 9: DPDK overheads.

from 64B to 1280B – to saturate the DPDK. We further consider 4 scenarios: ① CubeTrace is disabled, ② CubeTrace is enabled but triggered by no flow, ③ CubeTrace is enabled and 50% traffic triggers tracing, and ④ CubeTrace is enabled and all traffic is traced.

CubeTrace incurs memory usage in DPDK; however, this consumption is minimal, amounting to just a few KBs, which is negligible for modern commodity servers. But unlike other hardware platforms, tracing in DPDK introduces additional computational overhead to packet forwarding functions. As shown in Figure 9(a), the forwarding rate decreases by 2% to 5% when CubeTrace is activated but not actively tracing. This rate decreases by an additional 1% to 2% when either some or all traffic matches the tracing rules. Notably, the reduction in the forwarding rate inversely reflects the increase in CPU cycle usage. Therefore, CubeTrace accounts for using 2% to 8% of the CPU cycles for DPDK processing.

CubeTrace may affect DPDK forwarding latencies. According to Figure 9(b), the impact on DPDK's forwarding latency is minimal for packet lengths of 512B or less, regardless of the volume of traffic hitting the tracing rules. The maximum increase in forwarding latency is about 1%, observed when packet length is 1280B and all packets trigger tracing rules.

Note that in production, the proportion of traffic requiring tracing is significantly lower than our assumptions in scenarios ③ and ④. Overall, while the performance impact of CubeTrace in DPDK is more pronounced compared to other hardware platforms, it remains within the acceptable range given its substantial benefits in precise error localization and detailed performance analysis.

Communication overhead. We also evaluate the communication overhead for uploading results to the analysis backend. Overall, CubeTrace incurs low communication overhead; we defer more details to Appendix C.

Table 7: Packet loss analysis in production. The reported localization time is based on historical incident records.

Packet Loss Root Cause	Occurrence	Problem Localization W/O CubeTrace	Problem Localization W/ CubeTrace
Gateway Software Faults	10.5%	Hours	Minutes
Gateway Hardware Faults	2.3%	Hours	Minutes
Third-Party Dependency	25.4%	Hours to Days	Minutes
Misconfiguration	42.4%	Hours	Minutes
Non-Gateway Faults	15.9%	Hours	Minutes
Both Gateway and Non-Gateway Faults	0.9%	Hours to Days	Minutes
Performance Issue	2.6%	Hours	Minutes

5.3 Tracing Analysis In Production

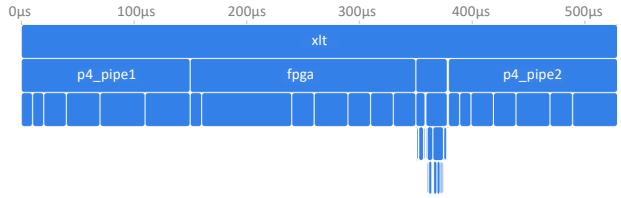
Debugging with CubeTrace. CubeTrace has been deployed in our production data center. Table 7 outlines the root causes of packet losses, their frequency, and localization times with and without CubeTrace. For losses originating within the cloud gateway (top 4 rows), misconfiguration is the most frequent cause (~40%), followed by third-party dependency issues. Gateway software faults account for ~10%, while hardware faults contribute minimally.

As previously discussed (§ 2.2), CubeTrace also aids in debugging upper-layer applications, even when the issue does not stem from the cloud gateway itself. Non-gateway-related root causes account for 15.9% of confirmed cases, and in rare instances (0.9%), both the gateway and upper-layer applications must work together to resolve packet loss.

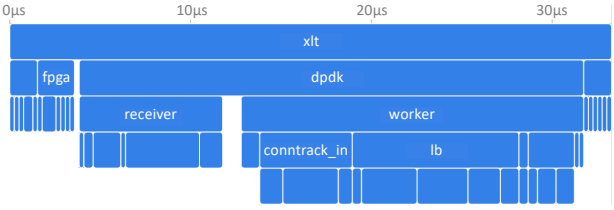
Regardless of the root cause of the packet loss, CubeTrace significantly shortens the time needed for problem localization, reducing it from hours or even days to within an hour, thanks to its automated, fine-grained flow tracing and intuitive data visualization which we demonstrate below.

Performance analysis with CubeTrace. We first demonstrate CubeTrace's data visualization capabilities with a sample flow. Figure 10(a) shows a flame graph from the analysis backend interface, illustrating the cumulative time spent at each cube during the flow's lifecycle. Most packet processing occurs in the fast path, including the programmable switch ("p4_pipe1" and "p4_pipe2") and the FPGA, with the switch consuming more time. In contrast, DPDK accounts for only a small fraction of the processing time.

Next, we analyze per-path forwarding performance by decoding CubeTrace data into trace trees (§ 3.4). Typically, the first packet of a flow traverses the slow path in DPDK, where the routing decision is made. Forwarding instructions are then offloaded to the



(a) Flame graph of flow performance.



(b) Flame graph of slow path performance.

Figure 10: Performance measurements by CubeTrace.

programmable data plane, which handles subsequent packets via the faster path. Figure 10(b) shows the flame graph for slow-path forwarding, where DPDK dominates (~80%) processing time, handling firewall, load balancing, routing, and more functions, while the programmable switch and FPGA contribute minimally.

Figure 11 provides a heat map of processing durations across cubes for 100 sampled production flows (Y-axis) and cubes (X-axis). It reveals flow-specific cube traversal patterns, with generally consistent processing times per cube, though some deviations occur. This highlights the complexity and variability of cloud gateway operations, making CubeTrace essential for performance analysis.

Finally, Table 7 highlights occasional performance issues in production, such as tail processing time spikes (e.g., from 50ns to 5ms). CubeTrace effectively captures these anomalies using preset tracing rules or by reproducing them with defined rules. Root causes, such as inefficient segmentation reassembly in DPDK or hash-based route table lookups, were identified and resolved using CubeTrace.

Case study I: post-handshake RST debugging. We report a production case study of debugging with CubeTrace. We noticed that clients intermittently received a TCP RST and failed to connect to the Redis service. Host-side analysis reveals that the client observed a successful handshake followed by an immediate reset, while the Redis receiver (RS) observed application traffic with an unexpected source port and issued the RST.

We replayed the offending traffic and enabled a targeted tracing rule for the affected flow. CubeTrace reconstructed per-path traces and counters for the handshake packets and the subsequent application packets. The handshake packets completed normally through the slow-path session manager, indicating that session creation succeeded. However, the subsequent packets followed a different path involving fast-path NAT. This path divergence localized the investigation to the handoff between slow-path session management and fast-path NAT state, rather than unrelated ACL, routing, or application components.

Guided by this localization, we exported the targeted session-manager state, including SNAT mappings, reuse decisions, and pending deletions. The snapshot showed a reuse-then-delete sequence for a previously seen five-tuple, yet packets observed at

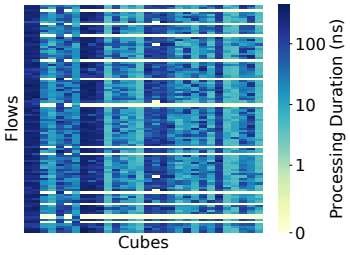


Figure 11: Heat map of cube latencies in production.

the RS still carried a stale SNAT port. Code inspection confirmed that the fast-path deletion used an incorrect key, leaving the stale hardware entry in place. Subsequent packets were therefore NATed to the wrong source port, causing RS to issue RSTs.

Overall, this case study shows how CubeTrace uses path reconstruction and cube counters to narrow failures across slow-path and fast-path components, then guides targeted state inspection to identify the root cause. Here, CubeTrace localized the issue to stale fast-path NAT state and pinpointed a failed hardware session deletion. Without CubeTrace, engineers would typically replay traffic on a separate gateway replica and inspect logs/code across both paths, which can take hours.

Case study II: RST storming. We report another production incident where a gateway cluster raised packet-loss alarms during a sudden traffic surge: aggregate load increased from below 20 Mpps to about 160 Mpps, accompanied by a noticeable rise in drop rate.

After we noticed the issue, we enabled CubeTrace with a broad tracing rule matching all traffic. Flow-level counters showed that nearly all of the additional 160 Mpps traffic missed hardware sessions and was directed to the software slow path, where the "no-session / non-SYN" drop counter rose sharply. Path reconstruction revealed a consistent processing sequence: hardware session miss, software session miss, failed session creation because the TCP packet carried no SYN flag, packet drop, and TCP RST generation.

We then traced representative dropped flows and found that the offending traffic itself consisted largely of TCP RST packets. Upon receiving these RSTs, the gateway again failed session lookup and entered the same drop-and-reply logic, generating additional RSTs. This indicated that the traffic surge was caused by an RST amplification loop rather than client traffic or a forwarding failure.

To identify the source of these RSTs, we extended tracing across gateway clusters and reconstructed the end-to-end path: client → gateway node A → gateway node B → Redis server. Drops were visible at both gateway nodes. Since both gateways generated RSTs when no session existed or session creation failed, we immediately disabled reply-RST-on-no-session. The gateways then simply dropped such packets, and the storm stopped.

After mitigation, we correlated CubeTrace's path/counter evidence with operation and control-plane logs. The root cause was an ongoing LB2 service migration from gateway B' to gateway B without migrating existing sessions; we reproduced the scenario in a test environment to confirm the diagnosis.

During this incident, CubeTrace played a major role in helping with quickly identifying the causes of the RST storming and allows engineers to stop the incident within minutes, effectively maintaining our data center's SLA.

6 DISCUSSION

Compatibility with other network tracing systems. Existing network tracing systems focus on cloud-wide tracing [36, 96, 104–106], using custom pipelines and algorithms for analysis. Integrating CubeTrace with these systems can follow two approaches: (*i*) adapting their custom data processing pipelines to CubeTrace's distributed tracing framework, or (*ii*) modifying CubeTrace's outputs to be compatible with existing systems. While the first approach is preferable, challenges such as data structure and granularity mismatches remain. We leave further exploration to future work.

Generality and applicability. CubeTrace is not tied to our exact CPU/switch/FPGA deployment. A cube represents the smallest observable processing unit exposed by a gateway implementation, e.g., a software function, switch table/stage, FPGA module, or SmartNIC block. This abstraction matches the trend toward heterogeneous gateway/NF acceleration [59, 74, 97, 98]. Porting CubeTrace to a new deployment requires developers to instrument tracepoints at function/module boundaries and export the resulting cube graph for ambiguity checking and refinement before deployment. CubeTrace assumes only minimal primitives: traffic identification, per-cube counters, and runtime latency measurement where available or profiled latency otherwise. We acknowledge that the fidelity of latency profiling may be limited on FPGA and depends on the design: simple pipelines as in our deployment can be profiled, while deeper or more contended FPGA datapaths can exhibit large workload-dependent latency variance and may require coarser attribution or additional telemetry. Conversely, platforms with richer telemetry primitives, such as CPUs and some SmartNICs, could enable finer cubes or more accurate latency measurement. We leave a systematic study of target-specific mitigations/enhancements to future work.

Future evolution of cloud gateway. While Intel Tofino series [6] has established itself as one of the widely used programmable switches, Intel's decision to cease the development of the Tofino line has cast a shadow of uncertainty over the future of this technology [5]. In the wake of this development, there are two potential replacements for Tofino: (*i*) other programmable switch chips, such as Trident4 [9], and (*ii*) SmartNICs [15, 16]. Regardless of the specific technology adopted, CubeTrace is designed to be adaptable, *i.e.,* tracing cubes can be realized on other programmable switches and SmartNICs. This demonstrates the broad applicability of CubeTrace for the evolving network infrastructure.

7 RELATED WORK

Network tracing. Network tracing remains a critical challenge for cloud providers. Traditional methods, including active [17, 33, 34, 43, 63, 88, 99] and passive monitoring [24, 31, 35, 41, 61, 62, 70, 80], often fall short due to limited visibility. SDN-based tools [18, 27, 60, 71, 87] have become prevalent but still struggle to identify the root causes of certain issues. INT [54] and related technologies [45, 107] have enhanced tracing granularity by utilizing virtual/programmable network devices [11, 42] to collect data at each network hop. Building on this, systems like [19, 36, 58, 96, 104–106] have been developed for network management. However, they still trace per hop without exploring internal details. In contrast, CubeTrace focuses on the internal mechanics of cloud gateways, providing richer insights beyond the capabilities of current telemetry systems.

Finally, FlowRadar [57] uses per-flow counters as inputs, making it somewhat similar to our work. But it operates on a simpler graph of routers without ambiguities and focuses on decoding aggregated counters and mitigating errors introduced by Bloom filters. Instead, our work focuses on the internal workflow of cloud gateway, addresses hardware heterogeneity, and extends to include performance analysis.

Host network stack tracing. This work resembles packet tracing on host machines. For example, the Linux network stack has undergone extensive scrutiny [21, 28, 29, 72], facilitated by a diverse range of tracing frameworks [3, 8, 12, 40]. Virtual networks and applications [51, 77, 86] have also received much attention. The rise of high-speed networking has driven the adoption of lightweight, kernel-bypass frameworks like DPDK [1] and others [48, 53, 65, 76, 78, 81] to enhance throughput and reduce latency. Corresponding tracing methodologies have been developed to analyze these frameworks [2, 23, 64, 90, 101, 102]. Unlike prior research focusing on the software architecture of a single device, CubeTrace tackles the complexities of three hardware platforms with different programming paradigms and physical constraints.

Distributed tracing framework. Dapper [84] pioneered distributed tracing frameworks, followed by systems with various focuses, including minimizing overhead [56, 79, 94], performance profiling [46, 47, 66], and failure analysis [44, 75, 92, 103]. Recently, distributed tracing systems have evolved into comprehensive observability tools by integrating logging and metrics, with OpenTelemetry [7] emerging as a widely adopted framework. However, most frameworks target application/RPC workflows [51, 83] and operate above the transport layer, so transport-level reliability events (e.g., packet loss) are unmodeled. X-Trace [38] augments in-band context with out-of-band reports to reason about losses, but per-packet fidelity is costly. In contrast, CubeTrace propagates only a trace ID as an aggregation key, exports out-of-band flow-cube statistics over a unified cube abstraction, and decodes flow traces into request-like trees to interoperate with existing tools.

8 CONCLUSION

We introduce CubeTrace, a microscopic network tracing system for heterogeneous cloud gateways. CubeTrace unifies tracing abstraction across three different hardware platforms as "cubes", and ensures tracing consistency and reliability. We also introduce a decoding scheme that integrates CubeTrace with established distributed tracing systems, reducing engineering effort and improving accessibility, and streamlining network monitoring for operators. Evaluations show CubeTrace is lightweight, adding minimal latency and resource overhead, while delivering precise flow-cube granularity insights, enhancing packet visibility and gateway reliability.

ACKNOWLEDGMENT

We sincerely thank our shepherd Akshay Narayan (Brown University) and the anonymous reviewers for their constructive feedback and guidance. We also thank Konstantin Makarychev (Northwestern University) for his insightful discussion on the tracing graph ambiguity problem. Congcong Miao is the corresponding author.

REFERENCES

[1] Data Plane Development Kit. https://www.dpdk.org/.

[2] DPDK Programmer's Guide: 9. Trace Library - Documentation. https://doc.dpdk.org/guides/prog_guide/trace_lib.html.

[3] eBPF. https://ebpf.io/.

[4] General SDK Configuration | OpenTelemetry. https://opentelemetry.io/docs/languages/sdk-configuration/general/.

[5] Intel is halting development of the networking chip it got from Barefoot Networks. https://www.bizjournals.com/sanjose/news/2023/01/26/intel-halts-development-of-tofino-switch-chips.html.

[6] Intel® Tofino™ 2. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2.html.

[7] OpenTelemetry – High-quality, ubiquitous, and portable telemetry to enable effective observability. https://opentelemetry.io/.

[8] The netfilter.org project. https://www.netfilter.org/.

[9] Trident4 / BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series.

[10] Uprobe-tracer: Uprobe-based Event Tracing. https://docs.kernel.org/trace/uprobetracer.html.

[11] In-band Network Telemetry (INT) Dataplane Specification, 2020. https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[12] sysstat - System performance tools for the Linux operating system, 2020. https://github.com/sysstat/sysstat.

[13] Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach $679 Billion in 2024, 2023. https://www.gartner.com/en/newsroom/press-releases/11-13-2023-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-679-billion-in-20240.

[14] Intel® Tofino 2 12.8 Tbps, 20 stage, 4 pipelines, 2023. ://www.intel.com/content/www/us/en/products/sku/218648/intel-tofino-2-12-8-tbps-20-stage-4-pipelines/specifications.html.

[15] NVidia BlueField-3 DPU Datasheet, 2023. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.

[16] NVIDIA Mellanox Innova-2 Flex Open Programmable SmartNIC, 2023. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/.

[17] A. Adams, P. Lapukhov, and J. H. Zeng. Netnorad: Troubleshooting networks via end-to-end probing. *Facebook White Paper*, 2016.

[18] K. Agarwal, E. Rozner, C. Dixon, and J. B. Carter. SDN traceroute: tracing SDN forwarding without changing network behavior. In *Proceedings of the third workshop on Hot topics in software defined networking, HotSDN '14, Chicago, Illinois, USA, August 22, 2014*, pages 145–150. ACM, 2014.

[19] M. Anand, R. Subrahmaniam, and R. Valiveti. POINT: an intent-driven framework for integrated packet-optical in-band network telemetry. In *2018 IEEE International Conference on Communications, ICC 2018, Kansas City, MO, USA, May 20-24, 2018*, pages 1–6. IEEE, 2018.

[20] M. T. Arashloo, P. Shirshov, R. Gandhi, G. Lu, L. Yuan, and J. Rexford. A scalable VPN gateway for multi-tenant cloud services. *Comput. Commun. Rev.*, 48(1):49–55, 2018.

[21] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*, pages 1–10. IEEE, 2007.

[22] C. Beckmann, R. Krishnamoorthy, H. Wang, A. Lam, and C. Kim. Hurdles for a dram-based match-action table. In *23rd Conference on Innovation in Clouds, Internet and Networks and Workshops, ICIN 2020, Paris, France, February 24-27, 2020*, pages 13–16. IEEE, 2020.

[23] A. Belkhiri, M. Pépin, M. Bly, and M. R. Dagenais. Performance analysis of dpdk-based applications through tracing. *J. Parallel Distributed Comput.*, 173:1–19, 2023.

[24] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, pages 267–280. ACM, 2010.

[25] J. Bermond and C. Thomassen. Cycles in digraphs- a survey. *J. Graph Theory*, 5(1):1–43, 1981.

[26] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[27] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen. Is every flow on the right track?: Inspect SDN forwarding with rulescope. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9. IEEE, 2016.

[28] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 65–77. ACM, 2021.

[29] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, and M. Zuppelli. Kernel-level tracing for detecting stegomalware and covert channels in linux environments. *Comput. Networks*, 191:108010, 2021.

[30] F. R. K. Chung, W. Goddard, and D. J. Kleitman. Even cycles in directed graphs. *SIAM J. Discret. Math.*, 7(3):474–483, 1994.

[31] B. Claise. Cisco systems netflow services export version 9. *RFC*, 3954:1–33, 2004.

[32] T. Coughlin. Impact of COVID-19 on the consumer electronics market. *IEEE Consumer Electron. Mag.*, 10(1):58–59, 2021.

[33] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. J. Luckie, R. K. P. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and kc claffy. Inferring persistent interdomain congestion. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 1–15. ACM, 2018.

[34] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. Netdiagnoser: troubleshooting network unreachabilities using end-to-end probes and routing data. In *Proceedings of the 2007 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2007, New York, NY, USA, December 10-13, 2007*, page 18. ACM, 2007.

[35] B. Eriksson, P. Barford, R. A. Bowden, N. G. Duffield, J. Sommers, and M. Roughan. Basisdetect: a model-based network event detection framework. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC 2010, Melbourne, Australia - November 1-3, 2010*, pages 451–464. ACM, 2010.

[36] C. Fang, H. Liu, M. Miao, J. Ye, L. Wang, W. Zhang, D. Kang, B. Lyv, P. Cheng, and J. Chen. Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 31–43. ACM, 2020.

[37] S. Fang, Q. Liu, and W. Wu. Hypernat: Scaling up network address translation with smartnics for clouds. In *IEEE Global Communications Conference, GLOBECOM 2021, Madrid, Spain, December 7-11, 2021*, pages 1–6. IEEE, 2021.

[38] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*. USENIX, 2007.

[39] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 281–297. USENIX Association, 2020.

[40] M. Gebai and M. R. Dagenais. Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead. *ACM Comput. Surv.*, 51(2):26:1–26:33, 2018.

[41] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 350–361. ACM, 2011.

[42] A. Gulenko, M. Wallschläger, and O. Kao. A practical implementation of in-band network telemetry in open vswitch. In *7th IEEE International Conference on Cloud Networking, CloudNet 2018, Tokyo, Japan, October 22-24, 2018*, pages 1–4. IEEE, 2018.

[43] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. A. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 139–152. ACM, 2015.

[44] T. Gupta, J. B. Leners, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 427–441. USENIX Association, 2013.

[45] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 71–85. USENIX Association, 2014.

[46] J. Huang, B. Mozafari, and T. F. Wenisch. Statistical analysis of latency through semantic profiling. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 64–79. ACM, 2017.

[47] L. Huang and T. Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 76–91. ACM, 2021.

[48] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 113–129. USENIX Association, 2017.

[49] Z. Jia and E. Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In T. Sherwood, E. D. Berger, and C. Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 152–166. ACM, 2021.

[50] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.

[51] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 34–50. ACM, 2017.

[52] N. P. Katta, O. Alipourfard, J. Rexford, and D. Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research, SOSR 2016, Santa Clara, CA, USA, March 14 - 15, 2016*, page 6. ACM, 2016.

[53] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. E. Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 24:1–24:16. ACM, 2019.

[54] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, L. J. Wobker, et al. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, pages 1–2, 2015.

[55] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. TEA: enabling state-intensive network functions on programmable switches. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 90–106. ACM, 2020.

[56] C. Lai, J. Kimball, T. Zhu, Q. Wang, and C. Pu. milliscope: A fine-grained monitoring framework for performance debugging of n-tier web services. In *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 92–102. IEEE Computer Society, 2017.

[57] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 311–324. USENIX Association, 2016.

[58] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 44–58. ACM, 2019.

[59] S. Lin, S. Jiang, L. Wang, J. Wang, C. Pei, J. Zhao, W. Wu, K. Ren, L. Zhuang, Q. Liu, H. Yu, S. Li, Y. Huang, Y. Zhu, Y. Xiao, A. Chen, L. Kong, and C. Miao. Xfir: Accelerating new-flow setup on host servers of a large cloud network. In *Proceedings of the ACM SIGCOMM 2026 Conference, ACM SIGCOMM 2026, Denver, CO, USA, August 17-21, 2026*. ACM, 2026.

[60] G. Liu, M. Trotter, Y. Ren, and T. Wood. Netalytics: Cloud-scale application performance monitoring with SDN and NFV. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, page 8. ACM, 2016.

[61] V. Liu, D. Halperin, A. Krishnamurthy, and T. E. Anderson. F10: A fault-tolerant engineered network. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 399–412. USENIX Association, 2013.

[62] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 290–301. ACM, 2011.

[63] P. Marchetta, A. Botta, E. Katz-Bassett, and A. Pescapè. Dissecting round trip time on the slow path with a single packet. In *Passive and Active Measurement - 15th International Conference, PAM 2014, Los Angeles, CA, USA, March 10-11, 2014, Proceedings*, volume 8362 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 2014.

[64] I. Marinos, R. N. M. Watson, and M. Handley. Network stack specialization for performance. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 175–186. ACM, 2014.

[65] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. D. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. E. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 399–413. ACM, 2019.

[66] H. Mi, H. Wang, H. Cai, Y. Zhou, M. R. Lyu, and Z. Chen. P-tracer: Path-based performance profiling in cloud computing systems. In *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pages 509–514. IEEE Computer Society, 2012.

[67] C. Miao, Y. Xiao, M. Canini, R. Dai, S. Zheng, J. Wang, J. Bu, A. Kuzmanovic, and Y. Wang. TENSOR: lightweight BGP non-stop routing. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 108–121. ACM, 2023.

[68] C. Miao, Z. Yao, J. Lv, J. Wang, S. Lin, X. Zhang, Y. Xiao, W. Guo, J. Bu, Y. Wang, X. Zou, Y. Jiang, M. Canini, and G. Xie. Cost-effective and reliable global internet peering with programmable switches. In *23rd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2026, Renton, WA, May 4-6, 2026*, pages 2671–2684. USENIX Association, 2026.

[69] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 15–28. ACM, 2017.

[70] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 129–143. ACM, 2016.

[71] T. Nelson, D. Yu, Y. Li, R. Fonseca, and S. Krishnamurthi. Simon: scriptable interactive monitoring for sdns. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015*, pages 19:1–19:7. ACM, 2015.

[72] H. Nemati, F. Tetreault, J. Puncher, and M. R. Dagenais. Critical path analysis through hierarchical distributed virtualized environments using host kernel tracing. *IEEE Trans. Cloud Comput.*, 10(2):774–791, 2022.

[73] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 361–378. USENIX Association, 2019.

[74] T. Pan, N. Yu, C. Jia, J. Pi, L. Xu, Y. Qiao, Z. Li, K. Liu, J. Lu, J. Lu, E. Song, J. Zhang, T. Huang, and S. Zhu. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 194–206. ACM, 2021.

[75] C. Pham, L. Wang, B. Tak, S. Baset, C. Tang, Z. T. Kalbarczyk, and R. K. Iyer. Failure diagnosis for distributed systems using targeted fault injection. *IEEE Trans. Parallel Distributed Syst.*, 28(2):503–516, 2017.

[76] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 325–341. ACM, 2017.

[77] M. A. Qureshi, J. Yan, Y. Cheng, S. H. Yeganeh, Y. Seung, N. Cardwell, W. de Bruijn, V. Jacobson, J. Kaur, D. Wetherall, and A. Vahdat. Fathom: Understanding datacenter application network performance. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 394–405. ACM, 2023.

[78] L. Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 101–112. USENIX Association, 2012.

[79] A. Satish, T. Shiou, C. Zhang, K. Elmeleegy, and W. Zwaenepoel. Scrub: online troubleshooting for large mission-critical applications. In R. Oliveira, P. Felber, and Y. C. Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 5:1–5:15. ACM, 2018.

[80] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. R. Kompella, and D. G. Andersen. csamp: A system for network-wide flow monitoring. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 233–246. USENIX Association, 2008.

[81] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. Isostack - highly efficient network processing on dedicated cores. In *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*. USENIX Association, 2010.

[82] H. Shao, X. Wang, Y. Lu, Y. Yu, S. Zheng, and Y. Zhao. Accessing cloud with disaggregated software-defined router. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 1–14. USENIX Association, 2021.

[83] J. Shen, H. Zhang, Y. Xiang, X. Shi, X. Li, Y. Shen, Z. Zhang, Y. Wu, X. Yin, J. Wang, M. Xu, Y. Li, J. Yin, J. Song, Z. Li, and R. Nie. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 420–437. ACM, 2023.

[84] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[85] J. Son, Y. Xiong, K. Tan, P. Wang, Z. Gan, and S. Moon. Protego: Cloud-scale multitenant ipsec gateway. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 473–485. USENIX Association, 2017.

[86] K. Suo, Y. Zhao, W. Chen, and J. Rao. vnettracer: Efficient and programmable packet tracing in virtualized networks. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pages 165–175. IEEE Computer Society, 2018.

[87] P. Tammana, R. Agarwal, and M. Lee. Cherrypick: tracing packet trajectory in software-defined datacenter networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015*, pages 23:1–23:7. ACM, 2015.

[88] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang. Netbouncer: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 599–614. USENIX Association, 2019.

[89] C. Thomassen. Even cycles in directed graphs. *Eur. J. Comb.*, 6(1):85–89, 1985.

[90] M. Trevisan, A. Finamore, M. Mellia, M. M. Munafò, and D. Rossi. Traffic analysis with off-the-shelf hardware: Challenges and lessons learned. *IEEE Commun. Mag.*, 55(3):163–169, 2017.

[91] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[92] J. Weng, J. H. Wang, J. Yang, and Y. Yang. Root cause analysis of anomalies of multitier services in public clouds. *IEEE/ACM Trans. Netw.*, 26(4):1646–1659, 2018.

[93] Y. Xiao, X. Luo, Y. Jiang, A. Wang, H. Chen, Z. Zhou, H. Yu, J. Cao, Y. Jiang, J. Wang, M. Xu, Y. Chen, and C. Miao. Ddos detection at the scale of one hundred tbps. In *23rd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2026, Renton, WA, May 4-6, 2026*, pages 883–898. USENIX Association, 2026.

[94] S. Yang, S. J. Park, and J. K. Ousterhout. Nanolog: A nanosecond scale logging system. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 335–350. USENIX Association, 2018.

[95] W. Yang and C. Chang. Smartgate: Accelerate cloud gateway with smartnic. In *Proceedings of the 8th International Conference on Cyber Security and Information Engineering, ICCSIE 2023, Putrajaya, Malaysia, September 22-24, 2023*, pages 8–14. ACM, 2023.

[96] D. Yu, Y. Zhu, B. Arzani, R. Fonseca, T. Zhang, K. Deng, and L. Yuan. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 207–220. USENIX Association, 2019.

[97] H. Yu, J. Wang, J. Zhao, K. Ren, G. Lin, B. Zhang, Y. Guan, X. Li, H. Yin, J. Liang, L. Wang, C. Pei, Y. Wang, X. Jin, J. Wang, and C. Miao. Fornax: A hardware-centric session management in large public cloud network. In *Proceedings of the ACM SIGCOMM 2025 Conference, SIGCOMM 2025, São Francisco Convent, Coimbra, Portugal, September 8-11, 2025*, pages 663–676. ACM, 2025.

[98] C. Zeng, L. Luo, T. Zhang, Z. Wang, L. Li, W. Han, N. Chen, L. Wan, L. Liu, Z. Ding, X. Geng, T. Feng, F. Ning, K. Chen, and C. Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 1345–1358. USENIX Association, 2022.

[99] H. Zeng, R. Mahajan, N. McKeown, G. Varghese, L. Yuan, and M. Zhang. Measuring and troubleshooting large operational multipath networks with gray box testing. *Mountain Safety Res., Seattle, WA, USA, Rep. MSR-TR-2015-55*, 2015.

[100] M. Zhang, J. Bi, K. Gao, Y. Qiao, G. Li, X. Kong, Z. Li, and H. Hu. Tripod: Towards a scalable, efficient and resilient cloud gateway. *IEEE J. Sel. Areas Commun.*, 37(3):570–585, 2019.

[101] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi. Flowmon-dpdk: Parsimonious per-flow software monitoring at line rate. In *Network Traffic Measurement and Analysis Conference, TMA 2018, Vienna, Austria, June 26-29, 2018*, pages 1–8. IEEE, 2018.

[102] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, and D. Rossi. Flowatcher-dpdk: Lightweight line-rate flow-level monitoring in software. *IEEE Trans. Netw. Serv. Manag.*, 16(3):1143–1156, 2019.

[103] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In T. Brecht and C. Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 131–146. ACM, 2019.

[104] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, and N. Zhang. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 991–1010. USENIX Association, 2021.

[105] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 76–89. ACM, 2020.

[106] Y. Zhou, Y. Zhang, M. Yu, G. Wang, D. Cao, Y. E. Sung, and S. H. Y. Wong. Evolvable network telemetry at facebook. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 961–975. USENIX Association, 2022.

[107] Y. Zhu, N. Kang, J. Cao, A. G. Greenberg, G. Lu, R. Mahajan, D. A. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 479–491. ACM, 2015.

Appendices are supporting material that has not been peer-reviewed.

A SCALABILITY ISSUE FOR PER-PACKET GRANULARITY

Scalability concerns arise when determining the granularity of data collection, which can range from per-packet to per-flow or even broader.

At the per-packet level, detailed data is collected for each packet, including which DPDK functions, programmable switch stages, or FPGA modules it traverses, and the time spent at each point. We refer to this granularity as the "packet-cube" throughout this paper. At the per-flow (or "flow-cube") level, data is aggregated for an entire flow, capturing the number of packets traversing each cube along with the minimum/average/maximum processing latencies. Coarser granularity is similar to the per-flow level but aggregates information across multiple flows, reducing specificity for broader insights.

Naturally, finer granularity enables more detailed analysis and supports a broader range of questions. For instance, packet-cube granularity can meet nearly all operational demands. Additionally, it aligns well with distributed tracing frameworks, where a "trace" in cloud gateways refers to all contexts of one single packet. In this case, "cube" is nothing different from "node" or "span". Packet-cube data provides the most compatible input for these frameworks.

However, high granularity comes at a cost: collecting, storing, and analyzing packet-cube data requires significant resources. Logging at the packet-cube level incurs substantial memory and processing overhead. For example, tracing a single flow with thousands of packets would require 1,000x more memory than flow-cube granularity. When tracing rules are expanded, such as including all packets from a single sender IP, the scale of data collection could grow exponentially, potentially overwhelming available resources.

B REMOVAL OF AMBIGUITY

When our algorithm in § 3.4 identifies a tracing graph as ambiguous, adjustments are required to clarify it, e.g., by adding or removing hook points within the program. Our general strategy for these modifications is to insert an extra vertex between any pair of adjacent vertices flagged by our algorithm. For instance, referring to Figure 4(a), introducing a new vertex between any pairs of vertices (A, C), (A, D), (B, C), or (B, D) can effectively break the loop in the derived graph G', as demonstrated by the change from Figure 4(c) to Figure 4(d). It follows that ambiguity is removed from the original tracing graph.

However, ambiguity can arise in multiple places, *i.e.,* there could be multiple loops in the derived graph G'. This is particularly common during the initial design phase of the tracing graph or after making significant updates to the tracing program. In fact, in a directed graph, the potential number of loops could scale factorially with the size of the graph, *i.e.,* the number of vertices [50]. As a

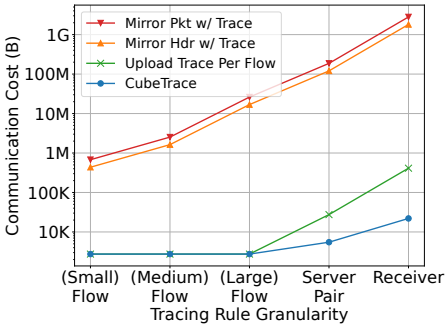


Figure 12: Communication costs for uploading traces.

result, identifying all possible loops constitutes a #P-Complete challenge, signifying a complexity on par with, or exceeding, NP-Hard problems [91].

Fortunately, our analysis indicates that the tracing graph tends to be sparse. Therefore, we utilize a tailored version of Johnson's algorithm [50] to find all the loops, achieving a time complexity of $O((|V|+|E|)\cdot(c+1))$, where c represents the total count of simple cycles in the graph. The outputs from this algorithm are used to guide adjustments to the tracing graph accordingly.

C COMMUNICATION COST EVALUATION

We proceed to evaluate the communication overhead involved in uploading trace data to the analysis backend. We compare CubeTrace with simulations of its alternative design that collects tracing data at per-packet granularity (see § 3.2). This design may be realized by the alternative implementation in § 4.1 – despite facing reliability issues – where each packet includes an extra custom header encapsulating tracing data. These packets are mirrored and forwarded to the analysis backend, without any aggregation.

We consider five different tracing scenarios: (i) a small flow, (ii) a medium flow, (iii) a large flow, (iv) all traffic between a pair of servers, and (v) all traffic towards a receiver. For the first three scenarios, our simulation is based on our data center metrics, using the 10th/50th/90th percentiles of the number of packets per flow, respectively. For the latter two scenarios, we assume typical workloads between a server pair and towards a receiver, assuming 10 and 150 active flows respectively, each flow with an average number of packets.

Figure 12 shows the communication costs in different tracing scenarios. For flow tracing, if we opt to mirror packets with embedded trace data (the red line), communication costs vary from approximately 1MB to 30MB depending on the flow size. For tracing all traffic between a pair of servers, the data upload increases to ~200MB. Tracing all traffic directed towards a receiver involves transmitting several GBs of data to the analysis backend. An optimization is to mirror only the packet headers alongside the trace data (the orange line), which can reduce communication costs by about 40%. However, it still needs to transmit over 1GB of data in the last tracing scenario. It is noteworthy that, in practice, a "busy" receiver may need to handle thousands of flows concurrently, which easily makes the costs reach tens or hundreds of GBs.

In contrast, if the tracing data is aggregated locally, it reduces communication costs by orders of magnitude. For instance, when aggregating the tracing data at the flow granularity (the green line), only a few KBs are required in the first three scenarios. The commutation costs do not exceed 1MB even in the last scenario of tracing all traffic to a receiver. CubeTrace, on top of that, realizes flexible granularity (§ 3.2), which means that the traffic is aggregated at each gateway node. Because only less than 10 cloud gateway nodes are involved in the last scenario, CubeTrace realizes further traffic reductions – using less than 20KB – compared to the approach with a fixed per-flow granularity.