Network-Boosted Applications

Satadal Sengupta

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Adviser: Professor Jennifer Rexford

May 2026

Abstract

Modern Internet applications such as video conferencing, cloud gaming, and AR/VR demand high media quality, low latency, and massive scale. Yet networks still operate largely as best-effort conduits with limited visibility into application behavior or quality of experience (QoE), leaving operators unable to react in time to short-lived congestion, stealthy attacks, or dynamic scaling needs that disproportionately affect application performance, security, or scalability. Addressing these problems requires packet-level measurement and real-time control in the network itself, where traffic can be observed and acted upon before users feel the impact. High-speed programmable switches and Smart Network Interface Cards (SmartNICs) offer a path to boosting applications at up to terabit rates, but only with careful designs that operate within stringent limits on memory and computation, while handling sophisticated, stateful protocols such as Transmission Control Protocol (TCP) for on-demand traffic (for example, web browsing and video streaming) and Real-Time Communication (RTC) for real-time interactive traffic (for example, video conferencing and online gaming).

This dissertation develops efficient, hardware-amenable techniques for real-time network monitoring and control across both traffic classes. For on-demand traffic, we present DART (Data-plane Actionable Round-trip Times), which measures TCP round-trip time (RTT) at line rate by carefully handling retransmissions, reordering, and selective acknowledgments—TCP behaviors that can otherwise distort RTT samples or exhaust limited switch memory. Building on this capability, we develop HiDe (Hijack Defense) to promptly detect and mitigate stealthy long-distance routing attacks using passively monitored RTT variation as the detection signal.

For interactive traffic, we analyze representative video-conferencing systems, including Zoom (proprietary) and MediaSoup (open-source), and derive detailed packet-level performance metrics. For Zoom, we show that despite encryption and a proprietary RTC protocol, packet headers retain enough information to recover per-media-stream quality indicators such as bitrate, frame rate, and jitter. These insights motivate Scallop, a scalable hardware-

software co-designed architecture for video-conferencing infrastructure inspired by Software-Defined Networking (SDN). In Scallop, a high-speed data plane executes high-frequency, latency-sensitive operations, while a lean software control plane handles infrequent tasks, improving performance and scalability, and reducing cost relative to conventional software solutions.

Overall, this dissertation advances scalable, application-aware network monitoring and control techniques that improve the performance, security, and scalability of the Internet's two most prevalent application classes. Our open-source research artifacts, including hardware prototypes and traffic-analysis tools, are already being used by both academia and industry for further study and development.

Acknowledgments

This dissertation emerged from a deeply meaningful and rewarding period of my life. Although I was its primary contributor, it was also shaped by a remarkable community of mentors, collaborators, friends, and family. Their influence touched not only the dissertation itself, but also my development as a researcher and as a person.

I am deeply grateful to my advisor, Jennifer Rexford, whose mentorship shaped both this dissertation and my development as a researcher in fundamental ways. She taught me how to identify research problems that are both technically rich and grounded in operational realities, how to develop those problems into a coherent research agenda, and how to communicate that agenda with clarity and conviction. Her advising combined intellectual rigor with trust: she gave me the freedom to pursue my own research instincts while consistently helping me sharpen my ideas and strengthen my arguments. Over the years, I learned from her not only how to conduct strong research, but also how to work productively, balance ambition with a full personal life, mentor students thoughtfully, support them with generosity, and enjoy the process of tackling difficult problems. Much of what I value about research, and much of what I have become as a researcher, reflects her example.

I am also grateful to the other members of my dissertation committee, Maria Apostolaki, Hyojoon "Joon" Kim, Ravi Netravali, and David "Dave" Walker, for their thoughtful feedback and engagement with this dissertation. Their questions and comments helped strengthen both the individual chapters and the broader narrative connecting them. Joon, who has been a close collaborator throughout my Ph.D., offered valuable perspective on Internet measurement and systems research and helped me learn to communicate ideas more crisply in research papers. Maria, as a collaborator on the HiDe chapter, brought invaluable perspective on security research and impressed upon me the importance of creative thinking in tackling difficult problems. Ravi, as a collaborator on both the Zoom and Scallop chapters, taught me the value of thinking deeply about problem selection, motivation, and the long-term impact of research. Dave asked thoughtful questions and provided insightful

feedback on my presentations, which helped me communicate this work more clearly. I am deeply grateful to all of them for their encouragement and intellectual support throughout this dissertation.

I have also been extraordinarily fortunate in the broader circle of collaborators who shaped this work, alongside the committee members acknowledged above. I am deeply grateful to Oliver Michel for his camaraderie, his shared enthusiasm for systems research, his joy in seeing measurement efforts come to fruition, and the friendship and mentorship he continues to bring to my life. I am similarly grateful to Sophia Yoo for her camaraderie, her shared passion for systems research, and for being both a fellow traveler through Ph.D. life and a philosophical sounding board on research and beyond. I also thank David Hay for his support, collaboration, and leadership in exploring the space of programmable network platforms. Finally, I am grateful to Emma Farkash, Daniel Jubas, Hamza Khalid, and Veronika Kitsul, with whom I had the opportunity both to collaborate and to mentor; working with them brought fresh perspectives on systems research and opened the door to new research directions.

I also want to express my gratitude to the larger circle of academic and professional mentors who shaped my dissertation experience, both directly and indirectly. I am grateful to Sergey Fedorov and Jordan Holland for their invaluable guidance and support during my internships at Netflix; from them, I learned a tremendous amount about both industry research and the development and operation of a global-scale platform. I also thank Fangfei Zhou, Eli Lindsey, Olivier Poitrey, Renata Teixeira, and T. Y. Huang for their support during those internships. I am thankful to Robert "Bob" Dondero, faculty instructor for the Princeton courses for which I served as a teaching assistant, for setting an example of what it means to be a just and thoughtful teacher, and for showing how teaching methods can evolve in response to changing circumstances, as they did during the COVID pandemic. I am similarly grateful to Marion Alberty and the entire team at the Prison Teaching Initiative, including my co-instructors Fiona Bare and Abby Mintz, whose thoughtfulness and support

made my time teaching there a life-changing experience. I also thank Ankit Singla and Debopam Bhattacherjee from my time at ETH Zürich; Sandip Chakraborty, Niloy Ganguly, and Pradipta De from my time at IIT Kharagpur; and Subrata Nandi and Sujoy Saha from my time at NIT Durgapur, all of whom helped prepare me for doctoral research through their mentorship and support.

Beyond research, this dissertation rests on the support of four families, each of which shaped my Ph.D. life in a distinct and indispensable way.

The first is my given family. I am most deeply grateful to my parents, Pradip Sengupta and Suparna Sengupta, for their unconditional love, their unwavering faith in my dreams and in my ability to achieve them, and the many sacrifices that brought us this far as a family. Their love forms the foundation of all that is good in my life. I am also profoundly grateful to my extended family, and especially to my aunt and uncle, Aparna Sengupta and Siharan Sengupta, and my cousins, Swagata Sengupta and Shilpi Gupta Nag, for their constant love, encouragement, and support.

The second is my chosen family: the friends who have remained constants in my life and continue to make it richer and more meaningful. I am especially grateful to Arunangshu Sen and Soumak Pal from my school days; Hillol Samanta, Portia Banerjee, and Saptarshi Mukherjee from my undergraduate years; and Madhumita Mallick, Sudipa Mandal, and Sankarshan Mridha from my Master's days. I want to mention Rohit Verma, Vijay Shah, and Saish Redkar in particular, both as close friends and as fellow undergraduate researchers with whom my research journey began.

The third is my academic family: the friends, mentors, labmates, colleagues, and University staff who made Princeton such an energizing place to work. Beyond those I have already thanked, I am also grateful to my labmates over the years: Mary Hogan for the energizing conversations and constant support; Robert MacDavid for bringing humor to the lab; Xiaoqi "Danny" Chen for the thoughtful conversations and guidance; Yufei Zheng for the friendship and support; John Sonchack for the steady support and technical guidance;

Henry Birge-Lee for his expert guidance on all things BGP; Anchengcheng "Ann" Zhou and Fengchen Gong for the technical and personal conversations that made me look forward to going to the lab; and Mengying "Molly" Pan, Minhao Jin, Hongyu Hè, and Constantine "Kostas" Doumanidis for their presence and support. I am also grateful to Joseph "Joe" Karam from the Office of Information Technology for his patience, competence, and constant technical support. More broadly, I thank the larger systems group, especially Neil Agarwal and Victor Ongkowijaya for their close friendship and camaraderie, and Natalie Popescu, Jennifer Lam, Grace Cimaszewski, Leon Schuermann, and Christopher Branner-Augmon for the many technical conversations and their warm friendship.

My academic family also extended well beyond the lab. I thank my roommates, especially Shanka Subhra Mondal, my roommate for many years, for their friendship and support. I am also grateful to my Princeton friends outside of work, in particular Arnab Bhattacharjee, Chandrakanta Acharyya, Ariel Bronner, Jewel Mohajan, Parth Jatakia, and Ishita Chaturvedi. I also thank my Netflix co-interns, especially Dena Markudova, Altanai Bisht, and Sagar Bharadwaj, whose support was invaluable during the internships and who remain friends today. I am grateful to the Computer Science staff, especially Nicki Mahler, for always answering my questions and solving my problems with promptness and a smile. I thank Donna Nitchun and Tarik Shahbender for being the warmest and most supportive landlords one could have hoped for. Finally, I want to especially thank my support system at Princeton—Anna Braverman, Aron Talenfeld, and Jesse Chiero—who helped ensure that my mental, physical, and nutritional well-being were cared for while I pursued my research goals.

Finally, I would like to thank the fourth family—my future family. I am profoundly grateful to my partner, Nana Ama Sarfo, for being a constant source of love and strength. I thank Ama not only for her companionship, but also for setting an example of how to combine an exceptional professional life with an active and meaningful personal life. I am also grateful to her parents, Kwasi Sarfo and Monica Sarfo, and her siblings, Akua Sarfo and

Abbie Sarfo, for welcoming me into their lives with warmth and care, and for celebrating my successes as though they were their own.

I close these acknowledgments with deep gratitude for the many people and communities who made this dissertation, and this period of my life, so meaningful. Their mentorship, friendship, generosity, and love shaped not only this work, but also the person and researcher I have become. I look ahead with gratitude and hope.

Funding Acknowledgments. The research in this dissertation was supported by the National Science Foundation awards CNS-1704077 and OAC-2018308, and DARPA contracts HR0011-20-C-0107 and HR0011-20-C-0160.

To my four families.

Contents

List of Tables

List of Figures

Bibliographic Notes

The material presented in Chapter 3 is joint work with Hyojoon Kim and Jennifer Rexford, and has been previously published and presented at the Internet Architecture Board Workshop on Quality of Experience (IAB-QoE) 2021 [143], and at the ACM SIGCOMM Conference 2022 [144]. The material presented in Chapter 4 is joint work with Hyojoon Kim, Daniel Jubas, Maria Apostolaki, and Jennifer Rexford, and has been previously published and presented at the New Ideas in Networked Systems Conference (NINeS) 2026 [142] and has appeared on arXiv [141]. The material presented in Chapter 5 is joint work with Oliver Michel, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford, and has been previously published and presented at ACM Internet Measurement Conference (IMC) 2022 [103]. The material presented in Chapter 6 is joint work with Oliver Michel, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford, and has been previously published and presented at the ACM SIGCOMM Conference 2025 [105] and has appeared on arXiv [106].

Chapter 1

Introduction

Daily, millions of Internet users around the world rely on networked applications in their personal and professional lives. These applications span traditional ones like web browsing and email, modern staples such as video streaming and conferencing, and emerging ones including AI (Artificial Intelligence) chatbots and AR/VR (Augmented Reality/Virtual Reality). Across this spectrum, users expect applications to be responsive, reliable, and secure.

These applications, in turn, depend on computer networks, which are complex webs of distributed devices that process data traffic at global scale. Today's networks are being pushed to their limits as applications become more sophisticated, traffic volumes continue to grow, expectations for quality-of-experience (QoE), security, and scalability rise, and the Internet becomes increasingly heterogeneous—both in the devices that connect (e.g., mobile) and the access networks they use (e.g., cellular). Networks therefore must evolve to meet these demands.

In this dissertation, we argue and illustrate that they can do so through two complementary approaches. First, networks should *deeply observe* application traffic by continuously extracting the attributes that matter for application performance, security, and scalability (Chapters 3 and 5). Second, networks should *intervene meaningfully* by analyzing these

attributes and acting on them to boost application performance, security, and scalability (Chapters 4 and 6).

1.1 Applications, Networks, and the Gap

Modern Internet applications span a wide range of traffic patterns and user expectations, but they increasingly share one common feature: they require networks to deliver a level of performance and security that was never guaranteed as part of their original *best-effort* design. Interactive applications such as video conferencing, cloud gaming, and AR/VR demand consistently low latency and low jitter, because even brief disruptions can lead to frozen video, garbled audio, or lag. On-demand applications such as web browsing, file transfer, and video streaming place heavy demands on throughput, while still relying on low tail latency to avoid stalls and long completion times. Across both classes, users and service providers also expect security and scalability, requiring networks to defend against threats while supporting many concurrent users and sessions at reasonable cost. In this dissertation, we use the term application *outcomes* to refer to these desired properties of performance, security, and scalability, which together capture what applications ultimately expect from the network. The demand for these outcomes generates pressure on the underlying network.

1.1.1 End-to-End Outcomes Shaped by In-Network Events

Application outcomes are end-to-end properties, but what happens inside the network strongly shapes them. For example, congestion at a router can inflate delay and jitter, turning a smooth video conference into an unusable one. Routing changes, load-balancing decisions, and failures can alter paths or availability in ways that applications experience as performance degradations or outages. Security outcomes are similarly end-to-end: whether policies are enforced and whether attacks are detected and mitigated depends on decisions made along the path, not just at the endpoints. In short, the network is an active envi-

ronment where the composition of many local, hop-by-hop decisions determines eventual application outcomes.

1.1.2 End-to-End Protocols Insufficient

End-to-end protocols help applications cope with network variability through adaptation and control. Application-layer protocols, such as DASH (Dynamic Adaptive Streaming over HTTP [152]) for on-demand applications and WebRTC (Web Real-Time Communication [17]) for interactive applications, explicitly adapt user-facing QoE metrics—for example, video resolution/bitrate, rebuffering rate, and interactivity—while transport protocols such as TCP (Transmission Control Protocol [112]) primarily adapt quality-of-service (QoS) metrics like throughput, loss, and round-trip time (RTT). These mechanisms are essential, but they cannot meet all the demands of modern applications for three main reasons.

First, end-to-end control loops run at the granularity of sessions and therefore only observe metrics from their own sessions before adapting. In contrast, an intermediate network vantage point can observe thousands of concurrent sessions sharing the same links and devices, which enables more accurate detection of shared performance and security problems. Second, the network sits closest to where many issues originate, such as congestion, failures, misconfigurations, and attacks. This proximity allows it to detect and mitigate problems early, before they cascade into QoE or security degradation. End hosts typically react only once symptoms reach the edge, when it may already be too late. Third, for security, relying entirely on endpoints is risky. Attack traffic can exploit end-host vulnerabilities, and traffic from volumetric attacks must be blocked early before it overwhelms network resources.

For these reasons, the network should complement end-to-end protocols when such support is useful for modern applications. This perspective aligns with a later interpretation of the widely known *end-to-end principle* by David Clark, a coauthor of the original paper, that allows carefully delegated functions at intermediate points while preserving endpoint control over application behavior [36, 129].

1.1.3 The Application–Network Gap

However, the end-to-end and layered structure of the Internet creates a significant gap between what applications need and what the network can readily observe and control (described in detail in Chapter 2). Applications reason in terms of user requirements, sessions, and outcomes such as QoE and security. Network devices, by contrast, operate hop by hop on individual packets, using only header fields and a small amount of flow state from preceding packets, and must make decisions under strict per-packet timing constraints at massive traffic volumes. Assuming no tight co-ordination between applications and networks, as is typically true on the wide-area Internet, by the time application data reaches the network, much of its semantics have already been abstracted away by the protocol stack. Also, key semantics are often encoded in proprietary header formats or hidden behind encryption. As a result, network devices cannot naively parse packets to recover the application-level context needed to reason directly about QoE or security.

In summary, endpoints have the richest context and can run sophisticated logic, but they have limited insight into where and why problems arise inside the network. Network devices sit on the traffic path and can observe issues as they emerge, but they are constrained in computation, memory, and semantic visibility. Put together, the entity that understands what matters is far from where problems manifest, while the entity best positioned to respond quickly has the least context. For reasons of practicality and deployability, this dissertation does not assume that applications or endpoints can be redesigned to expose their full internal state to the network, nor does it try to relocate application logic wholesale into the network. Instead, it asks what limited forms of in-network observation and intervention are possible using only the small amount of application-relevant information already available in packets, within the operational constraints of network devices. This application–network gap motivates novel mechanisms that let networks extract metrics relevant to application outcomes from packet headers and act on them to improve those outcomes.

1.2 Network-Boosted Applications

The gap between what applications need and what networks can natively provide is wide. However, modern programmable packet processors, such as high-speed programmable switches and SmartNICs, make on-path computation practical, expanding the network's role beyond best-effort forwarding. This makes it possible to view the network as an active contributor to application outcomes rather than a passive conduit for traffic. We refer to this perspective as *network-boosted applications*. Under this view, end-to-end protocols remain essential, but they no longer need to bear the full burden of performance, security, and scalability, especially when many problems originate inside the network, where programmable devices can address them swiftly.

These programmable platforms are powerful, but they are not general-purpose devices. They process packets as a stream at line rate, with only a small, bounded amount of computation, memory, and memory access available per packet (described in more detail in Section 2.2). In high-speed programmable switches, packet processing typically follows a fixed pipeline of stages, with limited branching, simple arithmetic, and no loops; even when a packet can be recirculated for another pass, doing so consumes valuable bandwidth. SmartNICs offer more flexibility, but their throughput and latency still degrade as per-packet logic becomes more complex. As a result, network-boosted designs must be resource-aware: they must keep the fast path simple, use compact state, and push richer or less frequent logic into software when needed.

In this dissertation, we structure this vision as a progression of capability levels, as illustrated in Figure 1.1. Each level builds on the previous one, moving from understanding traffic to monitoring it continuously, to taking real-time actions, and eventually to offloading parts of application functionality into the network, while also pointing toward a longer-term opportunity for application-network co-design. Prior work has addressed sub-problems at individual levels independently, but has not typically framed them together as stages in this broader progression. Later, we describe how we apply this structure to two major classes of

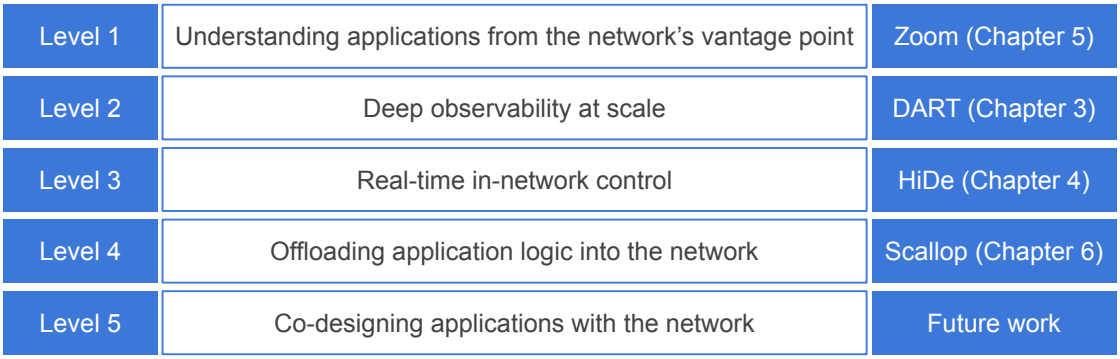


Figure 1.1: Network-boosted applications: Progression of network support for demanding applications, and our contributions in advancing those directions.

modern applications—on-demand and interactive.

Level 1: Understanding applications from the network's vantage point. Before a network can improve an application, it must be able to interpret what it sees on the wire. This is increasingly difficult because modern applications use proprietary protocols and widespread encryption—at the application layer and increasingly at the transport layer (e.g., QUIC [84])—which obscures semantics. Even so, we observe that packets often retain enough structure and metadata in their headers for the network to infer meaningful application semantics and outcome-relevant metrics passively. Prior work has shown that this kind of passive inference is possible across a range of applications and protocols [11, 21, 62, 63, 103, 108, 118, 140, 146, 150, 170].

Level 2: Deep observability at scale. Once the network can interpret application behavior, it must be able to monitor it continuously and at scale. This requires moving beyond coarse-grained traffic sampling and intermittent active probes, toward collecting metrics directly from live traffic in the data plane. A central theme in this dissertation is that *deep observability* is feasible even on resource-constrained hardware, as long as measurement is carefully designed around the device's constraints. Related work on in-network telemetry supports this direction [61, 78, 114, 144, 183, 188].

Level 3: Closing the loop with real-time in-network control. Observability is valuable, but it is not sufficient on its own. Many performance and security issues are time-sensitive, and reacting after end hosts observe symptoms can be too slow. At this level, the network analyzes what it measures and once it detects a performance or security issue, it takes action immediately, closing the control loop. Prior systems for fast in-network control illustrate parts of this step [33, 41, 153, 159, 180, 182].

Level 4: Offloading application logic into the network. At the next level, the network goes beyond monitoring and control to execute selected portions of an application's workload directly. The key is to identify operations that are repetitive, latency-sensitive, and bandwidth-intensive, and implement them where the traffic already flows. Doing so calls for hardware–software co-design, with programmable switches or SmartNICs handling the high-volume fast path while a lightweight software control plane handles infrequent, complex, or state-heavy tasks. When applied effectively, this split improves QoE and scalability, enabling support for many more users at the same cost as software-based implementations. Prior work on in-network caching, offload, and hardware-software co-design supports this direction [7, 74, 105, 147].

Level 5: Co-designing future applications with the network. Finally, looking further ahead, we have the opportunity to design applications with the network as an active partner rather than an opaque substrate, a direction that some prior work has explored [73, 82, 174, 175]. Emerging workloads such as AR/VR and real-time AI inference place stringent demands on coordination, latency, and throughput across many endpoints. Realizing network-boosted applications at scale will require abstractions and primitives that span a heterogeneous set of programmable platforms and that application developers can use when building their systems. Developers will also need safety mechanisms, especially around AI-driven control, that they can trust before deploying such systems in production. This dissertation takes initial steps in this direction by distilling design principles for building

application-aware functionality on modern programmable network devices. We leave the full realization of this level to future work.

1.3 Contributions

In this dissertation, we apply the lens of network-boosted applications to the two most prevalent application classes—on-demand and interactive. Across both, the central idea is to close the application–network gap by combining (i) *deep observability* to extract outcome-relevant signals passively from live traffic at line rate with (ii) meaningful *in-network intervention* that acts on those signals, while operating within the tight resource constraints of programmable packet processors. All our artifacts, including hardware prototypes and analysis tools, have been open sourced and are being leveraged in both academia and industry.

1.3.1 Continuous Visibility and Real-Time Security for On-Demand Traffic

On-demand applications such as web browsing, file transfers, and video streaming generate high-volume TCP traffic where performance is often stable, but security risks can be severe. These settings call for (i) always-on, passive monitoring that captures what applications experience, and (ii) fast, in-network defense that minimizes exposure of traffic to attacks. Our first two contributions show how programmable network devices can provide this combination at line rate despite tight constraints on per-flow state and per-packet computation.

Continuous, In-Network RTT Monitoring (Chapter 3) [144]. Our first contribution demonstrates for the first time that it is possible to measure RTT both accurately and continuously from passive TCP traffic inside the network at production speeds. Doing so requires overcoming a core challenge: naively matching data packets to acknowledgments is insufficient under real TCP behavior. Retransmissions, reordering, and cumulative acknowledgments can corrupt RTT samples and can also cause state to grow quickly, exhausting

limited device memory. We develop DART, a design that maintains a small amount of per-flow state to identify when an RTT sample is valid and carefully manages per-packet state so measurement remains accurate and memory-efficient across thousands of concurrent long-lived flows. Implemented on a programmable switch, DART generates 99% of the RTT samples produced by an offline baseline based on a variant of the popular tool *tcptrace* [121]. The resulting RTT measurements are indicative of real-time performance and security and enable interventions that improve application outcomes.

Real-Time Detection and Mitigation of Long-Distance Routing Attacks (Chapter 4) [142]. Our second contribution illustrates that it is practical to detect and mitigate long-distance routing hijacks using passive RTT measurements in real time, reducing exposure of traffic to the attacker to just tens to hundreds of milliseconds. Doing so requires overcoming a core challenge: while hijacks can induce an immediate increase in the propagation-delay component of RTT, RTT measurements in the wild are noisy, and the RTT surge must be distinguished from benign variation such as congestion and routine route changes. At the same time, performing sophisticated changepoint detection on RTT time series is intractable on programmable hardware. We develop HiDe, a design that aggregates passive RTT measurements at the granularity of destination IP prefixes, denoises them using minimums within time windows to approximate propagation delay, and applies a switch-friendly changepoint detector to flag suspicious minimum RTT surges as they occur. When HiDe suspects an attack, it triggers operator-configurable mitigation to contain the event immediately, while allowing optional follow-up checks to reduce false positives.

1.3.2 Application-Aware Monitoring and Scalable Infrastructure for Interactive Traffic

Interactive video conferencing is characterized by stringent latency and jitter requirements, and by scalability pressures due to ever-growing usage and meeting sizes. Network boosting

in this setting requires two capabilities. First, the network must recover enough application structure from passively observed traffic to compute QoE-relevant metrics, even when protocols are proprietary and encrypted. Second, it must offload the highest-volume work in the video-conferencing infrastructure, namely selective replication, onto high-speed packet-processing hardware. The next two contributions address these needs.

Passive, application-aware monitoring of Zoom traffic (Chapter 5) [103]. Our third contribution develops a methodology for extracting QoE-aware performance metrics for video conferencing from the network's vantage point, using only passively captured traffic and without any end-host instrumentation. Doing so requires overcoming a core challenge: modern conferencing applications use proprietary protocols, with mostly encrypted packet headers, so packet traces do not directly reveal the session structure or the metrics that matter for QoE. We address this by performing controlled experiments on Zoom traffic to identify the header fields that remain stable and informative, and using them to reconstruct application semantics. The resulting measurement pipeline can group traffic into meetings, identify and separate media streams, and compute metrics such as bitrate, frame rate, and latency and jitter directly from passively-observed packets.

Hardware–software co-designed architecture for scalable video conferencing (Chapter 6) [105]. Our fourth and final contribution presents a hardware–software redesign of the video-conferencing infrastructure, motivated by the observation that much of the work of a Scalable Forwarding Unit (SFU)—the server at the heart of video-conferencing infrastructure—is similar to standard networking tasks. We leverage this observation to build an SDN-inspired split architecture. A programmable, high-throughput data plane handles the frequent, latency-critical operations on media packets, while a software control plane handles semantically rich but lower-frequency tasks such as session management and feedback-driven rate adaptation. The design maps selective replication decisions onto hardware-friendly replication primitives, enabling the hardware fast path to carry most

packet processing while software controls policy at longer time scales, improving scalability by 7–422× and median SFU-induced latency by 27×.

Chapter 2 provides further background on the application-network gap, the opportunities that programmable networks offer and the constraints that come with them, the tailoring of network-boosting for different contexts, and the challenges of achieving the same. Chapters 3–6 present our contributions in detail. We then conclude by summarizing the key lessons from the dissertation and outlining how they can be extended to achieve broader impact in future work.

1.4 Ethical Considerations

This research study was reviewed and approved by our Institutional Review Board (IRB). All campus packet-trace data come from our university network and were anonymized at the point of collection by network engineers who are expressly authorized to handle private data. Anonymization followed the exact procedures laid down by the IRB—anonymizing all IP and MAC addresses, and stripping all payloads. Researchers never had access to any raw or deanonymized data. To validate *HiDe*'s detection and mitigation capabilities in a live Internet environment, we performed two controlled BGP hijacks using prefixes assigned to us by the PEERING testbed [132]. We temporarily announced these prefixes from our own hosts under testbed guidelines, ensuring no impact on any external networks or clients. All BGP announcements and withdrawals adhered to PEERING's guidelines, and only our own test prefixes were affected.

Chapter 2

Background and Challenges

In this chapter, we provide background on the properties of the network that cause a persistent mismatch between what applications require and what the network can provide (Section 2.1), the programmable network platforms that we leverage to realize our vision of network-boosted applications (Section 2.2), an exposition of how to tailor network boosting to different kinds of applications and workloads (Section 2.3), and the challenges that such tailored network-boosting imposes (Section 2.4).

2.1 Network: End-to-End and Up-and-Down

Meeting the demands of modern applications remains challenging for networks. To see why, it helps to view the Internet along two dimensions: *end-to-end*, from sender to receiver, and *up-and-down*, across the protocol stack from the application layer down to the physical layer.

From an end-to-end perspective, traffic traverses multiple hops, passing through a sequence of network devices, such as access points, routers, and switches. Typically, each device only has a local view of the traffic it forwards, and it makes local decisions, such as how packets are queued, scheduled, and forwarded at each hop.

From the up-and-down perspective, applications generate data using sophisticated application-layer protocols. For example, on-demand streaming services (e.g., Netflix in a

web browser) and interactive applications (e.g., Zoom as a desktop app) rely on protocols such as Dynamic Adaptive Streaming over HTTP (DASH) and Web Real-Time Communication (WebRTC). These protocols manage traffic at the granularity of an application session—for instance, between a user and a video server during video playback, or between participants in a video call. Within a session, the application encapsulates data into messages that reflect application-level units, such as audio and video frames. These messages are then broken into segments (sometimes called datagrams) carried by transport protocols like TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). Transport segments in turn become IP (Internet Protocol) packets, which is the unit that network devices forward across hops.[1] This design ensures modularity by cleanly separating concerns across layers in the protocol stack. However, it also means that the application's intent and semantics are largely obscured by the time traffic appears as packets in the network. In addition, modern applications often encrypt the packet payload as well, leaving the network to reason about traffic using only a small amount of information in packet headers.

2.2 Programmable Networks

Network devices sit on the critical path for essentially all Internet traffic, so application outcomes ultimately depend on how efficiently these devices can process packets. At a high level, network devices perform two complementary functions: the *control plane* decides how traffic should be handled, while the *data plane* executes those decisions on packets at *line rate*, meaning at the full speed of the link. Historically, network devices primarily performed this second function, i.e., packet forwarding. Today, however, a growing class of *programmable* network devices, such as high-speed programmable switches and Smart Network Interface Cards (SmartNICs), makes it possible for network operators to deploy new functionality directly in the data plane on the device itself. As a result, they no longer have to rely on

[1]This process continues down the stack: link-layer protocols (e.g., Ethernet or Wi-Fi) further encapsulate packets into *frames* which are then delivered as bits across the physical medium from one device to the next.

general-purpose servers that cannot keep up with line-rate traffic, or wait for device vendors to roll out new features. Consequently, the same devices that already forward traffic can increasingly be leveraged to improve application outcomes in line with modern demands.

2.2.1 Evolution of Network Devices

As hinted before, network devices were originally built as *fixed-function* switches, which worked well when the network's main job was to forward packets and most sophisticated logic lived at the endpoints. As application demands and traffic volumes increased, operators needed to introduce new network functionality faster than the hardware release cycles of switch vendors would allow, and with finer-grained control than vendor configuration knobs could provide. Software switches emerged to improve flexibility by running packet-processing logic on CPUs, but CPU-based processing could not keep up with modern link speeds.

The advent of *Software-Defined Networking (SDN)* [99] and *OpenFlow* [100] was an important step toward network programmability. They introduced a clean separation between a software-based, programmable control plane that computes forwarding policy and installs match-action rules in switch flow tables, and a simpler data plane that executes those rules at line rate. However, OpenFlow was designed around common switch features, which limited data-plane programmability to a narrow set of actions even as control-plane software became more flexible. Programmable devices emerged to address these limitations, aiming to combine hardware speed with substantially richer, operator-defined data-plane behavior.

2.2.2 Programmable Network Devices

Programmable packet-processing platforms are now highly prevalent in production networks. Some platforms are accessible to researchers and third-party developers through public toolchains and reference implementations (e.g., the Intel Tofino line of switches and NVIDIA BlueField-3 SmartNICs), while others remain proprietary in the sense that although

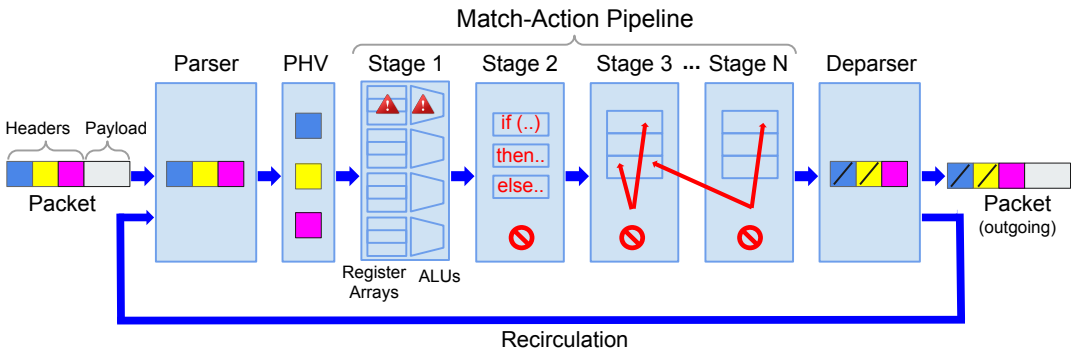


Figure 2.1: Overview of the Protocol Independent Switch Architecture (PISA) and the key features and constraints of PISA-based switches, such as the Tofino [68].

their underlying packet-processing logic is programmable, only the vendor can modify that logic directly, and network operators are restricted to a limited set of configuration options.

This ecosystem spans multiple device classes, each occupying a different point in the trade-off between performance and programmability. A major class is programmable switch ASICs (Application-Specific Integrated Circuits), which are built to process packets at terabit-scale throughput with low, predictable latency on the order of nanoseconds. Many such switches follow a pipeline-based design, commonly described by the Protocol Independent Switch Architecture (PISA), illustrated in Figure 2.1, in which each packet is processed through a fixed sequence of *stages*. This design enables line-rate packet processing at extremely high throughput, but only under tight constraints on per-packet computation and state. Because this dissertation primarily uses PISA-based switches for prototyping, we return in the next subsection to describe the PISA pipeline and its constraints in more detail.

Other switch designs take a *run-to-completion* approach, where each packet is handled by a thread executing microcode rather than flowing through a fixed pipeline. They can sustain line-rate throughput and low latency for relatively simple per-packet logic, and degrade gracefully as the per-packet processing becomes more complex.

Another major class is SmartNICs, which place packet-processing capability closer to servers and applications. SmartNICs range from System-on-Chip (SoC)- or Data Processing Unit (DPU)-based designs (e.g., NVIDIA BlueField) to Field-Programmable Gate Array

(FPGA)-based accelerators (e.g., AMD Alveo). Compared to switch ASICs, they typically offer more general-purpose compute and integration with host software, but their packet-processing throughput and latency are—similar to run-to-completion switches—sensitive to the complexity of per-packet processing.

Despite their differences, these platforms share similar constraints. To sustain production-scale line rate, they can only afford a small, bounded amount of per-packet computation, memory size, and memory access. As a result, they are powerful but resource-constrained environments, and practical designs must carefully limit per-packet work while choosing and organizing state intelligently. Throughout this dissertation, we primarily use PISA-style switches (Intel Tofino family) as a prototyping platform, and we additionally use a SmartNIC (NVIDIA BlueField-3 DPU) in Chapter 6. The broader principles, however, are not tied to any single device family and apply across programmable packet-processing platforms.

2.2.3 PISA Architecture and Constraints

Many high-speed programmable switches, including the Intel Tofino family [3, 68], follow the PISA architecture as mentioned before and illustrated in Figure 2.1. A PISA switch contains three main components: a programmable parser, a packet-processing pipeline, and a deparser. When a packet enters the switch, the parser extracts relevant information from its headers and places it into a Packet Header Vector (PHV). The PHV contains fields from the packet header, such as source and destination addresses or transport-layer port numbers, together with additional per-packet metadata. This metadata stores temporary values and intermediate results needed by the switch program as the packet moves through the pipeline. After processing is complete, the deparser uses the updated PHV to reconstruct the outgoing packet.

Between the parser and the deparser sits a feed-forward packet-processing pipeline composed of a fixed sequence of stages. As a packet traverses the pipeline, it is processed at each

stage in turn. Each stage provides a bounded set of resources that together determine the switch's packet-processing capabilities, including PHV space for carrying extracted header fields and metadata, match-action tables stored in TCAM (Ternary Content-Addressable Memory) or SRAM (Static Random-Access Memory), register arrays for persistent state, ALUs (Arithmetic Logic Units) for computation, and hash units for mapping packets to state. This staged design enables extremely high-throughput packet processing, but only under strict constraints: the number of stages is limited, memory is limited, branching (i.e., if-then-else statements) is restricted at each stage, only a bounded amount of stateful processing can be performed per stage, ALUs support only simple operations, and loops are disallowed. If processing cannot be completed in a single pass, the packet can be recirculated to the beginning of the pipeline for further processing, although excessive recirculation reduces the bandwidth available for normal traffic.

Because this dissertation primarily uses Intel Tofino switches for prototyping, it is useful to make these constraints more concrete. Tofino and Tofino2 process packets through fixed pipelines of 12 and 20 stages, respectively, while sustaining line-rate throughput of 6.4 Tbps and 12.8 Tbps [3, 68]. In each stage, actions execute in parallel, so dependent operations must be placed in different stages. Conditional branching is also limited within a stage, which translates to a limited depth of if-then-else ladders. Memory size is limited to a few megabytes, and memory access is tightly constrained: a packet in a stage can access only its own local memory, can perform only one memory access, and can read or write only one memory address. It cannot access state associated with earlier or later stages. The PHV also has fixed width, which bounds how much packet state and metadata can be carried through the pipeline. Only simple operations like addition, subtraction, and bit manipulation are allowed, but more complex arithmetic operations like multiplication and division are not possible. Tofino further provides a Packet Replication Engine (PRE) for replication and multicast, but the depth of the replication tree is limited. Tofino also

supports recirculation, with dedicated recirculation bandwidth, but once that budget is exceeded, further recirculation cuts into the bandwidth available for normal traffic.

2.2.4 Maximizing Return with Resource-Aware Design

Programmable packet-processing platforms create an attractive opportunity, but taking advantage of them requires more than simply porting functionality from general-purpose software to hardware. In practice, many programs that are straightforward in software do not map directly to these devices. Some will never fit within the available computation and memory budget, while others may fit only by consuming so many resources that little remains for other functions sharing the same hardware.

As a result, practical designs often rely on *streaming algorithms*, which process packets online as they arrive, one at a time, using only *compact state*, that is, a small amount of memory. They also often replace exact solutions with *approximate data structures* or simplified decision logic, which use limited resources to summarize information efficiently but introduce some error (e.g., by storing a 16-bit hash of a 32-bit IP address). These choices make demanding functionality feasible at line rate, but they also create a trade-off between resource consumption and approximation error. Extracting the most value from programmable packet-processing platforms therefore requires resource-aware designs that carefully balance functionality, accuracy, and hardware footprint, especially when multiple programs must share the same device.

2.3 Tailoring Network-Boosting

"Network-boosted applications" is not a one-size-fits-all solution. The network sits on the path of many applications with very different traffic patterns, constraints, and notions of success, and the same intervention can help one setting while being useless or even counterproductive in another. Effective network boosting therefore has to be tailored to the specific

problem class being targeted. In our experience, this tailoring is shaped by four key parameters: the application class, the outcome being boosted, the stakeholder who benefits, and the type of intervention.

2.3.1 Application Class (e.g., On-Demand vs. Interactive)

We focus on the Internet's two major user-facing application classes: *on-demand* applications, in which users retrieve content as needed, and *interactive* applications, in which users generate and exchange data in real time. Examples of on-demand applications include web browsing, video streaming, and file transfer, and examples of interactive applications include video conferencing, cloud gaming, and emerging AR/VR experiences. Although the Internet also carries other kinds of traffic, including background traffic from software updates and cloud backup, service-to-service communication such as transfers between a Content Delivery Network (CDN) and the application's origin server or Remote Procedure Calls (RPCs) between distributed services, and control traffic such as Domain Name System (DNS) lookups and Transport Layer Security (TLS) handshakes, the on-demand and interactive classes capture most of the applications whose performance and security affect users directly. As discussed earlier in Chapter 1, these classes place different demands on the network. On-demand applications typically prioritize throughput and completion time, whereas interactive applications require consistently low latency and jitter, since even a few milliseconds of excess delay can severely degrade interactivity.

Thanks to a long line of prior work on improving the QoE of on-demand applications—for example, adaptive-bitrate algorithms for video streaming and techniques to reduce flow completion time for web browsing—performance is relatively mature in many settings (with cellular networks as a notable exception, especially for fast-moving users). In contrast, a major challenge for on-demand traffic is *security*. For example, attackers have been known to intercept sensitive documents during downloads, or they may subject browsing traffic to techniques such as web fingerprinting. Often, these attacks take effect inside the net-

work itself, such as when traffic is silently diverted onto an attacker-controlled path by a compromised Autonomous System (AS). The network can boost on-demand applications by detecting and mitigating such attacks as quickly as possible (Chapter 4).

By contrast, a major challenge for interactive traffic is *scalability*. The rise of remote work has pushed real-time communication workloads, such as video conferencing, to unprecedented and steadily growing scale. Today, much of the burden falls on servers inside the network, which replicate each participant's media stream to all others. Networks can boost these workloads by taking on this replication task, while remaining careful about the tight latency and jitter constraints that interactive applications require (Chapter 6).

2.3.2 Outcome (e.g., Security vs. Scalability)

Network boosting must be outcome-driven, and the target outcome shapes both what signals the network should observe and how it should act.

For security, a practical approach is to implement a tight control loop of measurement, detection, and mitigation in the data plane. The network can continuously measure and analyze lightweight signals, such as RTT, that help indicate whether traffic is following expected paths or may be getting diverted via an attacker. Once an anomaly is detected, the network can respond immediately using operator-specified actions, such as dropping suspicious traffic or steering it through a more comprehensive detection step to reduce false positives. Acting early helps contain attacks and minimizes exposure of traffic to attackers, whether measured in time or bytes (Chapters 4 and 6).

For scalability, the network must first extract just enough application context from packet headers to take on part of the workload safely. In video conferencing, for example, packets often carry sufficient information, such as media frame identifiers and quality indicators, for intermediate infrastructure to make forwarding decisions without inspecting or decoding application content. Software systems already exploit this structure to decide what to replicate to each receiver based on observed receiver conditions. The network can apply the

same logic at much higher throughput by leveraging efficient replication primitives, such as multicast-style packet replication.

2.3.3 Stakeholder (e.g., Campus/Enterprise Network Operator vs. Application Provider such as Zoom)

The stakeholder who deploys and benefits from network boosting largely determines what is both feasible and useful. Different stakeholders sit at different vantage points, observe different slices of traffic, control different parts of the infrastructure, and are accountable for different outcomes. As a result, boosting for the same application class can look quite different depending on whether it is pursued by, for example, a large campus or enterprise operator versus an application or service provider.

For a network operator, the key advantage is broad visibility from a shared vantage point. Operators can observe traffic from many clients within an aggregation group (e.g., an IP address block, also called an IP *prefix*) and aggregate measurements across thousands of sessions to build a network-wide view that no single endpoint can obtain. This is especially useful for boosting the security of on-demand traffic. For example, an operator can combine RTT measurements across clients to infer whether traffic to a particular Internet service is following expected paths or is likely being diverted. When anomalies arise, the operator can take immediate, policy-driven actions such as blocking, rate-limiting, or steering traffic through stronger defenses (Chapter 4).

Network operators can also boost performance for interactive traffic. By tracking delay on the upstream path from client to Internet, they can identify media packets that have already consumed most of their latency budget (based on the ITU-recommended value for lag-free human interactivity) before leaving the local network, for example due to poor

WiFi conditions. The operator can then prioritize such packets, including selecting a better external path when multiple paths exist (e.g., in multihomed settings).[2]

By contrast, boosting scalability for interactive traffic is often limited for network operators because the dominant bottleneck typically sits in the application's own infrastructure. This is where an application provider (e.g., Zoom) has a much stronger opportunity. The provider can re-architect the system around a deliberate split between a simple, high-throughput data plane and a richer control plane. The highest-volume work, selective replication of media packets, can be pushed into programmable hardware in the data center, while a software control plane decides on longer time scales how sending rates should be adapted to each receiver's network conditions (Chapter 6).

2.3.4 Intervention (e.g., Monitoring vs. Control)

Network boosting can be achieved through two broad types of intervention. *Monitoring* provides real-time, scalable visibility into what the traffic is experiencing. *Control* builds on that visibility to decide how traffic is handled, for example by forwarding, prioritizing, replicating, rate-controlling, or blocking it. Although effective control almost always depends on monitoring, systems for monitoring and control can have distinct design requirements.

A monitoring system must provide accurate, continuous, and actionable measurements at line rate. For example, RTT is a particularly useful metric for on-demand traffic because it helps surface problems such as congestion and suspicious route changes (Chapter 3). For interactive traffic, QoE is more directly reflected in metrics such as frame rate, bitrate, and jitter (Chapter 5). Extracting these metrics from passive traffic, however, requires knowing in advance which header fields to parse and what flow state to track, all while staying within the tight memory constraints of programmable hardware.

Control requires analysis on top of the measurements, such as anomaly detection. For example, routing attacks can be detected in the data plane by tracking RTT over time and

[2]While this dissertation provides the foundations of such an approach in Chapter 5, we leave the final implementation to future work.

applying *change-detection* techniques, meaning methods that identify sudden, unexpected, and persistent shifts in a signal, to distinguish route changes from regular variation due to congestion (Chapter 4). This analysis must still fit within the compute constraints of programmable devices, which rules out many sophisticated change-detection techniques. In contrast, control aimed at scalability must support selective, application-level replication using only information in packet headers to infer application semantics, and it must map these decisions onto the available replication mechanisms, such as IP-multicast-style primitives (Chapter 6).

2.4 Challenges

Realizing the promise of tailored network boosting necessitates building systems that operate inside high-speed packet processors, while bridging a deep semantic gap between applications and networks. This, in turn, requires solving three main challenges.

2.4.1 Constraints of Programmable Network Devices

As described in Section 2.2.4, solutions implemented on programmable packet-processing devices must be carefully designed to take advantage of their high throughput and low latency in the face of their severe resource constraints. In this subsection, we discuss the design implications as they pertain to this dissertation.

Memory. Most forms of network boosting rely on keeping per-flow or per-session state, or tracking aggregate metrics that evolve over time. Yet programmable-data-plane pipelines can store only limited state, which forces designs to be selective about what they track and how long they retain it. Even seemingly simple monitoring tasks can become memory intensive once protocol complexities are accounted for. For example, continuous RTT monitoring must deal with TCP behaviors such as retransmissions, reordering, and cumulative acknowledgments, all of which can make it harder to determine which packets should be

paired together to form a valid RTT sample. Handling these cases incorrectly can distort measurements, while handling them comprehensively can exhaust limited switch memory rapidly (Chapter 3). This necessitates designs that store compact representations of state and implement judicious eviction policies for state that is no longer needed.

Compute. Similarly, many forms of network boosting require computation over packet header fields and metadata. For example, to compute RTT from TCP traffic, the system must infer when a returning acknowledgment should be paired with an earlier data packet, using only the information visible in packet headers (Chapter 3). Likewise, to detect routing attacks using RTT, it must maintain a running minimum RTT over a window of samples and track changes that may indicate an unexpected route change (Chapter 4). However, sustaining line rate leaves only a small, bounded compute budget per packet. This constraint rules out complex per-packet logic and makes even moderately sophisticated algorithms difficult to implement directly in the data plane. The tension is greatest when monitoring, analysis, and control must share the same hardware resources, alongside other functions.

Replication. As discussed earlier, boosting the scalability of interactive applications hinges on replication, since real-time systems must deliver one sender's stream to many receivers, sometimes even hundreds in webinar-style meetings. Replication is challenging on programmable devices because, even when they offer efficient multicast-style primitives, application needs are rarely to send every packet to everyone in the multicast group (e.g., meeting). Instead, packets must be replicated *selectively*, often making different forwarding decisions for different receivers based on their network conditions, using only information visible in packet headers. Achieving this requires inferring the appropriate action from header fields and control-plane inputs, and mapping those decisions onto the device's replication primitives so that selective replication remains efficient at line rate (Chapter 6).

Control–Data Plane Split. Because the programmable data plane is highly constrained, and because not every task benefits from running on every packet at line rate, it is rarely

effective to place all network-boosting functionality in the data plane. The data plane is best suited for simple operations that must be applied at very high volume, such as continuously tracking RTT for an on-demand flow or performing large-scale media replication for an interactive flow. The control plane, in contrast, can run richer logic that depends on broader context or more complex inputs, such as making rate-adaptation decisions from detailed feedback in video conferencing. However, finding the right split is nontrivial. Pushing too much into the data plane may require simplifying the logic so much that the result becomes less effective, while pushing too much into the control plane can make software processing a bottleneck and add delay or jitter. Effective designs therefore keep simple, high-volume packet processing in the data plane and reserve more complex or lower-frequency tasks for the control plane (Chapter 6). Another common pattern is a layered design, in which the data plane quickly flags suspicious events using lightweight change detection, and the control plane then performs deeper analysis before confirming the event or taking more disruptive action (Chapter 4).

2.4.2 Application and Protocol Complexity

As we have discussed in Section 1.1, application behavior is expressed at the level of sessions and messages, but the network sees packets, after the protocol stack has abstracted away much of the application's semantics. Bridging this gap requires reconstructing higher-level structure from information embedded in packet headers. For example, in the case of on-demand traffic, continuous RTT monitoring from passive TCP traffic must contend with higher-level protocol behaviors such as retransmissions, reordering, and selective acknowledgments, which can distort measurements or result in state explosion if not accounted for efficiently. Identifying such protocol behaviors requires reconstructing a *valid* range of TCP sequence numbers per flow to understand when it is safe to collect RTT samples (Chapter 3). Similarly, in case of interactive video-conferencing traffic, whether a media packet should be replicated to a receiver, depends on the identity of the sender, the media type (au-

dio/video/screenshare), and the rate adaptation status of the media stream—higher-layer semantics that must be reconstructed from packet headers (Chapter 6).

2.4.3 Encrypted and Proprietary Packet Headers

Visibility into network traffic is increasingly limited by protocol opacity. Modern applications often use proprietary header formats (e.g., Zoom) and encryption, including at the transport layer (e.g., QUIC), which reduces how much the network can infer from packet headers alone. At the same time, there are strong incentives to retain at least some useful information about traffic in packet headers. Protocol designers and application providers often leave certain information (like the spin bit in QUIC for RTT computation) in the headers to support operational needs like network monitoring, or to enable intermediate infrastructure to perform functions like selective replication for video conferencing traffic without the overhead of decrypting payloads (Chapter 5). In addition, adoption of encrypted protocols is far from universal. Transport protocols like QUIC are not universally supported, and many connections fall back to TCP when QUIC is unsupported. Operators may also intentionally restrict QUIC in some environments to preserve visibility and the ability to perform traffic engineering. Taken together, these realities suggest that the practice of embedding useful semantic information in packet headers is unlikely to disappear any time soon.

Chapter 3

Continuous In-Network Round-Trip Time Monitoring

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

the RTT samples of an offline, software baseline—a variant of the popular *tcptrace* tool that has access to unlimited memory.

3.1 Introduction

RTT is a key indicator of network performance, user Quality of Experience (QoE), and routing-protocol attacks [67, 156]. RTT relates directly to Transmission Control Protocol (TCP) throughput and also heavily influences higher-level metrics such as video QoE and page load time [13, 21]. Changes in RTTs can also be symptoms of malicious activities like traffic interception attacks [67]. In the following examples, RTT monitoring can inform important network adaptation decisions:

- RTT can be a good indicator of network congestion. When network performance starts to decline, and multiple paths are available, the network can reroute traffic to an alternate, less-congested path. This applies to routing in the wide-area network [4, 87] as well as within data centers [85].

- RTT monitoring is useful for latency-sensitive applications. For example, multiplayer cloud-gaming applications need to select the best game server for players spread across different geographical regions [30, 46]. The network can monitor the propagation delay (minimum RTT over time) en route to each potential server, and select the best one for each gaming session.

- RTT can help in inferring the QoE of on-demand video applications. For instance, an RTT hike can cause an increase in video startup delay and a decrease in video resolution [21], thus prompting the service provider to select an alternate server or path.

- RTT monitoring can help reveal routing attacks. Nation-state actors can eavesdrop on traffic by launching Border Gateway Protocol (BGP) interception attacks [8]. These at-

tacks cause a sudden and substantial increase in RTT. The network could detect and immediately stop sensitive traffic on such occasions.

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

Many popular measurement tools use *active* probes such as ICMP pings to estimate the RTT to remote hosts (e.g., iperf3 [48] and RIPE Atlas [115]). However, probe-based RTT estimates do not capture application-specific RTTs. Active probes also introduce extra traffic load and may be blocked by the remote host or network. Instead, measuring RTTs *passively* by observing the actual user traffic provides a more accurate estimate [72]. Passive RTT monitoring at the end-host requires special software (e.g., a native mobile app) or permissions. Instead, monitoring at a vantage point en route to many end-hosts (e.g., near the gateway router) enables easy aggregation of network-wide RTTs. Passive RTT monitoring for TCP involves matching packets with their corresponding acknowledgments.[1] RTT measurements of TCP traffic can also be used to infer RTTs for UDP traffic (e.g., QUIC and RTP-based video conferencing) between the same end-points or IP prefixes [6, 109].

One prevalent passive-measurement technique estimates a flow's RTT based only on the TCP three-way handshake [47]. This approach can be inaccurate for long flows (e.g., video streaming), since RTTs can vary significantly over minutes let alone hours. Also, handshake RTTs tend to be smaller than a connection's average RTT [47]. Therefore, we must monitor RTTs continuously, beyond the initial handshake.

Computing RTTs continuously is hard. The idiosyncrasies of the TCP protocol—including retransmissions and reordering—can make some RTT samples inaccurate (Section 3.2). Software tools such as *tcptrace* and *pping* ensure correctness by maintaining

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

expensive flow state and performing complex computations [116, 121]. Unfortunately, RTT monitoring in software is computationally expensive and therefore too slow for networks with high traffic volumes. For example, DPDK-based targets can process at most a few million packets per second [1]. Fortunately, the advent of commodity programmable switches (e.g., the Intel Tofino [3]) and the P4 language [20] opens up the possibility of monitoring RTTs and making adaptation decisions directly in the data plane [143].

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

In this chapter, we present *DART* (Data-plane Actionable Round-trip Times), a system that monitors on-path RTTs in real time. Our key insights are that we: (1) Strategically limit tracking of packets to only those that can lead to useful RTT samples, and (2) Identify the synergy between per-flow and per-packet state for efficient memory utilization (Section 3.3). We implement *DART* in the P4 language on the Intel Tofino switch (Section 3.4). We evaluate *DART* on traffic from our campus network and show how our system can detect a traffic-interception attack within only 63 packet exchanges (Section 3.5). We also implement a faithful Python simulator and report *DART*'s performance under different configurations (Section 3.6). *DART* is able to detect 98% of RTT samples compared to a variant of a software-based baseline *tcptrace* [121].

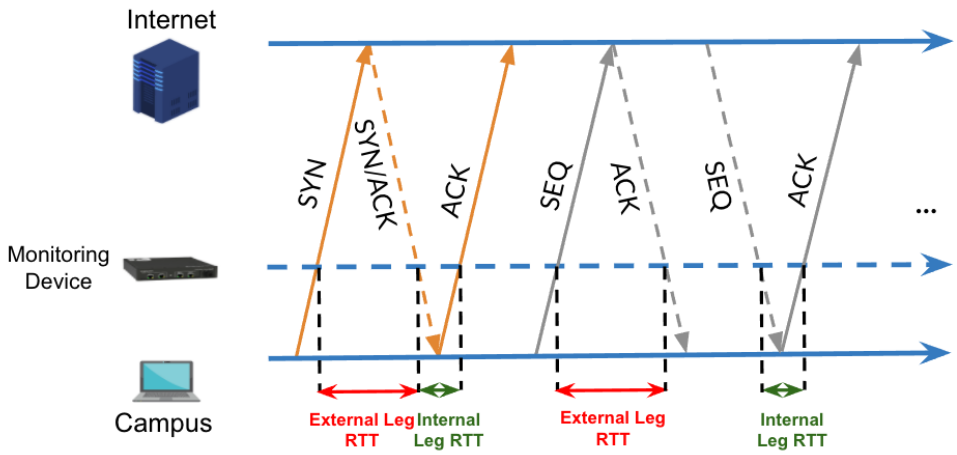


Figure 3.1: Continuous RTT measurement at a monitoring device by matching TCP data (SEQ) packets with corresponding acknowledgement (ACK) packets.

3.2 RTT Measurement Challenges

In this section, we describe a simple strawman design [32] for continuous RTT measurement of TCP flows in the data plane (Section 3.2.1). We then discuss how the many idiosyncrasies of TCP raise correctness (Section 3.2.2) and efficiency (Section 3.2.3) challenges that we address in Section 3.3.

3.2.1 Strawman for Measuring RTT

TCP carries a bidirectional data stream between two end-hosts; bytes in one direction are acknowledged in the other direction by appropriately setting sequence and acknowledgment numbers in the TCP header. When placed strategically, a monitoring device can leverage its location to continuously monitor RTTs by matching data and ACK packets. The RTT measurements include end-host delays (e.g., processing time and delayed ACKs) in addition to network delays. We briefly discuss eliminating end-host delays in Section 3.7.

Seeing both directions of the traffic. To match data packets with corresponding ACKs, monitoring needs to run on a device that can "see" both sides of the traffic. As illustrated in Figure 3.1, we denote the direction of the TCP data segment as the *SEQ* (sequence) direc-

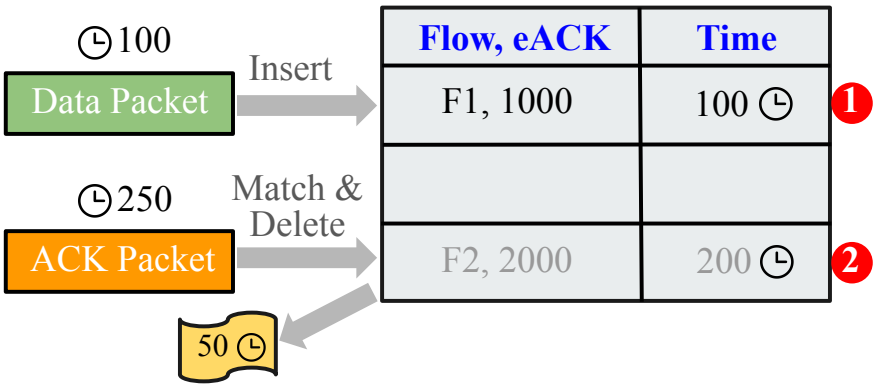


Figure 3.2: Strawman design: A hash table with flow ID + expected ACK as key and timestamp as value. Arrival of a data (SEQ) packet causes insertion into the hash table, whereas arrival of an ACK triggers deletion of the matching SEQ entry and collection of an RTT sample.

tion, and the direction of the acknowledgment segment as the *ACK* direction. Interestingly, different portions of the end-to-end path can be measured separately depending on the direction of each segment. For example, the RTT computed for a SEQ/ACK pair between the monitoring device and the Internet constitutes the *external leg* of the RTT, whereas that between the monitoring device and the client within the campus constitutes the *internal leg*. The external leg RTT is representative of the wide-area latency to the application server, whereas the internal leg RTT reveals the latency introduced by the campus infrastructure. A combination of consecutive external leg and internal leg RTTs provides the end-to-end application-level RTT for the client.

Matching data packets with ACKs. Continuous RTT measurement requires a data structure for storing SEQ information until the corresponding ACK arrives [32], as shown in Figure 3.2. The table is indexed by the SEQ packet's unique identifier—the flow identifier (4-tuple of client and server IP addresses and TCP port numbers) and a unique packet identifier within the flow (the expected ACK number or eACK). An entry is created when a SEQ packet arrives, as shown by the entry labelled 1 (white literal within a red circle) in the figure. When a matching ACK packet arrives, we look up the SEQ entry using the key (see entry labelled 2), with the source and destination fields of the 4-tuple reversed. We subtract

the entry's SEQ timestamp (200) from the new packet's ACK timestamp (250) to compute the RTT sample (50).

3.2.2 Challenges with Correctness

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

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

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

3.2.3 Challenges with Memory Efficiency

Data planes have limited memory, typically on the order of a few tens of megabytes [79]. Quirks in the operation of TCP can lead to inefficient use of this limited memory under the

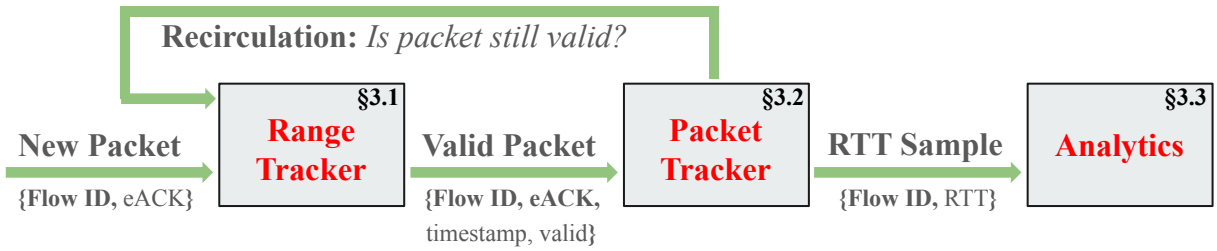


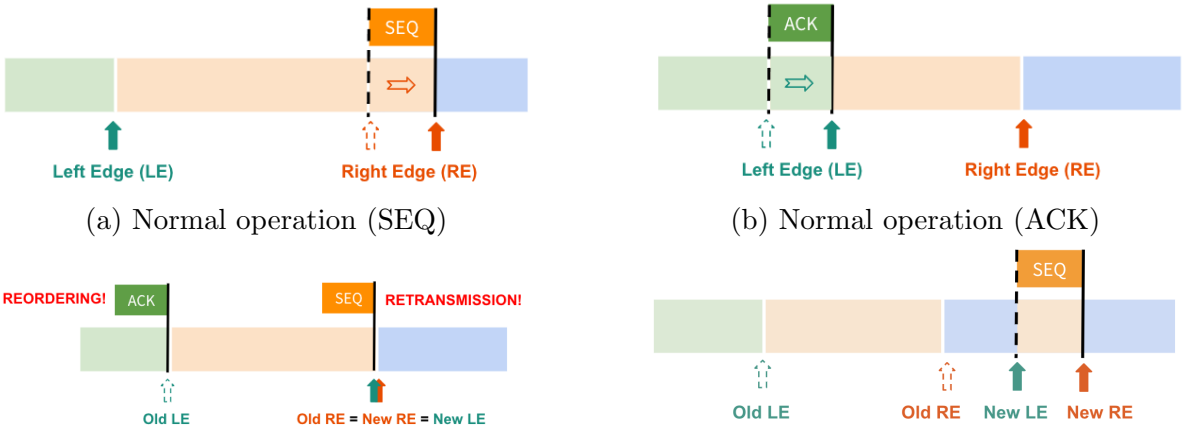
Figure 3.3: System architecture: New packets are checked at the Range Tracker table for validity. Valid packets update the RT measurement range and then await an ACK in the Packet Tracker table. Generated RTT samples are sent to the Analytics module.

strawman design.

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

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

Handling these two scenarios can be difficult, and expensive. One approach is to apply a timeout to evict SEQ entries that have not yet matched an ACK [32]. However, a small timeout would cause bias against long RTTs, and a large timeout would waste memory on storing SEQ entries that never receive an ACK. Another option is to allow new SEQ packets to evict old SEQ entries as needed. However, this approach also leads to bias against large RTTs. Instead, we need more sophisticated strategies that avoid wasting memory on SEQ entries that cannot lead to valid RTT samples.



(a) Normal operation (SEQ)

(b) Normal operation (ACK)

(c) Retransmission or reordering

(d) Hole in the sequence space

Figure 3.4: Adjustments to the flow measurement range upon arrival of new SEQ or ACK packets. Green space indicates bytes already covered by the left edge, Orange space indicates contiguous bytes in flight that can potentially lead to valid RTT samples, Blue space indicates sequence numbers not seen by the system.

3.3 *DART* System Design

To measure RTTs correctly and efficiently, *DART* must avoid tracking packets that cannot produce valid RTT samples. The challenge is to make the right decision as each packet streams through the data plane. Figure 3.3 illustrates our architecture. To avoid storing SEQ entries that could cause ambiguous RTT samples, we track the valid range of sequence numbers for each active flow (Section 3.3.1). However, we do not always know in advance that a SEQ entry will never match a future ACK. Instead, the packet tracker applies a lazy eviction strategy to reevaluate an old SEQ entry against its flow's up-to-date range of valid sequence numbers (Section 3.3.2). Finally, the analytics module aggregates the RTT samples (e.g., to compute the minimum RTT per IP prefix). The analytics module also helps optimize memory by purging SEQ entries that cannot produce a *useful* RTT sample—i.e., a sample that affects the analysis results (Section 3.3.3).

3.3.1 Tracking Valid Measurement Ranges

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

Normal operation without ambiguities. Under normal circumstances, the SEQ packets appear in order, causing the right edge of the measurement range to move forward, as shown in Figure 3.4a. Similarly, ACK packets typically arrive in increasing order, causing the left edge of the measurement range to move forward, as shown in Figure 3.4b.

Operation with ambiguities. Figure 3.4c illustrates the operation under TCP ambiguities:

- When a SEQ packet arrives with the expected ACK (eACK) *smaller* than the right edge of the measurement range, we infer a packet retransmission event. For a retransmitted packet, when an ACK arrives in the future, it is ambiguous whether the ACK acknowledges the old or the new copy of the SEQ. Furthermore, if selective acknowledgment (SACK) is not enabled, a retransmission might indicate that the receiver has been waiting on some intermediate SEQ packet(s) before sending out an ACK for a packet it had already received, thus artificially inflating the RTT.

- If an ACK packet arrives for the left edge, we conclude it is a duplicate ACK. We infer a reordering event, since duplicate ACKs are explicit markers of lost or reordered SEQ packets. Similar to retransmission, in this case, too, ACKs have been held up at the receiver thus inflating RTTs.

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

Processing ACKs for untracked SEQ packets. We might see an ACK packet with ACK number either (1) lesser than the left edge or (2) greater than the right edge. Type (1) indicates an ACK to a SEQ packet we have already deemed ambiguous and are not tracking anymore. Our system ignores such ACKs. Type (2) indicates an *optimistic ACK*—a form of early ACKing some receivers use to trick the sender into sending data more quickly [130]. *DART* ignores these ACKs too, meaning it is not misled into collecting artificially deflated RTTs (see Section 6.9).

Maintaining a single measurement range. Consider a scenario where our system encounters SEQ packets with one or more packets missing in between; e.g., the sender sends P_1 through P_4 but the system only sees P_1, P_2, and P_4, either due to packet reordering or drop. Note that these packets pass the check illustrated in Figure 3.4a since they are over the right edge and in sequence. Now, if P_3 is dropped, the system will know only upon detecting a retransmission. If reordered, P_3 arrives after P_4, thus filling in the "hole". The most optimistic approach in this scenario is to assume reordering and to store the states for both byte-ranges, i.e., P_1-P_2, and P_3-P_4. As shown in Figure 3.4d, this requires storing two measurement windows instead of one. More generally, there could be n such holes, requiring $n + 1$ measurement windows. We reason that such a strategy is too expensive in the data

plane, and the system should use a constant space for this purpose. We, therefore, store the measurement range for only the highest byte-range ahead of any hole. In the current example, the measurement window would point to only P_4's starting and ending byte numbers.

Robust against congestion and SYN attacks. Maintaining a measurement range per flow with the Range Tracker (RT) table helps significantly with memory efficiency. The mechanism becomes even more crucial when network congestion happens, where more reordering and retransmission events occur. *DART* is robust to such situations, making sure the space in both the RT and PT tables is only used for producing valid, unambiguous samples. The measurement ranges collapse more often, allowing the entries with closed measurement ranges to be deleted or overwritten. This prevents the RT table and PT space from exploding. However, this also means that *DART* may collect fewer RTT samples for some flows during heavy congestion. In order to mitigate this, *DART* can be adjusted to report the frequency of measurement range collapses for a flow as an indicator of congestion. *DART* can also be adjusted to aggregate RTT samples for flows going to the same subnets (e.g., /24 prefixes) before analyzing them. The aggregated RTTs will provide a more complete view of the congestion status of the target subnet.

DART is also robust against harsh environments, where SYN flooding attacks are common. *DART* does not create an entry in the RT table or entries in the PT table until *after* a TCP connection is established, i.e., after the three-way handshake is complete. That is, *DART* completely ignores SYN and SYN-ACK packets. Therefore, the memory usage does not explode in either the RT or PT tables during SYN flooding attacks either. We discuss *DART*'s performance under other kinds of attacks in Section 6.9.

3.3.2 Lazy Eviction with a Second Chance

The Range Tracker (RT) table helps with memory efficiency, but the Packet Tracker (PT) table can still end up storing a SEQ packet that never matches a subsequent ACK. For example, the receiver might just cut off the TCP session, never sending an ACK. Sometimes,

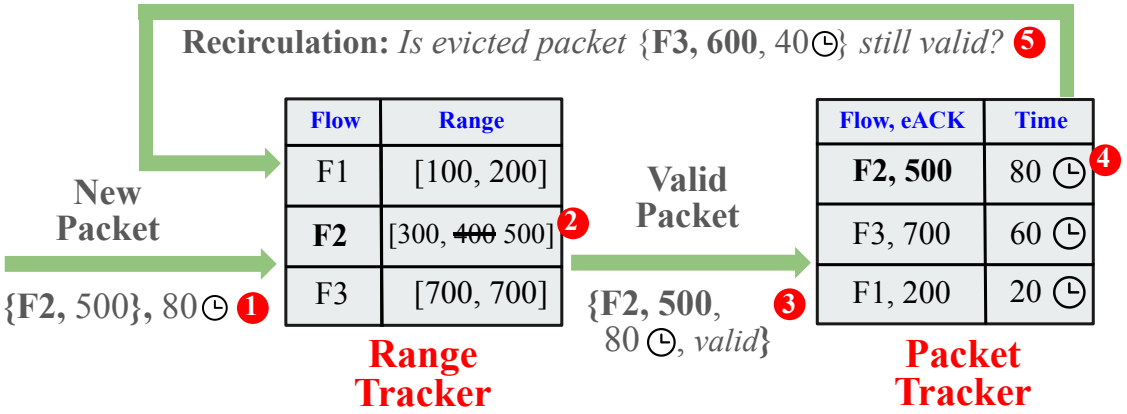


Figure 3.5: Working of *DART*: The system ensures correctness by consulting and updating the Range Tracker (RT) table before inserting packet records into the Packet Tracker (PT) table. Memory efficiency is improved by having packet records consult the RT before re-insertion, a strategy that enables *lazy eviction*.

cumulative ACKs render certain packet entries unmatched since a higher byte number has already been ACKed by the receiver. These unmatched entries occupy expensive space in the PT table; we would rather reclaim this space for tracking new SEQ packets. The goal is clear: when a new packet comes and hash collision occurs, we want to keep an old entry if it is not stale, but otherwise replace it with the new packet. What is the most accurate and efficient way to achieve this?

Timeout is biased, garbage collection is expensive. One obvious way to deal with this issue is to associate a timeout with each entry [32] in the PT table. However, packets with naturally long RTTs can suffer from undersampling under this arrangement, on top of the challenge for finding the right timeout value. A more accurate alternative is to actively scan the PT table for stale entries when an ACK arrives. For example, there can be an active garbage collector that periodically polls the RT table and remove stale entries from the PT. This strategy, however, is far too expensive, especially if implemented in the data plane.

Lazy eviction. The goal is to create an efficient mechanism that replaces entries that are indeed stale and do this without bias. Ideally, this should also happen without additional

control-plane interaction, as that would introduce extra overhead. Our first observation is that an entry does not need to be evicted until there is actually a hash collision with a new entry. We employ this *lazy eviction* strategy: this gives an entry, no matter how old it is, a chance to produce a valid sample as long as a new entry does not cause a hash collision.

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

This process is illustrated in Figure 3.5. The new packet for flow F2 arrives with expected ACK no. 500 at time t=80 (event 1, marked by the white literal in a red circle). It causes the measurement range to expand from [300, 400] to [300, 500] since it is valid (event 2). The corresponding packet record with key (F2, 500) and timestamp value 80 arrives at the PT table and collides with an existing entry (F3, 600, t=40) (event 3). The new entry gets stored (event 4) while the old entry is recirculated to the RT table for re-validation (event 5). The highlight of this mechanism is that it provides *another chance* to the packet that was evicted from the PT table instead of just discarding it right away. A

TCP flow with naturally long RTTs, for example, will be preserved by such a mechanism. While recirculations are useful in this case, they add overhead since packet recirculation bandwidth is limited in the data plane. Later in the chapter, we discuss a method to reduce recirculations by maintaining a *small cache of heavy flows* after the RT (Section 3.7).

Preventing infinite eviction loops. One might worry about an eviction loop: the old entry that was kicked out recirculates and gets re-inserted to the same index again, which kicks on the new entry, causing the old entry to get evicted again in the next iteration, and so on. To address this, we implement a method of detecting a "cycle", i.e., when the same entry that got inserted once ends up getting evicted and tries to re-insert itself. The method is to always compare the new entry with the currently evicted entry before trying re-insertion. In *DART*, we do this cycle detection before any recirculation. As another safeguard mechanism, we also set a limit on the number of recirculations per SEQ packet.

3.3.3 Tracking Only *Useful* Samples

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

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

RTT value in a time window. If the minimum RTT value significantly increases from one time window to the next one, it is worth notifying the operator through an alert.

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

3.4 Hardware Switch Prototype

In this section, we present our implementation of *DART* on the Intel Tofino programmable switches. One version of the prototype runs in the ingress pipeline of a Tofino2 switch, and the other spans the ingress and egress pipelines of a Tofino1 switch. While the Tofino2 version is more efficient and leaves the egress free for network operators to deploy their own analytics and adaptation mechanisms, the Tofino1 version enables us to deploy *DART* in our campus testbed. Our code is open-source.[2] The resource usage of the two prototypes is summarized in Table 3.1. We discuss our hardware implementation challenges and prototype features in this section, and then describe our experience deploying our prototype in the wild in Section 3.5.

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

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

Table 3.1: Data plane resource usage in the Tofino (1 and 2)

Accessing memory sequentially. Our implementation requires that actions on the values of RT and PT table records happen sequentially. For example, a RT record is updated only when the flow signature matches with that of an incoming packet; thereafter, we first set the right edge to the maximum of the existing right edge and the incoming eACK, and then compare the eACK and sequence number with the existing right edge to decide the value of the left edge (Figure 3.4). Since memory once accessed cannot be revisited without recirculation in the data plane, we spread the RT and PT tables each across 3 component tables, and therefore 3 stages.

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

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

the aforementioned header parameters.

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

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

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

Monitoring the external and internal legs simultaneously. We describe in Section 3.2.1 how it is possible to compute RTTs on both the *external leg* (i.e., monitor to the Internet) and the *internal leg* (i.e., campus client to monitor). However, monitoring both legs simultaneously in the data plane is a challenge since the same packet has to be processed both as a SEQ and an ACK packet. We are able to do this by first processing the packet as a SEQ packet, but then recirculating it with a custom header that *remembers* the relevant ACK headers (i.e., 4-tuple and ACK number).

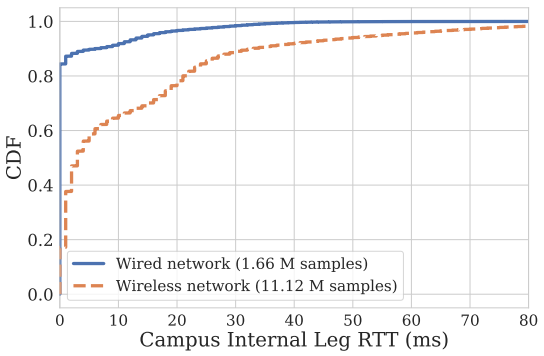


Figure 3.6: Difference in distribution of internal leg RTTs between wired and wireless subnets in the Princeton campus.

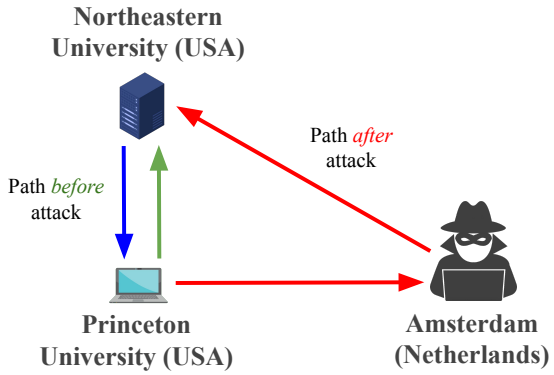


Figure 3.7: Traffic between Princeton (USA) and Northeastern (USA) intercepted by an attacker in Amsterdam (Netherlands).

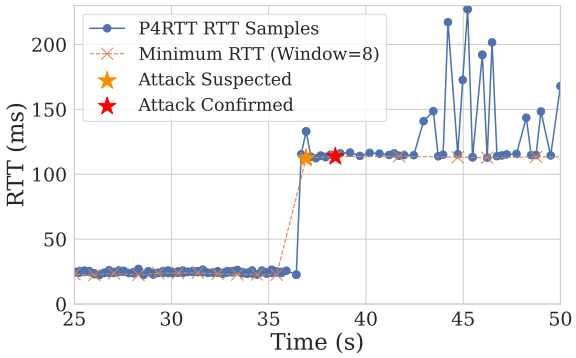


Figure 3.8: Interception attack is detected within 63 packets by observing change in minimum RTT over windows of samples.

3.5 *DART* in the Wild

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

3.5.1 RTT Monitoring on Campus Traffic

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

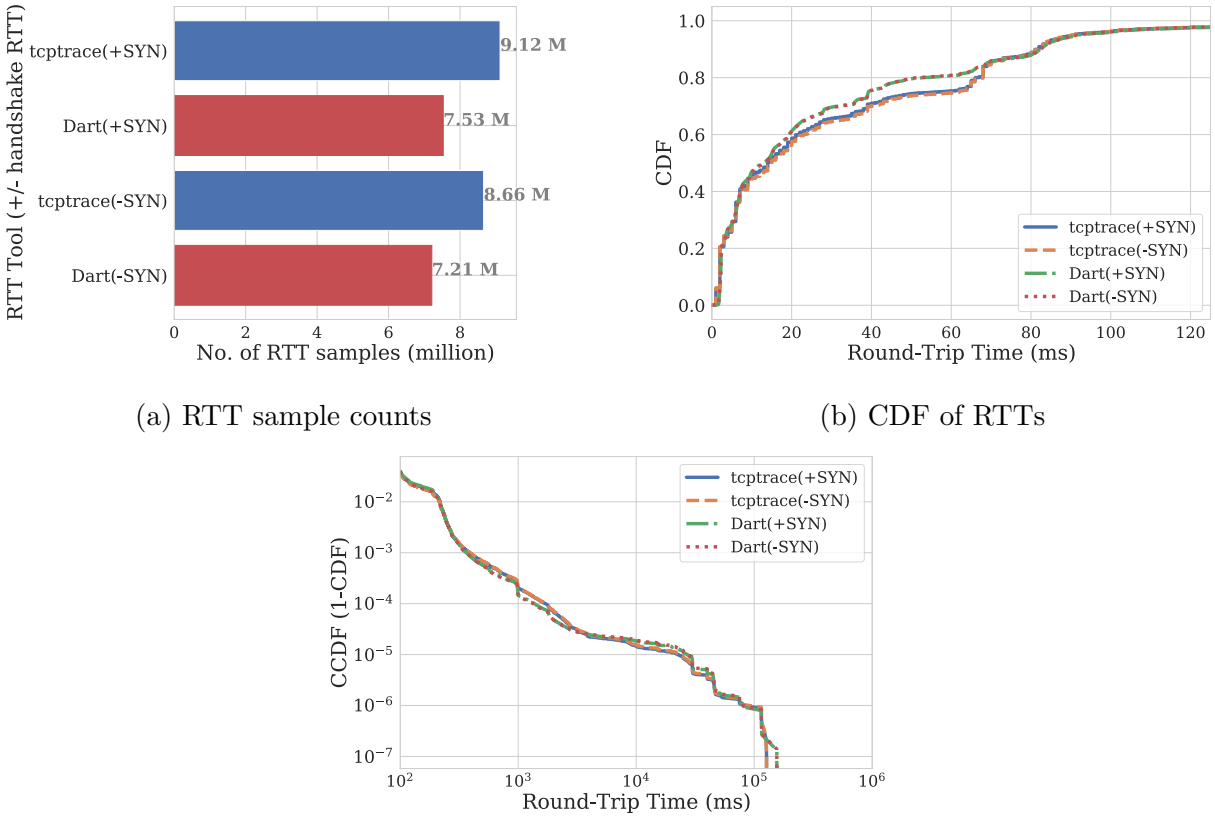
from a server connected to the switch. The RTT reports are captured at another server using `tcpdump`.

Wired network vs. Wireless network RTT. For this experiment, we replay the trace and monitor the RTTs of the *internal leg* (campus host to switch) for two campus subnets—one *wireless* and another *wired*. *DART* collects 11.12M RTT samples for wireless traffic and 1.66M for wired traffic, as most on-campus users connect via mobile devices. Figure 3.6 shows the difference in the distribution of RTTs across the two subnets. RTTs for wireless connections are uniformly larger than those for the wired connections. More than 80% of internal RTTs for the wired subnet are less than 1 ms, whereas less than 40% of RTTs for the wireless subnet are so; in fact, for wireless traffic, the internal RTT exceeds 20 ms for more than 20% of the samples. Often, the *internal* latency for wireless users rivals the *wide-area* latency. For example, for wireless clients accessing YouTube, the 90th percentile end-to-end RTT is 14 ms, including 8 ms of intra-campus delay. In contrast, the wired clients have a 90th percentile RTT of 9 ms with just 2 ms of intra-campus delay.

3.5.2 Interception Attack Detection

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

Attack setup. First, we launch an ethical traffic-interception attack using the PEERING testbed [132] and following the technique described by Birge-Lee et al. [16]. PEERING runs geographically distributed ASes to enable researchers to make real BGP announcements for controlled and isolated traffic. We control one PEERING site each at Northeastern University and Princeton University in the USA via co-located Amazon AWS server instances (Figure 3.7). We announce our PEERING-provided /23 prefix from Northeastern, and establish a TCP connection from Princeton with an IP address from the announced prefix. We then use a site at Amsterdam (adversary) to announce a more specific /24 prefix with



(a) RTT sample counts

(b) CDF of RTTs

(c) CCDF of large RTTs

Figure 3.9: *tcptrace* vs. *DART* with infinite memory: *DART* collects >82% RTTs as *tcptrace* and matches its RTT distribution closely.

a number of carefully selected BGP community attributes, such that traffic to the IP at Northeastern (victim) is now rerouted through it. We capture the traffic trace of this attack at Princeton. Our threat model assumes that *DART* is close to one of the end-hosts and can *see* both sides of the traffic, both pre-attack and post-attack. Therefore, our detection works irrespective of the direction of traffic (data or ACK) intercepted by the attacker.

Attack detection. Second, we deploy our *DART* Tofino1 prototype on a real hardware switch in our campus testbed. We replay the trace of this interception attack through this switch. *DART* collects raw RTT samples and sends them to a collection server, where a simple, threshold-based change-detection algorithm runs. The detection algorithm monitors propagation delay by computing the minimum RTT in a window of 8 consecutive raw RTT

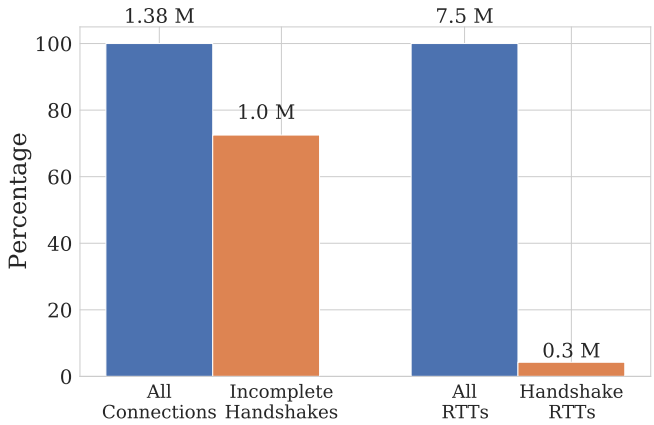


Figure 3.10: Skipping handshake packets: We can save memory for 72.5% of the connections while missing only 4.2% RTTs.

samples. An attack is *suspected* when the minimum RTT rises abruptly between consecutive windows, but *confirmed* only when the change sustains for another window. Figure 3.8 illustrates this mechanism: the attack takes effect at t=∼36 seconds, as can be observed from the abrupt rise in collected raw RTT from ∼25 ms to ∼120 ms (blue line). Based on the minimum RTTs (orange line), an attack is *suspected* almost immediately (orange star) and *confirmed* in the next window (red star). Only 63 packets are exchanged in 2.58 seconds between the interception attack taking effect and *DART* confirming the attack.

3.6 Evaluation

In this section, we evaluate *DART* using a faithful simulator written in Python. We replay a campus trace collected on April 7, 2020, for this evaluation. The trace was captured using a packet broker service near the campus gateway router. The trace contains 1.38M TCP connections and 135.78M TCP packets over a 15-minute duration. As mentioned in Section 3.5, *DART* can measure RTTs for either the *external leg* (monitor to Internet) or the internal leg (campus host to monitor), or both. In the following experiments, we limit our RTT measurement to the external leg only.

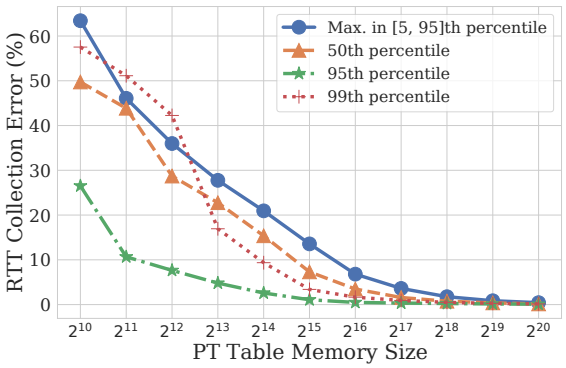
3.6.1 *DART* without Memory Constraints

We compare *DART* with our baseline *tcptrace*—a software tool used to compute RTTs on packet traces [121]. For this experiment, we assume that *DART* is not bottlenecked by the memory available in RT and PT tables, and that the available memory is fully associative (i.e., records can be stored in any available memory location).

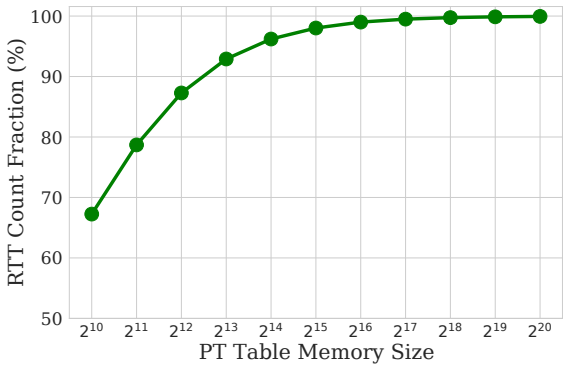
RTT sample count. Figure 3.9a illustrates the number of RTT samples captured by *DART* vs. *tcptrace* (*SYN* in the figure is short-hand for packets where the SYN flag is set, i.e., SYN and SYN-ACK). For the case where we collect handshake RTTs, i.e., tcptrace(+SYN) and *DART* (+SYN), *DART* is able to capture 7.53M RTT samples vs. 9.12M samples collected by *tcptrace*. The difference is explained by the fact that *DART* only keeps track of the latest contiguous unambiguous bytes when a *hole* appears in the sequence space, whereas *tcptrace* keeps track of all contiguous byte-ranges. Furthermore, unlike *DART*, *tcptrace* collects all valid RTTs when sequence numbers wrap around; additionally, *tcptrace* sometimes breaks one RTT sample into two (causing an inaccuracy) due to a design flaw.[3] In the case where handshake RTTs are foregone by ignoring SYN and SYN-ACK packets, i.e., tcptrace(-SYN) and *DART* (-SYN), *DART* captures 7.21M RTT samples vs. 8.66M samples captured by *tcptrace*.

RTT distribution. Figures 3.9b and 3.9c show the distribution of the RTTs captured in the four settings discussed above (i.e., *tcptrace* vs. *DART* and +SYN vs. -SYN). Figure 3.9b shows the distribution (CDF) of RTT samples with values between 0 and 125 ms. Figure 3.9c focuses on the tail and shows the complementary CDF (CCDF) of RTTs larger than 100 ms. The median RTTs for *tcptrace*(+SYN) vs. *DART* (+SYN) (14 vs. 13 ms) and for *tcptrace*(-SYN) vs. *DART* (-SYN) (15 vs. 13 ms) are comparable. There is a skew in the 95^{th} percentile, as can be observed from Figure 3.9b—*tcptrace*(+SYN) has it at

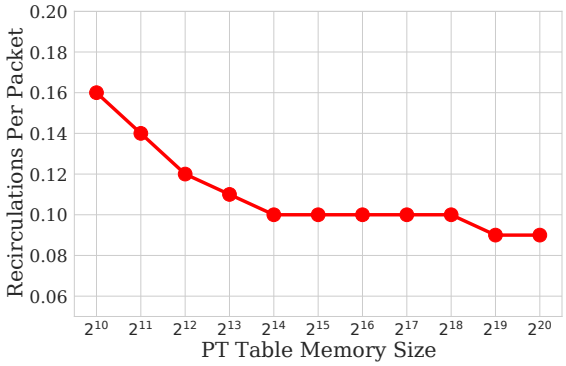
[3] *tcptrace* divides the sequence space (0 to $2^{31} - 1$) into four quadrants. We observed that when an RTT sample is generated for a packet that spans two consecutive quadrants, *tcptrace* wrongly generates an extra RTT sample for the part of the packet that ends in the earlier quadrant.



(a) RTT collection error



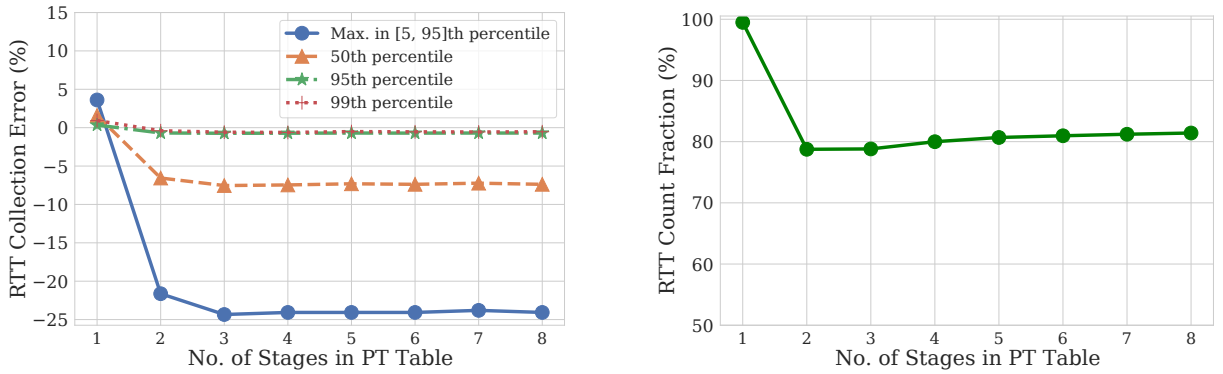
(b) Fraction of RTT samples collected



(c) Recirculations incurred per packet

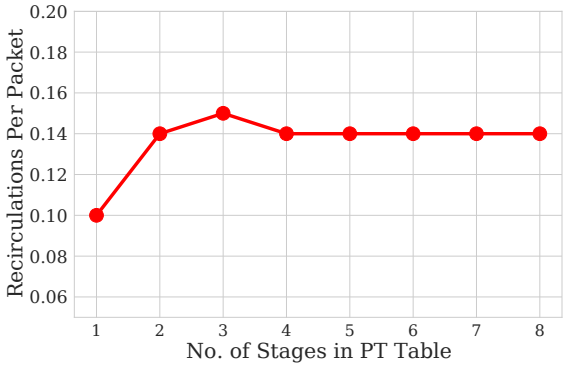
Figure 3.11: Performance of *DART* with a large RT table and varying PT table-size.

57 ms vs. 39 ms in *DART* (+SYN) (similarly, 62 ms in *tcptrace*(-SYN) vs. 39 ms in *DART* (-SYN)). The distributions converge at the tail, with 99^{th} percentile at 215 ms for both +SYN settings and 218 ms for both -SYN settings. We also observe that the large majority of RTT samples (96.3%) fall between 10 ms and 100 ms (Figure 3.9b). The tail, however, is long, and RTTs as large as 100 seconds or more are also seen (Figure 3.9c). Further analysis revealed that there are instances in the trace where a data packet does not receive an ACK for many seconds before a TCP keep-alive finally ACKs it. This may be due to the fact that our vantage point misses the original ACKs (with short RTTs) to these data packets, or a behavior of TCP where only a distant keep-alive acknowledges the last seen data packet. In any case, *DART* collects the long RTTs that *tcptrace* also collects, demonstrating a lack of bias against large RTTs.



(a) RTT collection error

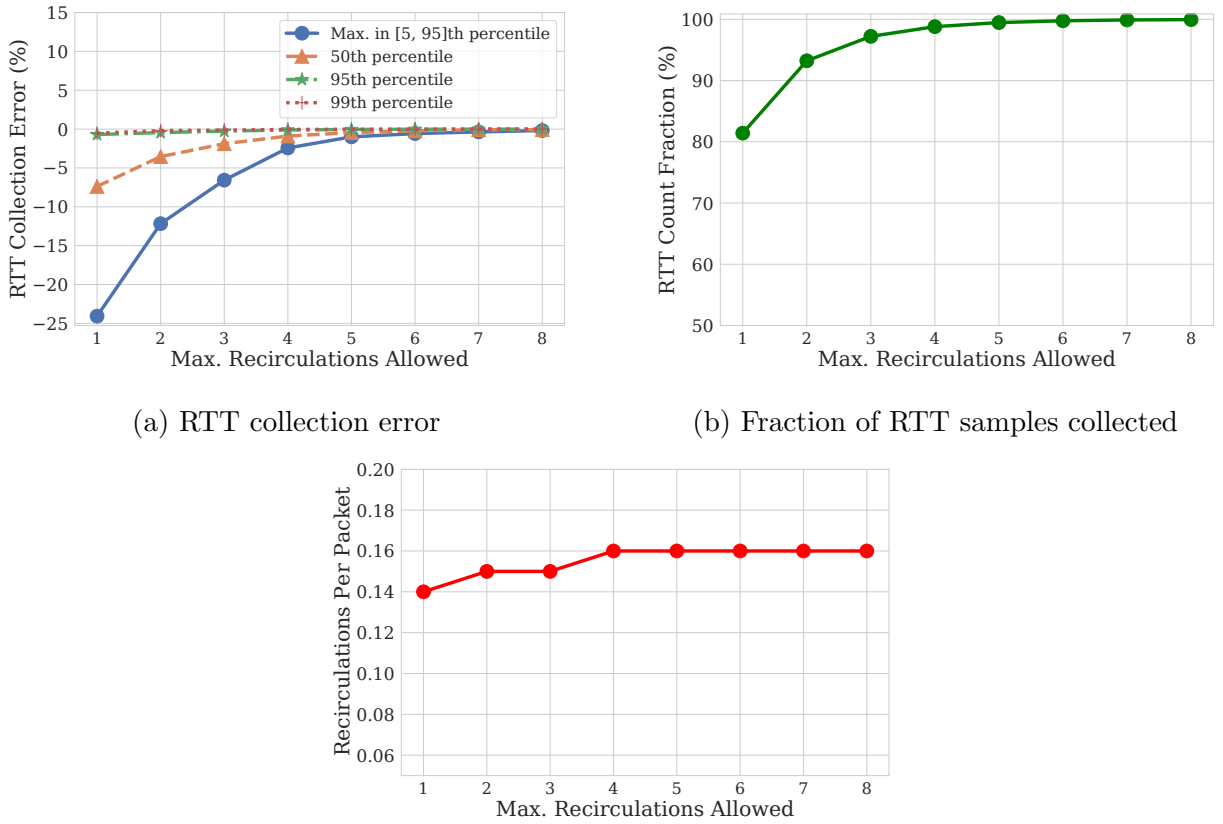
(b) Fraction of RTT samples collected



(c) Recirculations incurred per packet

Figure 3.12: Performance of *DART* with a large RT, fixed-size PT, and varying no. of PT stages.

Handshake RTTs. In order to understand the impact of foregoing handshake RTTs (i.e., by ignoring packets with the SYN flag set), we compare the relative memory savings obtained vs. the number of samples foregone. Figure 3.10 illustrates that around 1M out of 1.38M TCP connections (72.5%) seen in the trace are due to incomplete TCP handshakes! Ignoring SYN and SYN-ACK packets, therefore, provides significant advantages in terms of memory in the RT table, since otherwise these flows would occupy much of the table. The relative loss in terms of number of RTTs collected is much less—*DART* misses only 0.32M out of 7.53M samples (4.2%).



(a) RTT collection error

(b) Fraction of RTT samples collected

(c) Recirculations incurred per packet

Figure 3.13: Performance of *DART* with a large RT, a PT with a fixed size and no. of stages, and varying no. of allowed recirculations.

3.6.2 Impact of Table Configurations

In the ensuing experiments, we remove the assumptions that the tables are fully associative or of unlimited size. True to the prototype that can run on a hardware switch, the RT and PT tables are now one-way associative (i.e., only one memory location can be accessed per packet without recirculation). As a result, we grapple with contention for memory and recirculations (Section 3.3).

Baseline. Since *DART* (-SYN) from the previous subsection operates with unlimited and a fully associative memory, it is the best *DART* can do. The following experiments, therefore, treat *DART* (-SYN)'s performance—in terms of the number of RTT samples collected and the resulting RTT distribution—as the baseline. We notice that *DART* (-SYN) is actually

set up similarly to *tcptrace*, except for the fact that the amount of state it can track for each flow is *constant*, e.g., one measurement range. For this reason, we view it as a variant of *tcptrace* with constant space—we refer to it as *tcptrace_const*.

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

Impact of the PT table size. In this experiment, we try to determine the *right* size for our one-way associative PT table. First, we set the RT table size to a number *large enough* (i.e., 2^{20} in this specific case) to accommodate all flows in our campus trace. While this number is large, we reason that an operator would only monitor a subset of flows at a time, and therefore, during actual usage, the RT memory requirement would be small enough to fit in one stage of the Tofino (if not, the RT table could be expanded to a multi-stage

table). Second, we allow recirculations—we set the maximum allowable recirculations to 1 in the PT table. Now, we vary the PT table size between small ($2^{10} = 1,024$) and large ($2^{20} = 1,048,576$) and evaluate the impact on accuracy of RTT collection. Figure 3.11a shows the *RTT collection error*. We observe that, as expected, the error goes down as the PT table increases in size. The error at $p = 95$ is the least, followed by the error at $p = 99$, illustrating that *DART* is not biased against large RTTs. Figure 3.11b shows that the *fraction of RTT samples collected* increases with increasing memory size, as expected. In fact, *DART* collects more than 90% samples with a relatively small PT size (i.e., $2^{13} = 8,192$). We also observe from Figure 3.11c that 16 recirculations happen per 100 packets for a small PT table (i.e., size= 2^{10}), which goes down to 10 and lower as we increase the PT table size. For the next experiment, we fix the PT table size to $2^{17} = 131,072$: this is the lowest PT table size that results in less than 5% error for any value of p between 5 and 95 and collects more than 99% RTT samples.

Impact of the number of PT table stages. We explore the idea of implementing a multi-stage PT table, i.e., a k-way associative PT table. While a multi-stage PT table may be difficult to fit in Tofino1 or in the ingress pipeline of Tofino2, we can implement it in the egress pipeline of Tofino2. We fix the RT table size to *large enough* as before, the PT table size at 2^{17}, and maximum recirculations per packet at 1. We now try to determine the best value of k to divide the PT table into. Figure 3.12a shows the *RTT collection error* against k. We find that the error at $p = 95$ and $p = 99$ stay stable around zero, whereas the errors at $p = 50$ and the maximum error between $p = 5$ and $p = 95$ increase as soon as we divide the PT table memory into more than 1 stages. The error increases in the negative direction (i.e., *DART* starts overestimating the median RTT). This can be attributed to the fact that some smaller RTT samples are missed when there are more stages, since there is just enough space for records with large RTTs to get inserted but never get evicted (since older records are preferred). We see a similar effect on the *fraction of RTT samples collected* (in Figure 3.12b) and the *recirculations incurred per packet* (in

Figure 3.12c)—these metrics get significantly worse when the PT table is divided into 2 stages, and remain bad as the number of stages is gradually increase to 8. We determine from this experiment that simply dividing the PT table into multiple stages, without adding more memory, worsens performance. In the next experiment, we explore whether allowing more than 1 recirculation per packet might help a multi-stage PT.

Impact of the maximum number of allowed recirculations. In this experiment, we again ensure that the RT table is *large enough*, fix the PT table size to 2^{17}, and divide it across 8 stages. This time, we allow the maximum number of recirculations per packet to vary between 1 and 8. Our intuition is that allowing the hash-collided PT records to pass through the PT more times might help them find alternate locations to *live on* in the PT during memory pressure. Figure 3.13a shows that the RTT collection error rapidly improves as more recirculations are allowed; for a 8-stage PT, 4 recirculations bring down the error to nearly zero. We also see from Figure 3.13b that the *fraction of RTT samples collected* goes up to 99% and beyond when 4 or more recirculations are allowed. This improvement is achieved without the *number of recirculations incurred per packet* ever going up beyond 0.16, as shown in Figure 3.13c. We conclude that we can take advantage of spanning the PT across multiple stages of the Tofino—thereby increasing the total memory size beyond what 1 Tofino stage would allow—and still ensure good performance, if we also allow more recirculations per packet. This good performance is achieved without the consumption of significant recirculation bandwidth (16 recirculations per 100 packets in the worst-case).

3.7 Discussion

Limitations of *DART*. While *DART* is designed for accuracy in the face of TCP ambiguities, the tool cannot handle all situations accurately. First, if *DART* starts (or restarts after eviction) tracking a flow that is already in progress, it may not have enough information to infer retransmission or reordering. For example, what the RT table sees as a

new SEQ packet may actually be a retransmitted copy of a SEQ packet sent before *DART* started tracking the flow. Second, reordering may happen downstream from the monitoring device; in such cases, not just *DART* but no other RTT measurement tool can measure RTT accurately, since the vantage point is never aware of such an event. Apart from inaccuracies, we may sometimes conservatively close the measurement range when it could have collected valid samples. When we see an ACK packet that acknowledges the left edge of the measurement range, we infer a duplicate ACK and collapse the range. However, since the left edge does not always indicate the highest byte ACKed, but can indicate the highest byte affected by a retransmission or reordering event, the decision to close the measurement range conservatively may cost us some valid RTT samples. Similarly, we forego valid samples at the uppermost sequence numbers when a sequence number wraparound event happens; however, such events are rare—only 4 in our 15-minute campus trace. Sometimes, when the monitoring device does not see the last ACK in an exchange—either because of a packet drop at the device or a connection that closed abruptly—it may continue holding the RT and PT table records indefinitely, since we favor older entries over newer ones. Certain attacks can exploit this vulnerability by leaving a large number of data packets unacknowledged. A timeout mechanism for the RT table, with a very large timeout value (in seconds) could help.

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

inconsistent. This subsequently implies that the copy of the RT is *approximate* in nature. Moreover, such an approach requires additional memory space. Thus, this approach trades recirculation overhead with memory space.

End-host delays. When a TCP receiver anticipates having some data of its own to send, it may wait until a certain number of bytes accumulate and then *piggyback* the ACK along with the data—a strategy known as delayed ACKing. The waiting time is reflected in the RTT samples collected by measurement tools like *DART* and *tcptrace*. Some use cases may, however, require RTTs that do not include this end-host delay. Delayed ACKs are notoriously difficult to identify by a monitoring device in the end-to-end path. This is because there is no known way to infer whether an ACK has been intentionally delayed by the receiver. Delayed ACK implementation is known to vary widely across device types and operating systems. In fact, the QUIC protocol has proposed provisions to explicitly tag delayed ACKs to alleviate such measurement ambiguities [84]. Still, even without explicit detection of delayed ACKs, simple techniques such as "min. filtering" (e.g., computing the minimum RTT sample over a time window, as discussed in Section 3.3.3) can be quite effective at separating propagation delay along the network path from end-host delays such as delayed ACKs.

Extending *DART* to QUIC and IPv6. The QUIC protocol does not expose sequence and acknowledgment numbers as TCP does. Therefore, it is not possible to measure RTTs for QUIC packets using the same mechanism as *DART*. QUIC, however, uses a *spin bit* to indicate reversal of direction by a sender or receiver [84]. It is possible to track this bit and compute RTTs; however, this is fraught with challenges. First, the spin bit allows the measurement of a maximum of one valid RTT sample per congestion window. Second, inferring retransmissions or reordering is not possible using only the spin bit. Nevertheless, RTTs collected using the QUIC spin bit can augment RTTs collected for TCP traffic to the same destination. *DART* can also be extended to work with IPv6 by adjusting how the payload size is computed. However, since the 4-tuple size is much larger in IPv6, and the

RT flow signature size is fixed, *DART* may encounter more hash collisions.

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

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

Dealing with optimistic ACKs. Misbehaving TCP receivers can ACK data packets *before* they are received, to manipulate the sender into transmitting faster [130]. *DART* is largely robust to such *optimistic* ACKs, since it ignores any ACK packet that arrives before the corresponding data packet. In fact, *DART* can be easily extended to detect and report optimistic ACKs. However, if an optimistic ACK happens to reach *DART* *after* the corresponding data packet has already arrived at *DART* (but not at the receiver), *DART* (and any other passive monitoring tool) would be misled into collecting an incorrect RTT sample. We reason that it is difficult for the receiver to reliably time optimistic ACKs in this way.

3.8 Related Work

Passive RTT analysis. Previous work has proposed the idea of passive RTT monitoring [72] using SYN/SEQ and ACK packets. *tcptrace* [121] is a passive, open-source TCP traffic-analysis tool that can measure RTT per SEQ/ACK packet pair within a given packet trace. It runs in software on a general-purpose CPU. We use this tool as our ground truth for evaluating *DART*'s correctness as *tcptrace* is free from time and memory constraints. That said, unlike *tcptrace*, *DART* can generate RTTs on live traffic in real time. Also, while *tcptrace* rounds RTTs up to the nearest millisecond, the Tofino enables *DART* to report RTTs down to a nanosecond granularity, which may be useful in extreme low-latency use cases.

Instead of matching data and ACK packets, the *TCP time-stamp* option available in the TCP header can also be used for passive RTT analysis. The `pping` tool [116] adopts this approach. However, TCP timestamps are often too coarse-grained (e.g., 10 or 100 ms granularity), and many services do not use them at all since the TCP timestamp is an optional field [47]. Also, an end-host puts in its timestamp clock as the field value, but different TCP implementations increment their internal clocks at different rates—some by 100 every second, and others by 1000 [19]. This makes computing the absolute latency in milliseconds troublesome in a monitoring device that does not know the end-host's clock tick rate.

Passive RTT monitoring in the data plane. Dapper [56] is an early work that aims to measure TCP performance in the data plane. However, it can only track a single packet per congestion window when doing RTT measurement; it has to wait until the corresponding ACK arrives before starting to track another packet's RTT. In use cases where RTTs are large, or aggregate statistics (e.g., min., median, etc.) are being collected over time-windows, Dapper would report too few samples per unit time to be useful. Chen et al. [32] expanded Dapper's idea and proposed an algorithm and data structure that can track multiple SEQ

packets simultaneously in the data plane. However, this previous work does not correctly deal with ambiguities in TCP, such as packet retransmission and ordering. It also uses the memory space inefficiently and produces RTT samples biased against long RTTs. We use this previous work as our strawman in Section 3.2. Zheng et al. [186] proposed a novel data structure called *fridges* to measure delay directly in the data plane with the focus on collecting RTT samples without bias against larger delays. This is done by applying a correction factor inversely proportional to the probability of producing an RTT sample. This previous work, however, does not address the accuracy and memory efficiency issues caused by TCP ambiguities.

Liu et al. [90] proposed algorithms that run in the data plane for four separate performance-monitoring tasks: round-trip latency, packet loss detection, out-of-order detection, and retransmission detection. This previous work focuses more on using memory sublinear in both the size of input data and the number of flows, and the four performance-monitoring algorithms are independent of each other. Also, the round-trip latency is *not* calculated in a per-packet matching scheme; rather, it computes the average RTT by subtracting the sum of all server-to-client (SEQ) packets' timestamps from the sum of all client-to-server (ACK) packets' timestamps and then computes the average, assuming no missing or duplicate SEQ or ACK packets. RouteScout [6] measures packet loss and delay in a programmable data plane to help select a better Internet path driven by performance in real time. However, RouteScout measures round-trip time only at TCP handshake using the first SYN and ACK exchange. Thus, RouteScout only produces one RTT sample per flow during connection establishiment. In contrast, *DART* focuses on accurate, continuous round-trip time monitoring on a per-packet basis while taking into account the idiosyncrasies of TCP, including packet loss, reordering, and retransmission.

3.9 Conclusion

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

Chapter 4

Passive Data-Plane Telemetry to Mitigate Long-Distance BGP Hijacks

Poor security of Internet routing enables adversaries to divert user data through unintended infrastructures in attacks known as hijacks. Of particular concern—and the focus of this chapter—are cases where attackers reroute domestic traffic through foreign countries and still deliver it to the intended destination, exposing traffic to surveillance, bypassing legal privacy protections, and posing national security threats. Efforts to detect and mitigate such attacks have focused primarily on the control plane, while data-plane signals remain largely overlooked. In this chapter, we argue that *passively-monitored round-trip time (RTT)*—and, in particular, changes in its *propagation-delay* component—offers a promising signal for detection: the increased propagation delay is unavoidable and directly observable from affected networks, enabling opportunities for faster detection and mitigation. We explore the practicality of using RTT variations for hijack detection, addressing two key questions: (1) What coverage can this provide, given its heavy dependence on the geolocations of the sender, receiver, and adversary? and (2) Can an always-on RTT-based detection system be deployed without disrupting normal network operations? Focusing on cross-country interception attacks, we find that coverage is high: 97% under ideal (i.e., data travels at the speed

of light) conditions, and 91% and 86% with real traffic from two datasets. To demonstrate practicality, we design HiDe, which reliably detects delay surges from long-distance hijacks at line rate using commodity programmable hardware. We measure HiDe's accuracy and false-positive rate on real-world data and validate it with ethically conducted hijacks.

4.1 Introduction

BGP (Border Gateway Protocol) hijacks are a long-standing threat where attackers exploit the poor security of BGP—the Internet's default routing protocol—to redirect traffic through their own infrastructure. These attacks can be dangerous, allowing the theft of sensitive data and inflicting severe financial damage [2, 5, 12, 28, 35, 59, 117, 126]. Despite years of research, BGP hijacks remain a serious threat today [14, 127]. Major prior efforts have attempted to *proactively* neutralize this threat, but suffer from limited adoption or scope: Standards like BGPsec [10] and clean-slate alternatives such as SCION [124] require ubiquitous adoption to be truly effective, which is challenging given the decentralized nature of the Internet. Meanwhile, mechanisms like RPKI (Resource Public Key Infrastructure) [25] can prevent only a subset of attacks (i.e., forged-origin hijacks), leaving other attack classes unaddressed.

Consequently, network operators rely heavily on *reactive* approaches to defend against BGP hijacks. State-of-the-art reactive approaches primarily work in the control plane by monitoring BGP feeds from public monitors like RIS and RouteViews, private commercially-operated monitors, and/or the network's own routers. Such systems (e.g., ARTEMIS [145], DFOH [66]) detect suspicious announcements, and mitigate attacks using techniques such as announcing *more-specific* prefixes. These approaches detect attacks quickly and accurately when malicious announcements are immediately visible from their vantage points, but they fall short in two key scenarios. First, they may entirely miss attacks in which malicious announcements are specifically designed to evade control-plane-based detection [14, 16, 107, 181]. For example, attackers can manipulate BGP communities to surgically steer traffic

while limiting visibility at public monitors [16]. Similarly, in *interception attacks*, the adversary diverts traffic through its own infrastructure but still forwards it to the intended destination, reducing the likelihood of detection [9, 58, 181]. Second, some attacks may eventually become visible at route monitors, but only after sufficient time has passed for the attacker to cause significant damage. For example, website fingerprinting attacks can succeed within a few seconds of the hijack, since they may require only the first few packets, bursts, or bytes of page content [45, 148, 151]. Similarly, in hijacks such as the one on Amazon's DNS infrastructure in 2018, attackers could collect users' credentials within the first few seconds, which later enabled them to steal cryptocurrency [125]. Other reactive approaches (e.g., iSPY [185]) implement *active probing* in the data plane (e.g., ping, traceroute), but require responsive IPs and sustained probing to maintain accuracy, resulting in challenges with scalability (due to probe traffic overhead) and coverage (since some IPs may not be responsive). Hybrid methods (e.g., HEAP [131], Argus [149]) combine control and data plane signals to improve accuracy, but do not alleviate challenges with scalability, attack visibility, or detection speed.

In this chapter, we argue that *passively-monitored round-trip time (RTT)* is a highly useful yet underexplored data-plane signal for detecting BGP hijacks. In particular, hijacks induce a change in the *propagation-delay* component of RTT by changing the physical path of traffic. Unlike control-plane signals, RTT cannot be hidden from the victim. For instance, if an attacker from North Korea launches an interception attack on a connection between two hosts within the UK, the traffic must cover an additional $15,025$ kilometers at least, causing a minimum additional RTT of 75 ms[1]—an effect the victim will directly experience. The RTT change is also *immediate*, enabling an opportunity for faster detection and mitigation compared to existing reactive approaches. Additionally, contrary to active probing, passive monitoring does not require responsive IPs and incurs no probing overhead.

[1]This example assumes the speed of data transmission to be the speed of light in optical fiber, given by $c_f = 2c/3$ (approx.), where c is the speed of light in vacuum [89]. Hereafter, in this chapter, *speed of light* refers to c_f, which is approx. 200 km per millisecond (more precisely, 199.86 km/ms).

However, building an effective RTT-based BGP hijack detector is challenging. The expected increase in propagation delay—which is at the heart of an RTT-based approach—is highly *location-dependent*. To be detectable, this increase in propagation delay must be large enough to stand out from the natural RTT variation that occurs in real traffic. Such variation can be caused by many factors, including network congestion, host processing times, noise in access networks, and benign route changes (e.g., due to traffic engineering). As a result, to be reliably detected, the diversion must be geographically long enough to induce an RTT increase noticeably higher than the natural variation. For example, an RTT-based detector might successfully flag traffic between hosts in the US being diverted through the UK. In contrast, diversions over shorter distances may be difficult, and in some cases even impossible, to detect using RTT alone—for example, if US traffic is diverted through Canada, or in regions such as Europe where many countries are physically close and even cross-country detours may not induce a sufficiently large RTT increase. For this reason, RTT is not a silver bullet that enables a general-purpose solution for hijack detection.

Despite its limitations in scope, an RTT-based hijack defense is promising since it addresses key weaknesses of existing approaches in four main ways. First, it can detect the *long-distance* subset of stealthy attacks that control-plane-based approaches miss entirely. Second, for attacks that are not immediately visible in the control plane, it can detect the long-distance subset much more quickly (sometimes within milliseconds for high-data-rate flows), thereby reducing the amount of traffic exposed to the attacker before mitigation. Third, it can serve as an additional signal for existing hijack detectors, where combining control-plane and data-plane information can improve robustness and reduce false positives. Fourth, in such a combined setting, by virtue of its placement in the data plane, it enables finer-grained, operator-configurable mitigation strategies compared to control-plane-based mitigation that can only operate at the granularity of an entire prefix (e.g., a /24)—for example, traffic to sensitive IPs within the prefix could be temporarily rate-limited, or even

blocked when the data plane suspects an attack, awaiting confirmation from the control plane.

Among BGP hijacks that cause substantial harm, and that lie within the scope of BGP hijacks where an RTT-based defense is most effective, one class is of particular concern and is the focus of this chapter: hijacks that reroute domestic traffic through a foreign location and still deliver it to the intended destination. These *cross-country interception attacks* are especially troubling because they, unbeknownst to the user, expose the user's traffic to different jurisdictions and thus to different privacy and surveillance laws [57, 58, 181]. However, as noted earlier, an RTT-based defense may not be able to detect *all* cross-country attacks (e.g., US via Canada, Germany via France, etc.) Our first major goal in this study is to understand the effectiveness of RTT-based hijack detection for cross-country attacks. In particular, we ask the research question (RQ1): *What fraction of possible cross-country interception attacks can an RTT-based approach reliably detect?* We find that cross-country attacks are highly detectable using RTT: overall, 97% can be detected reliably under ideal conditions (i.e., data travels at the speed of light and is not affected by real-world noise), and detection remains high at 91% and 86% even with real traffic from two datasets (Section 4.3).

Although cross-country interception attacks are highly detectable, doing so in a scalable, real-time manner remains challenging. Per-packet RTT calculation is expensive at line rate, and maintaining state while monitoring every flow is intractable. Thus, we pose the following research question (RQ2): *Can we design a practical, always-on monitoring system to detect and mitigate cross-country interception attacks in a scalable manner without excessive cost?*

To this end, we design *HiDe*, a practical RTT-based system for detecting hijacks. First, BGP hijacks occur at the IP-prefix level and affect all traffic routed to the targeted prefix, meaning that during a real hijack, no flow to the targeted prefix can have an RTT less than the minimum required for the attacker's route. By passively measuring the RTTs experienced by as many packets as possible and relying on the minimum per prefix, *HiDe* can reliably and scalably detect spurious RTT surges. Second, we observe that a BGP hijack

induces a distinct pattern in the *denoised* RTT over time, clearly differentiating it from other events, such as congestion. Concretely, a hijack causes a *sharp* surge with location-dependent but calculable *minimum height*, which one can detect using a changepoint detection algorithm. Third, implementing *HiDe* can be made practical by deploying it on high-speed programmable hardware (e.g., switches). This is possible, despite the rigid computation and memory constraints of such devices, thanks to our switch-native implementation of changepoint detection and scalable latency measurements.

Our evaluation demonstrates that *HiDe* is reliable (zero false negatives by design), minimally disruptive to real traffic, and implementable on commodity hardware. To assess *HiDe*'s fidelity, we tested it against ethically-conducted real-world hijacks and found that it detects them within 0.5 second. Additionally, to evaluate its impact on regular operations, we run *HiDe* on campus network traces (19 billion packets, 5.3 TB bytes). The results show that its combination of algorithms effectively minimizes false alarms ($< 0.012\%$), even in the presence of highly noisy real-world RTT signals. Furthermore, *HiDe* reduces the impact of such false alarms by identifying and correcting them within a median of 0.75 seconds without human intervention. We implement *HiDe* entirely on a programmable switch, showcasing its potential for seamless deployability on a network's border gateway with minimal hardware cost and no delay overhead to normal traffic.

4.2 Background and Threat Model

4.2.1 BGP-Based Attacks

BGP is the primary protocol that connects Autonomous Systems (ASes) by enabling them to exchange and forward route announcements for IP prefixes. Each AS advertises routes for the prefixes it owns, including an AS path indicating the sequence of ASes to traverse to reach it. Routers independently select the best route for each prefix based on attributes like path length and routing policies.

BGP Hijack. A BGP hijack occurs when a malicious or compromised AS falsely advertises routes to IP prefixes it does not own or cannot reach, misleading other ASes into rerouting traffic through its infrastructure. Suppose $AS100$ legitimately owns the IP prefix 1.1.1.0/24. A malicious AS, $AS200$, falsely announces ownership of 1.1.1.0/24 to its BGP peers. These peers may accept the announcement as valid and propagate it to their own peers, spreading the false route across the network. As a result, traffic destined for 1.1.1.0/24—for instance, originating from another prefix like 2.2.2.0/24 owned by $AS300$—may get misrouted to $AS200$ instead of reaching $AS100$. This enables the attacker to eavesdrop on, fingerprint, manipulate, or drop this illegitimately-obtained traffic. In some cases, the attacker may even serve malicious content by impersonating the legitimate destination (e.g., by acquiring a *valid* certificate first by exploiting weakness in the certificate issuance verification process [15]).

BGP Interception Attack. A BGP *interception attack* (Figure 4.1) is a specific type of BGP hijack where the attacker intercepts traffic but forwards it to the original destination, enabling analysis or manipulation while remaining undetected by the end hosts. Of particular concern is a stealthy subset of these attacks where a sophisticated attacker employs techniques like *AS-path poisoning* and the manipulation of *BGP communities* to limit the propagation of malicious announcements in a bid to evade detection by BGP monitors near the victim. For instance, in the example above, $AS200$ could manipulate its announcements to propagate only to routers near $AS300$ while suppressing those to routers near $AS100$. This could cause traffic from 2.2.2.0/24 to 1.1.1.0/24 to get misrouted via $AS200$, while $AS100$ remains unaware as BGP monitors near it never observe the malicious route. Such an attack has been demonstrated by Birge-Lee et al. [16], which we ethically reproduce in Section 4.7.2.

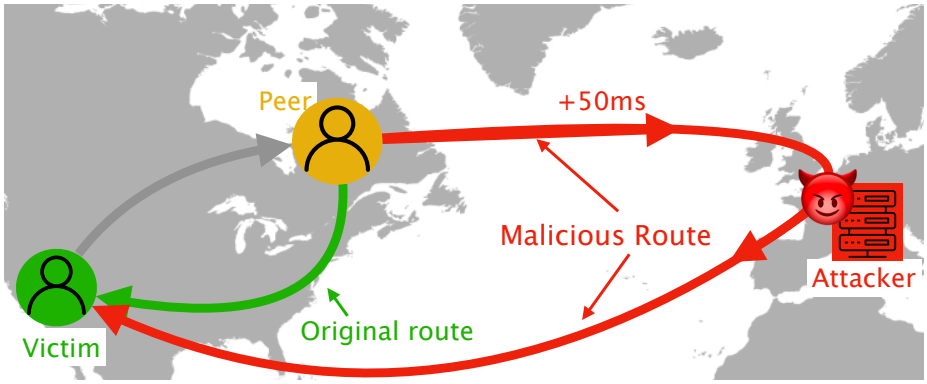


Figure 4.1: An attacker in the UK exploits the weakness of routing security to redirect traffic from a *peer* host in the US—originally destined for a *victim* host in the US—through the attacker's own infrastructure. The *mid-attack* path (in red) from the peer to the victim is longer than the original *pre-attack* path (in green), adding an extra 50 ms of propagation delay to the RTT of the traffic.

4.2.2 Threat Model

We consider an adversary performing a *stealthy* BGP interception attack—such as the one described above—to reroute traffic destined for a victim through distant infrastructure before forwarding it back to the victim. This infrastructure is located in a different country, potentially under different privacy and security laws. The adversary is sophisticated, aware of detection systems, and employs evasion techniques to prevent forged announcements from reaching BGP monitors leveraged by the detection systems [16]. By rerouting traffic back to the victim, the attacker keeps connections alive, enabling traffic analysis while evading detection at the application layer. Such attacks can serve as tools in cyber warfare or surveillance. Real-world examples of long-distance interceptions include (among numerous others) the rerouting of US-based traffic via the UK to enable surveillance (Figure 4.1) [57], rerouting of US-based traffic managed by China Telecom via China undetected over 2.5 years [58], and rerouting of traffic between two hosts in Denver, USA via Iceland [181].

4.3 Feasibility Study

HiDe relies on the propagation-delay component of RTT for real-time BGP hijack detection. Propagation delay is the time it takes a packet to travel across the network path from the sender to the receiver, determined primarily by the physical distance and the transmission medium, rather than by network congestion or processing delays at the end hosts. In this section, we examine the *feasibility* of using propagation delay alone to defend against cross-country interception attacks. In such attacks, both the victim and its peer reside in a *victim country* C_V, while the attacker operates from a different *threat country* C_T. Such attacks are common in cyber warfare and surveillance carried out by nation states.

4.3.1 Key Questions and Observations

We ask the following questions about cross-country attacks:

1. Does traffic within a single country experience significantly lower propagation delay compared to traffic across countries? If so, can we use this difference to detect cross-country interception attacks?

2. Are all countries equally *defendable* using propagation delay-based detection? If not, and some countries are more defendable, what geographic factors drive this difference?

3. What fraction of cross-country attacks can, in principle, be detected by comparing propagation delays—both under idealized assumptions and real-world measurements?

Our analysis in this section reveals the following:

1. Across 258 countries, the maximum distance within a country's borders (*max. intra-country distance*) is typically far smaller than the minimum distance from that country to any other country (*min. inter-country distance*). The $25^{th}/50^{th}/75^{th}$ percentiles of max. intra-country distances are 49 km, 413 km, and 1,129 km, respectively; the

corresponding percentiles for min. inter-country distances are 4,027 km, 7,689 km, and 11,420 km. Naturally, propagation delays follow a similar trend (Section 4.3.2).

2. Some countries are more defendable using propagation delay-based detection than others. For example, Russia ranks among the least defendable, whereas New Zealand is one of the most defendable. More generally, larger countries with many nearby neighboring countries are less defendable, while smaller or more geographically isolated countries are more defendable (Section 4.3.4).

3. Considering the worst-case (least defendable) attack paths between every pair of countries, we find that 97% cross-country attacks can be detected assuming speed-of-light RTTs, and 91% and 86% respectively, using real-world measurements from two production datasets (Sections 4.3.4 and 4.3.5).

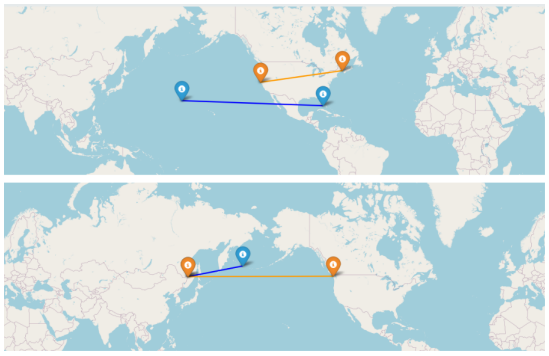
4.3.2 Intra-Country vs. Inter-Country Distances

Our first goal is to assess whether the great-circle distance (shortest distance along Earth's curvature) between two hosts in the same country is significantly smaller than between hosts in different countries, so a cross-country detour would induce a detectable increase in propagation delay.

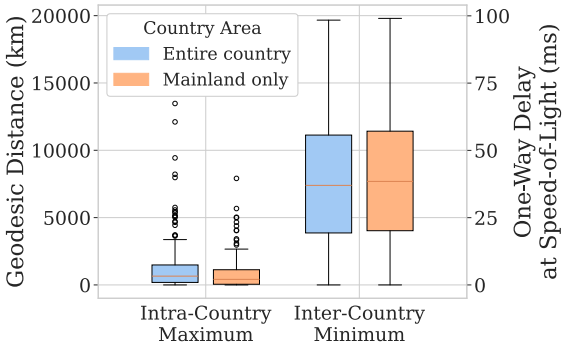
Dataset. We use the *Natural Earth Admin-0 Countries* dataset—one of the most popular boundary datasets in the Geographic Information System community. It provides high-resolution boundary coordinates (avg. 2.5 km) for 258 countries (Figure 4.2a) [42]. Since some countries consist of multiple disconnected regions (e.g., contiguous US plus Alaska and islands), we distinguish each country's *mainland* (largest contiguous landmass) from *entire country* (all regions combined).

Method. For each country (entire country and mainland), we calculate: (1) Max. intra-country distance: the largest pairwise great-circle distance among all boundary points of

(a) Centroids (orange icons) of the mainlands of 258 countries in the Natural Earth Admin-0 dataset [42].



(b) Top: Max. distance within country. Bottom: Min. distance between countries. (Blue: Entire country, orange: mainland).



(c) Distribution of max. within and min. between countries: distance (left y-axis) and one-way delay at c_f (right y-axis).

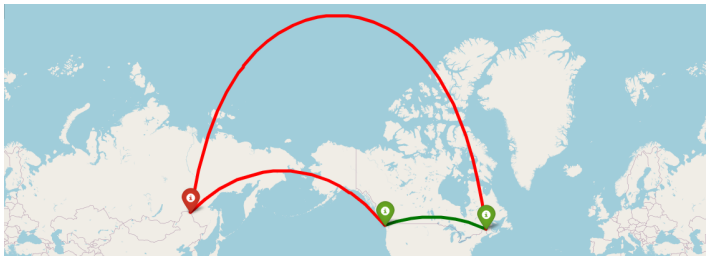
Figure 4.2: For all 258 countries (Figure a)—using both entire country areas and mainlands only (Figure b)—we compute each country's maximum internal distance and its minimum distance to every other country, then plot both distributions (Figure c). Typically, a country's foreign neighbors are more distant than its own farthest points.

the country, and (2) Min. inter-country distance: the smallest great-circle distance from any point in this country to any point in another. Figure 4.2b illustrates these distances within the US (top) and between the US and China (bottom).

Observations. Figure 4.2c shows: (1) min. inter-country distances far exceed max. intra-country distances, and (2) these distances for entire countries vs. mainlands are similar: we proceed with mainlands in the rest of this chapter for better interpretability. At the speed of light, the $25^{th}/50^{th}/75^{th}$ percentile max. intra-country one-way delay (OWD) are 0.2 ms,

(a) Distance before attack: $\delta_{pre} = 2\delta(S, D)$.

(b) Distance in the middle of the stealthy interception attack: $\delta_{mid} = \delta(S, D) + \delta(S, A) + \delta(D, A)$.



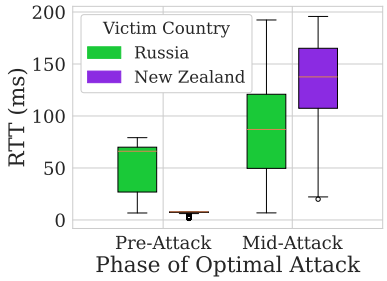
(c) Locations of source, destination in the US and attacker in China such that $\delta_{mid} - \delta_{pre}$ is minimized.

Figure 4.3: In this example, source S and destination D lie in mainland US and attacker A in mainland China. Figure (a) shows the *pre-attack* round-trip distance δ_{pre} and (b) the *mid-attack* round-trip distance δ_{mid}, leading to the deviation $\delta_{deviation} = \delta_{mid} - \delta_{pre} = \delta(S, A) + \delta(D, A) - \delta(S, D)$. Figure (c) shows the *most optimal attack* on the US from China, with curved lines indicating shortest great-circle paths. (Green: Original path, red: diversion due to attack.)

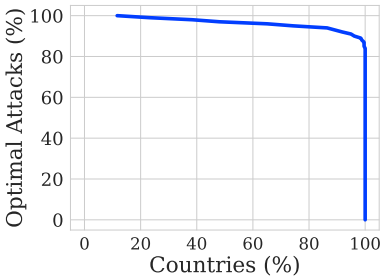
2.1 ms, and 5.6 ms, respectively. The corresponding values for min. inter-country OWD are 20 ms, 38 ms, and 57 ms—at least an order of magnitude larger.

4.3.3 Identifying Least Defendable Attacks

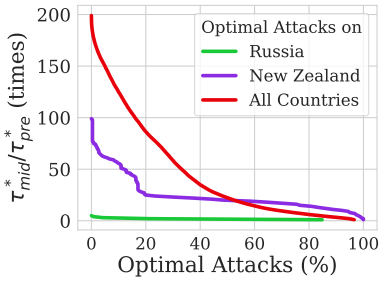
In this subsection, we describe how we identify the least defendable attack given a C_V and a C_T; later, we analyze defendability under ideal (Section 4.3.4) and realistic (Section 4.3.5) conditions. Figures 4.3a and 4.3b show the paths before and during the attack: with source S and destination D in C_V and attacker A in C_T, the round-trip distance changes from $\delta_{pre} = 2\delta(S, D)$ to $\delta_{mid} = \delta(S, D) + \delta(S, A) + \delta(D, A)$, resulting in a deviation of $\delta_{deviation} = \delta(S, A) + \delta(D, A) - \delta(S, D)$. In the worst-case (least defendable) attack scenario, $\delta_{deviation}$ is minimized by having S and D on C_V's border and as far apart from each other as possible, and



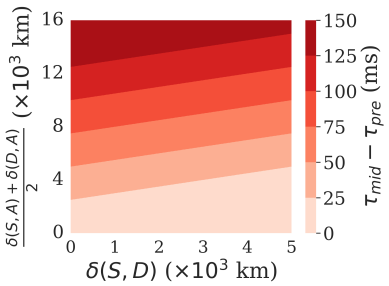
(a) Pre- and mid-attack RTTs during all possible optimal attacks on 2 example countries.



(b) Percentage of countries vs. percentage of defendable optimal attacks.



(c) Attack coverage vs. mid-attack RTT (as a multiple of pre-attack RTT).



(d) Deviation (z) as a function of dist. b/w hosts (x) and avg. dist. b/w hosts & attacker (y).

Figure 4.4: Defendability against optimal attacks assuming speed-of-light RTT: With Russia and New Zealand (NZ) as example victim countries, (a) shows mid-attack RTT is typically much higher than pre-attack RTT; size and proximity of victim country to other countries determine the extent. Figure (b) shows that for 86% countries can be defended against 94% optimal attacks. Figure (c) shows Russia's post-attack RTT peaks at 4× its pre-attack RTT (corresponding to 110 ms absolute difference), NZ at 100× (190 ms), and all countries combined at 198× (200 ms). Figure (d) shows that, when the victim and peer are co-located, the attacker must be 2,500 km away to induce a deviation of 25 ms.

A on C_T's border and as close to S and D as possible—we call this an *optimal attack*. Under such an attack, we denote the pre-, mid-attack distances, and deviation as δ^*_{pre}, δ^*_{mid}, and $\delta^*_{deviation}$, respectively, and the corresponding RTTs as τ^*_{pre}, τ^*_{mid}, and $\tau^*_{deviation}$. Figure 4.3c shows the optimal attack from China on the US, as an example. We compute the optimal attack for each ordered pair of victim and threat countries (258 × 257 attack scenarios) and use these scenarios in our analysis in the next two subsections.

4.3.4 Defendability under Ideal Conditions

In this subsection, we analyze the feasibility of detecting cross-country optimal attacks under *ideal conditions* (defined below).

Assumptions. *Ideal conditions* include: (1) Ideal network, i.e., data travels at the speed of light, and (2) Ideal measurements, i.e., our measurements capture the actual distance-based propagation delay. Under these assumptions, propagation delay = minimum RTT (minRTT) = RTT. Therefore, in this subsection, RTT is synonymous with propagation delay. We relax these assumptions in the next subsection where we analyze real-world measurements (Section 4.3.5).

Most and least defendable countries. Optimal attacks determine the lower bound of our defense capabilities. To assess defendability under optimal attacks, we compute propagation delay as the round-trip distance (δ^*_{pre} or δ^*_{mid}) divided by the speed of light. To illustrate the difference in defendability across countries, we select two examples: Russia, which has among the smallest median $\tau^*_{deviation}$, and New Zealand (NZ), which has one of the largest median $\tau^*_{deviation}$. Figure 4.4a shows their pre-attack and mid-attack RTT distributions. NZ's RTTs increase significantly mid-attack, making it more defendable; Russia's RTT changes are smaller making it less defendable. In general, small countries with distant neighbors (low $\delta(S, D)$, high $\delta(S, A) + \delta(D, A)$) are the most defendable, while large countries with many close neighbors are the least defendable.

Attack coverage. To quantify defendability, we define *attack coverage* as the percentage of optimal attacks that can be detected under some given condition. To compute overall coverage, we set the condition $\tau^*_{deviation} \geq 5$ ms because: (1) 5 ms far exceeds typical noise in measurements (e.g., due to coarse-grained timestamps, rounding off errors, approximations in distance calculations, etc.), and (2) At speed of light, 5 ms corresponds to approx. 1,000 km of extra path length, so it captures significant geographic detours. The attack coverage for Russia (over 257 optimal attacks) is 85% (the minimum for any country), while the

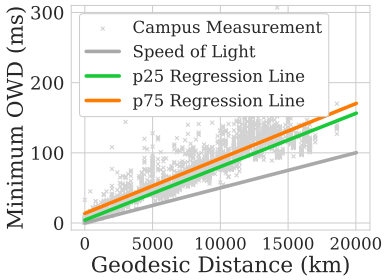
same for NZ is 100%. When expanded to all countries (i.e., 258 × 257 optimal attacks), the coverage is 96.6%. Figure 4.4b shows that 100% (i.e., all) countries can be defended against 84% attacks, 75% against 95%, 23% against 99%, and 11% against 100%. These results illustrate the promise and generality of propagation delay-based detection (in ideal conditions).

Attack coverage at given RTT deviations. To analyze the extent to which optimal attacks increase RTT in ideal conditions, we plot the attack coverage (x-axis) given the ratio between mid- and pre-attack RTT (Figure 4.4c). For NZ (purple), this ratio ranges from 1-100× (5-190 ms absolute difference), while for Russia (green), due to its large pre-attack RTTs, it is 1-4× (5-110 ms). For attacks on all countries (red), it is 1-198× (5-200 ms). The ratio is 2× (8 ms) for 95% attack coverage on all countries, and 4× (23 ms) for 85% coverage.

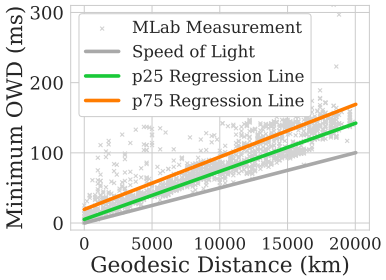
Deviation as a function of distances. The analysis of cross-country attacks does not inform us directly about the relationship between distance and RTT deviation. To bridge this gap, Figure 4.4d shows RTT deviation (z-axis)—as a function of pre-attack distance ($\delta(S, D)$) (x-axis), and average distance between S-A and D-A (y-axis)—in a heatmap. The x- and y-axis are capped at the maximum intra- and minimum inter-country distances. The $85^{th}/95^{th}$ percentile pre-attack distances are 1,090 km and 1,872 km; to induce an RTT deviation of 25 ms, the attacker needs to be at an average distance of 3,045 km and 3,436 km, respectively, from S and D.

4.3.5 Defendability in the Wild

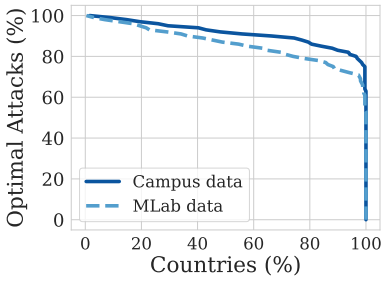
While our observations under ideal conditions are promising, defendability may differ in the real world because: (1) the actual propagation delay may be larger than the speed-of-light RTT due to longer physical paths, and (2) RTT measurements may not capture the actual propagation delay due to congestion or poor channel conditions (in wireless networks). In this subsection, we evaluate defendability in realistic conditions using two production



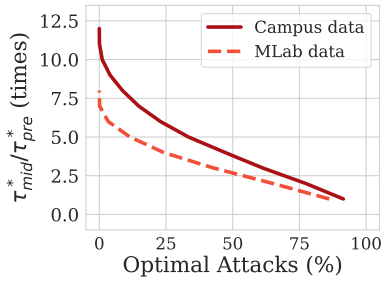
(a) Campus measurements over 12 hours in May 2022.



(b) Google MLab measurements over 5 days in Dec. 2024.



(c) % countries vs. % defendable optimal attacks.



(d) % attacks vs. ratio of mid-attack to pre-attack RTT.

Figure 4.5: Defendability against optimal attacks based on real measurements: We estimate (using linear regression) the $p25$ and $p75$ OWD for each 200 km distance bucket of our campus dataset and the Google MLab dataset, in (a) and (b) respectively. Using $p75$ OWD to estimate pre-attack and $p25$ to estimate mid-attack RTT, 85% countries can be defended against 85% attacks based on the campus dataset, and 85% against 78% based on MLab (Figure (c)). Figure (d) shows that the mid-attack RTT peaks at 12× pre-attack RTT in the campus dataset, and 7.5× in MLab.

datasets.

Datasets. Our datasets are outlined below:

1. ***Campus dataset:*** We collect traffic from 7.5M TCP flows on our campus over 12 hours on a weekday in May 2022, and compute RTTs by matching data packets with ACKs [32].

2. ***MLab dataset:*** We collect minRTTs of 4.3M TCP flows from NDT7-based measurements over 5 days in Dec. 2024 [101].

For each flow in each dataset, we collect geolocations of the source and destination. We discard flows whose minRTT indicates shorter distances than those permitted by their reported geolocations, which is physically impossible. Finally, we group remaining measurements by source-destination /24 prefixes, because (1) a /24 prefix is the smallest unit on which a BGP hijack can be launched, and (2) aggregating by prefix improves the chance of measuring true minRTT (see Section 4.5.2 for a more detailed discussion on the advantage of aggregation by prefix).

Estimating propagation delay from distance. To quantify defendability in realistic settings, we estimate real-world propagation delay between any two hosts from their great-circle distance d. This real-world propagation delay may vary across host pairs separated by the same distance d depending on: (1) Geolocation: Some regions in the world have denser network connectivity than others, (2) Routing policies: Some providers make shorter paths available than others, (3) Consistent queues: Some paths experience consistent queuing delay due to deep buffers and consistent traffic, etc. Furthermore, even if the true propagation delay is same, we may measure different minRTTs due to transient congestion. It is infeasible to collect reliable minRTTs from all possible attack locations in all 258 countries. Instead, we apply the following method to both our campus dataset (215 countries) and Google MLab (234 countries) to capture variability:

1. ***Bin distances:*** Divide all distances up to 20,075 km (Earth's diameter) into 200 km bins ($\approx$1 ms at c_f).

2. ***Assign prefixes to bins:*** For each source–destination prefix pair, compute its great-circle distance, assign it to the appropriate bin, and record its minOWD (minRTT/2).

3. ***Percentile computation:*** Within each bin, compute the pth percentile of these minOWDs for $p = 1, \ldots, 100$.

4. ***Regression fitting:*** Fit one linear regression per percentile across all bins (e.g., a p=1 line through every bin's 1-percentile).

Figures 4.5a and 4.5b plot the p=25 and p=75 regression lines for the campus and MLab datasets, respectively. The campus data shows a narrower inter-quartile range—likely because the location of one end is fixed (on campus) and the entire variability is due to the remote host—whereas for MLab, the servers and clients are in different locations. With these delay estimates, we proceed to evaluate defendability against optimal attacks under realistic conditions.

Estimating pre-attack and mid-attack minRTTs. The variability of minOWDs for the same distance poses a challenge: if our detection is unlucky, it could measure a higher percentile minRTT before the attack and a lower percentile during the attack, causing the deviation in minRTT to be much lower than speed-of-light deviation. To model such a scenario, we use the upper quartile (75^{th} percentile) minOWDs to estimate pre-attack minRTTs, and the lower quartile (25^{th} percentile) minOWDs to estimate mid-attack minRTTs. Then, we evaluate defendability using the same metrics as before.

Observations. Using the condition $\tau^*_{deviation} \geq 5\,\text{ms}$, the overall attack coverage is 91% on the campus dataset and 86% on MLab. Figure 4.5c shows that in the campus data, 100% countries can be defended against 63% attacks, 75% against 89%, and 2% against 100%. In MLab, 100% countries can be defended against 53% attacks, 75% against 80%, and <1% against 100%. Real-world defendability is therefore less than in the ideal case—especially in MLab, where minRTT variability is higher. Figure 4.5d plots coverage versus the mid-/pre-attack minRTT ratio. The ratio ranges from 1–12× (5–290 ms) for campus; and 1–7.5× (5–250 ms) for MLab.

4.3.6 Takeaways

Our analysis shows that propagation-delay measurements offer a highly effective signal to defend against cross-country interception, primarily because diverted paths almost always incur substantially greater delays than pre-attack paths. Although real-world variability de-

grades detection coverage compared to ideal conditions, our focus on worst-case (optimal) attacks means these results are conservative—actual detection performance will often exceed our current estimates. We believe our findings are sufficiently strong to justify designing an interception detector based solely on propagation delay. At the same time, a range of factors influences how accurately any given attack can be detected: the victim country's size and distance from potential adversaries; the exact geolocations and separation of victim and peer hosts; the true lengths of the pre-attack and mid-attack network paths; transient or persistent congestion along those paths; and the precision of our measurement and aggregation techniques. With these insights in mind, in the next section, we present *HiDe*—a scalable, always-on, data-plane system for detecting and mitigating interception attacks.

4.4 *HiDe*: Overview

HiDe is a system to detect and mitigate BGP interception attacks. HiDe runs entirely on a programmable switch and uses real-time minRTT measurements for attack detection. In this section, we present the key insights that drive *HiDe*.

Converting noisy RTT into a reliable detection signal. During a hijack, all traffic to a victim prefix must traverse the longer path via the attacker, so no RTT sample can be shorter than the minimum propagation delay via the attacker. *HiDe* exploits this by passively collecting RTT samples for every TCP data-ACK pair at the network border (thereby avoiding noise from the internal network), aggregating samples per prefix, and tracking the minRTT per time window. By monitoring these minRTTs, *HiDe* detects hijacks as *sudden, sustained spikes* in delay. For example (Figure 4.6), hijacking test traffic ethically from a Stockholm client via Amsterdam causes the minRTT to jump by about 20 ms at attack start and to fall back when the hijack ends. A changepoint detection algorithm can reliably identify such shifts.

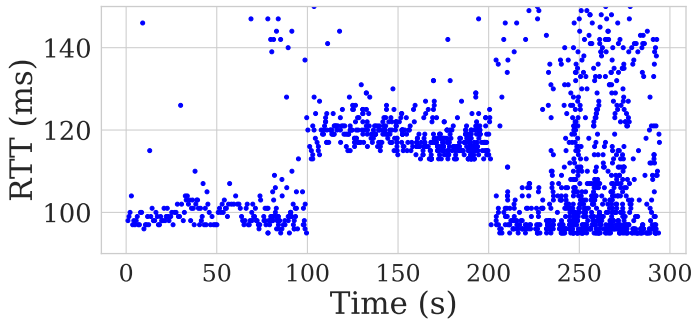


Figure 4.6: *Abrupt* and *significant* rise and fall in RTT due to interception attack launched (ethically) at 100^{th} second and withdrawn at 200^{th}.

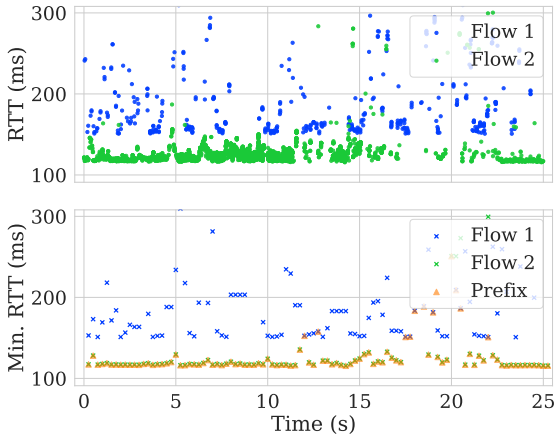


Figure 4.7: Top: Flow 1 (blue) with noisy RTTs and flow 2 (green) with stable RTTs. Bottom: Aggregating by prefix stabilizes minRTTs (orange).

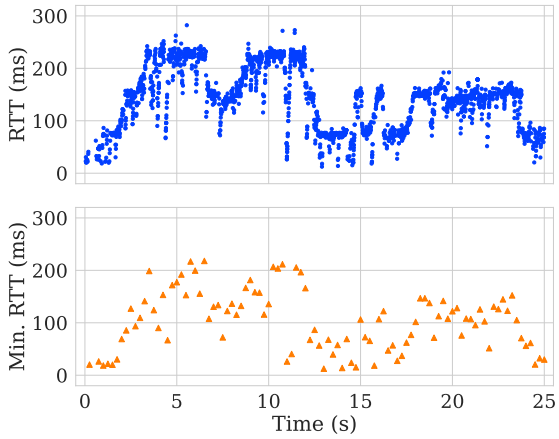


Figure 4.8: Prefix with noisy RTTs (top) that produce noisy minRTTs (bottom) despite windowing. Such prefixes are less defendable.

Prioritize guaranteed protection over broad coverage. We posit that operators would prefer a system that reliably defends a well-defined subset of prefixes rather than a best-effort approach that *covers* everything but floods them with false alarms. Section 4.5.3 shows how *HiDe* restricts its scope to prefixes it can protect with high confidence. When false positives do occur, *HiDe* (optionally, determined by the operator) continues measuring RTT and automatically rolls back its mitigation if the spike proves transient.

Optimize for commodity hardware. *HiDe* stores only per-prefix state—the running minRTT and count of RTT samples per time window—instead of expensive per-flow or per-packet state. It employs a lightweight, two-window, threshold-based changepoint detector that is hardware-amenable. On the Intel Tofino2, *HiDe* uses native primitives (*mirror* and *packetgen*) to generate packet replicas for RTT measurement, and occasionally—optionally—for false-positive correction and user alerting, while forwarding all other traffic at line rate with zero additional latency.

4.5 *HiDe*: Methodology

4.5.1 Computing Location-Based Lower Bound

Translating user input into geographic locations. The user provides *HiDe* with the IP prefix of the home network and the threat regions they want to protect their data from, based on policy decisions or anticipation of threats. The threat regions are either names of countries or enclosed polygons of geographic coordinates. *HiDe* also obtains from its data plane the destination prefixes observed by it. The user can, *optionally*, set threat regions *per destination prefix*. Eventually, the control plane converts all the information into triplets of geolocation information: {source_coordinates, destination_coordinates, threat_coordinates_list} using public geolocation services (*IPinfo*, *MaxMind*) and public geographic datasets (*Natural Earth Admin-0*) [42, 98, 158].

Computing lower bound of mid-attack RTT. For each location triplet, we first identify the *optimal attack*, i.e., the attacker's location in the threat region that minimizes mid-attack round-trip distance. Note that this distance can be much higher than in the optimal attacks in Section 4.3 because there, source and destination were always on the victim country's border whereas here, they are almost always inland. Next, we compute the minimum possible mid-attack RTT for this optimal attack (τ^*_{mid}), based on the speed of light. We designate this *lower bound* RTT as the *absolute threshold* of our changepoint detector: *HiDe* flags an attack whenever the observed minRTT reaches τ^*_{mid}, guaranteeing *zero false negatives* (see Section 4.5.6 for more on false positives). While we could choose a less conservative bound—e.g., the 25^{th}-percentile estimated latency in 200-km buckets (Section 4.3.5)—we opt for the most conservative threshold to *guarantee* protection, at the expense of coverage, as is our design goal (Section 4.4).

4.5.2 Reducing Noise in the RTT Signal

Aggregating by prefix to reduce impact of noisy flows. BGP attacks target prefixes, with a /24 prefix being the smallest possible target. All flows to an attacked prefix experience the same change in propagation delay, but noisy RTTs in individual flows can obscure this change. We aggregate RTT samples by prefix before computing the minimum RTT per window (discussed next), as at least one flow per window is likely to produce a sample representative of the true propagation delay. Figure 4.7 demonstrates this for a US-based destination prefix with one noisy and one stable flow. Also, prefix-level aggregation reduces switch memory requirements from per-flow to per-prefix, which is significant.

Windowing to discard short-term fluctuations. We divide streams of per-prefix RTT samples into non-overlapping time windows of a fixed size (i.e., *tumbling* windows) and compute the minRTT in each window. This filters out short-term RTT spikes due to benign confounding factors like queuing delay from short-lived congestion, end-host processing delays, and TCP oddities like *delayed ACKs* [144]. We select a sub-second (e.g., 100 ms) time

window—while minRTT can be measured more reliably with longer time windows, it would delay mitigation allowing an attacker more time to complete their attack. Finally, tracking minRTTs in *non-overlapping* tumbling windows requires only per-flow state, as opposed to *overlapping* sliding windows, making it more suitable for a switch implementation.

4.5.3 Vulnerable and Defendable Prefixes

Identifying vulnerable prefixes. BGP interception attacks primarily target prefixes that host sensitive services—government sites, banking portals, cryptocurrency nodes, and the like—because such websites handle sensitive data from users around the world. The attacker places itself between the user and the server—intercepting *valuable* data. *HiDe* prioritizes these vulnerable server prefixes for protection. By default, it excludes prefixes used exclusively by WiFi or cellular access networks—since they rarely host critical services—unless the operator explicitly includes them.

Identifying defendable prefixes. Some destination prefixes have RTTs that are *consistently* noisy or high even under benign conditions, making it nearly impossible to detect interception attacks on them from certain threat regions without excessive false positives. For example, Figure 4.8 shows a prefix where the minRTT often exceeds 100 ms, making it impractical to defend against a threat region that causes a small deviation in comparison. Further analysis of such prefixes reveals that often, they tend to be associated with client-side access networks, such as cellular or WiFi, which *HiDe* does not defend by default anyway. Concretely, during a *profiling* phase independent of the detection phase, we monitor the *max. of min. RTTs* in tumbling time windows for each destination prefix. Later, we defend a prefix against a threat region only if $\tau^*_{mid} - max(RTT_{min}) > \lambda$, where λ is called the *surge threshold*. λ defines the min. increase in RTT_{min} required between two consecutive windows to flag an attack, and can be set to a constant (e.g., 10 ms) or a fraction of $max(RTT_{min})$ (e.g., 10%). A lower λ provides broader coverage but increases susceptibility

to false positives, and vice-versa. Users can adjust λ based on their desired trade-offs. We evaluate the false positive rate for different values of λ in Section 4.8.

4.5.4 Hardware-Amenable Changepoint Detection

We implement changepoint detection directly in switch hardware by combining the techniques described so far in an approach called the *two-window algorithm*. This involves tracking the per-prefix min. RTT (RTT^i_{min}) for each tumbling window i. Once the i^{th} window completes ($i > 0$), we compare RTT^{i-1}_{min} and RTT^i_{min} and mark the prefix as *attacked* if both the following *surge* conditions are met:

1. $RTT^{i-1}_{min} < \tau^*_{mid}$ and $RTT^i_{min} > \tau^*_{mid}$: The minRTT crosses the absolute threshold between two consecutive windows.

2. $RTT^i_{min} - RTT^{i-1}_{min} > \lambda$: The minimum RTT surges by at least the surge threshold in consecutive windows.

4.5.5 Adaptive Windowing and Speed of Detection

Since HiDe relies on the change in minRTT across windows of RTT samples, its detection speed depends on the following factors:

1. ***Traffic-related factor:*** The RTT sample rate per second (R), which is a function of the number of ACK packets per second, and thus of the data rate (packets per second) and TCP's cumulative ACKing behavior.

2. ***HiDe's detection parameters:*** In particular, the size of each time window (T), and the minimum number of RTT samples in a *valid* (defined below) window (S).

Only time windows containing at least S samples are considered valid and used for detection; windows with fewer samples do not necessarily benefit from min-filtering and could lead to false positives. In our dataset, we observe that S=5 works well and we use this value in our

experiments (Section 4.8). The sample rate R is determined by the traffic, while T and S are set based on information learned during the profiling phase. In an ideal scenario, RTT samples arrive at a roughly uniform rate and $R = (1/T) \cdot S = S/T$, meaning that samples divide neatly into time windows that are each just large enough to contain S samples. In this case, the *time-to-detection* is approximately $2T = 2(S/R)$. For example, if the data rate is 1 MB/s and the cumulative ACK frequency is one ACK per five data packets, the expected time-to-detection is about 71 ms.

In the wild, the rate at which RTT samples are generated is not necessarily uniform, so the time-to-detection can be slightly higher and may vary across profiled prefixes, but it generally remains within a small fraction of a second for active flows. To mitigate cases where S is effectively too high for the prevailing traffic rate (e.g., when the data rate during an attack is significantly lower than during profiling), we implement an *adaptive time window*. Under this strategy, for each window, HiDe starts with size T, but if S samples are not accumulated by the end of the window, it increases the window to $2T$, $3T$, and so on until at least S samples are reached. This ensures that HiDe always has valid windows to work with, and that the time-to-detection is as low as possible while still remaining accurate.

4.5.6 Minimizing Impact of False Positives

Despite reducing false positives, our detection algorithm is not foolproof and may occasionally generate them. To minimize their impact and to eliminate the need for human intervention, *HiDe* optionally employs an automatic *false positive correction* mechanism. When an attack is detected, *HiDe* blocks the affected prefix and simultaneously initiates active probing by sending ICMP echo packets to the most recently active IP address in the prefix at each time window. It monitors the corresponding RTT, and if the RTT falls below τ^*_{mid}, the prefix is unblocked, and detection resumes, minimizing disruption to regular operations. To limit probe traffic, *HiDe* reduces the probe rate to one per minute after five minutes of attempts. These parameters are user-adjustable for flexibility.

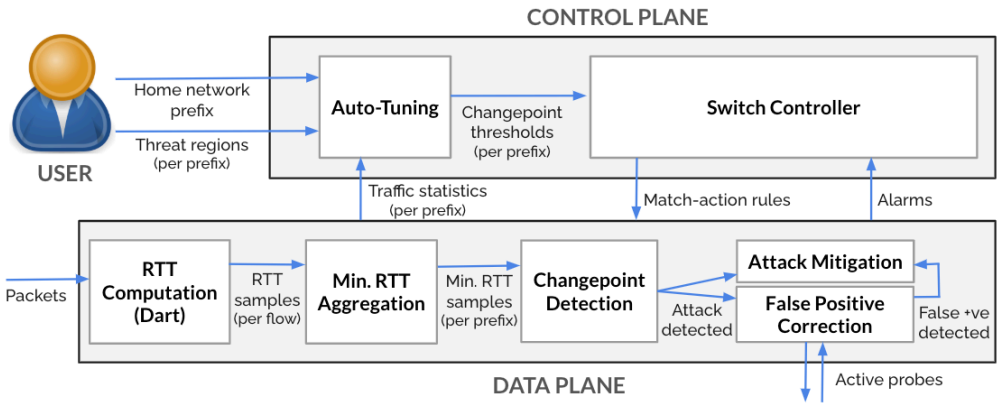


Figure 4.9: *HiDe* consists of a software control plane and a hardware data plane. The control plane auto-tunes per-prefix parameters for changepoint detection based on user inputs and traffic statistics from the data plane, and installs those parameters as match-action rules on the data plane. The data plane computes RTT samples, aggregates them by prefix, computes minRTT per window, and performs changepoint detection. Upon detecting an attack, the data plane blocks the corresponding prefix and triggers active probing to correct false positives.

4.6 *HiDe*: System

Figure 4.9 presents an overview of *HiDe*'s end-to-end workflow. *HiDe* comprises a control plane, implemented in software on a server, and a data plane, operating in high-speed hardware on a programmable switch. *HiDe* is deployed at the edge of a production network, and can observe all or most of its traffic depending on the network topology (Section 4.6.4).

4.6.1 Control Plane

User input. The user configures the control plane by providing the network prefix of their home network (source prefix) and specifying the threat regions (optionally, per prefix).

Auto-tuning. The auto-tuning component translates the user inputs and destination prefixes read from the data plane into corresponding geographic coordinates and computes the τ^*_{mid}. It also retrieves traffic statistics (specifically, $min(RTT_{min})$ and $max(RTT_{min})$) from the data plane for each prefix. Combining this information, the component identifies which prefixes can be effectively protected and generates changepoint detection parameters

for those prefixes, which it then sends to the switch controller. For prefixes that cannot be protected, it provides the user with a summary listing each prefix and their corresponding RTT_{min} statistics.

Switch controller. The switch controller translates the received parameters into corresponding match-action rules and installs them on the data plane. It also receives attack alarms from the data plane and (optionally) notifies the user.

4.6.2 Data Plane

RTT computation. We leverage DART, an existing system, to generate accurate RTT measurements per flow from all the traffic observed by the switch. DART achieves this at scale, handling a large number of flows without missing any RTT samples by efficiently managing switch resources [144].

Min. RTT aggregation. The next component aggregates RTT samples per destination prefix, breaks them down into time windows, and calculates the minimum RTT per window. Additionally, it computes traffic statistics like $min(RTT_{min})$ and $max(RTT_{min})$ per prefix to share with the control plane.

Changepoint. The data plane performs changepoint detection using our two-window algorithm, minRTTs per window per prefix, and parameters installed by the control plane.

Attack mitigation. When an attack is detected, the data plane rate limits or blocks the corresponding prefix—depending on operator configuration—and raises an alarm.

False positive correction. Optionally, if the operator seeks more confidence in the detection result, and HiDe is not being used in conjunction with control-plane-based approaches, it can craft and send active probes periodically to determine whether the detection was a false positive. If so, it unblocks the prefix.

4.6.3 Hardware Prototype

We implement our prototype in $P4_{16}$, and deploy it on the Intel Tofino2 high-speed programmable switch, which supports up to 12.8 Tbps of traffic at line rate [3, 23]. Our prototype does not depend on any specific features available on the Tofino, and can be ported readily to other programmable packet-processing hardware including other switches (e.g., Juniper Trio) and SmartNICs (e.g., NVIDIA BlueField3).

Switch control plane. The switch control plane installs per-prefix match-action rules specifying the window size (W), absolute threshold (τ^*_{mid}), and surge threshold (λ). If enabled, it listens on the CPU port for packets from the data plane containing information about either attack detections or non-coverage, and notifies the user. Additionally, it configures the Tofino's *packet generator* to send active probes during the *false positive correction* phase.

RTT computation. We leverage DART for continuous and accurate per-flow RTT computation [144], and utilize *packet mirroring*, a native feature that replicates packets, to enhance its functionality. First, the original packet is forwarded without added latency, with the mirrored copy used for RTT computation. Second, RTT samples generated by DART are passed to *HiDe*-specific data-plane components.

Per-prefix state. We maintain a *prefix table* in register memory to support changepoint detection. The table uses a *prefix signature*, derived by hashing the first 24 bits of the external IP, as the key. The stored values include the prefix's start timestamp, timestamp of most recent RTT, start timestamp of current window, number of RTT samples in current window, minRTTs for the current and previous windows, attack status, $max(RTT_{min})$, and $min(RTT_{min})$. The table accommodates up to 65,536 active prefixes, significantly exceeding the peak observed in our 12-hour campus trace (approx. 5K assuming a 5-second timeout), minimizing hash collisions. For collisions, we use *cuckoo hashing* [122]: the new prefix replaces the old one, which is loaded into memory, checked for timeout, and recirculated to a new index using a different hash seed if still valid. Each insertion allows up to 3 recirculations.

Resource Type	Compute RTT [144]	Track MinRTT	Detect Change	Mitigate Attack
Stages	7	2	4	3
TCAM	2.9%	0.0%	1.1%	0.0%
SRAM	4.5%	4.0%	2.4%	3.6%
Instructions	3.6%	2.4%	1.0%	1.1%
Hash Units	35.8%	12.5%	2.8%	5.6%
Input Crossbars	10.1%	3.0%	1.6%	1.9%

Table 4.1: Hardware resource usage of the Tofino2-based prototype, divided by functional component.

Changepoint detection. When an RTT sample is generated for a prefix, *HiDe* checks the status of the corresponding time window. If the window is not full, it updates the most recent timestamp, increments the RTT count, and replaces the current minimum RTT if the new sample is smaller. If the window is full, the current minimum RTT replaces the previous window's minimum, and $max(RTT_{min})$ and $min(RTT_{min})$ are updated. If the window is valid (i.e., it has enough samples), *HiDe* evaluates the surge conditions and, if those are satisfied, starts the mitigation process by blocking all non-ICMP packets from/to the prefix by adding it to a *block table*.

Active probing. Simultaneously, we start crafting and sending ICMP echo packets to the latest IP seen from the prefix and listening to responses to monitor its RTT. If a false positive is detected, the prefix is removed from the block table.

Resource usage. We analyze the resource usage of our prototype by function and find that its low resource consumption leaves ample resources for other concurrent switch functions (Table 4.1).

4.6.4 Deployment

HiDe is deployed at the edge of a production network (Figure 4.10), protecting clients within the network by monitoring the external leg of RTTs (*HiDe* to external hosts) rather than

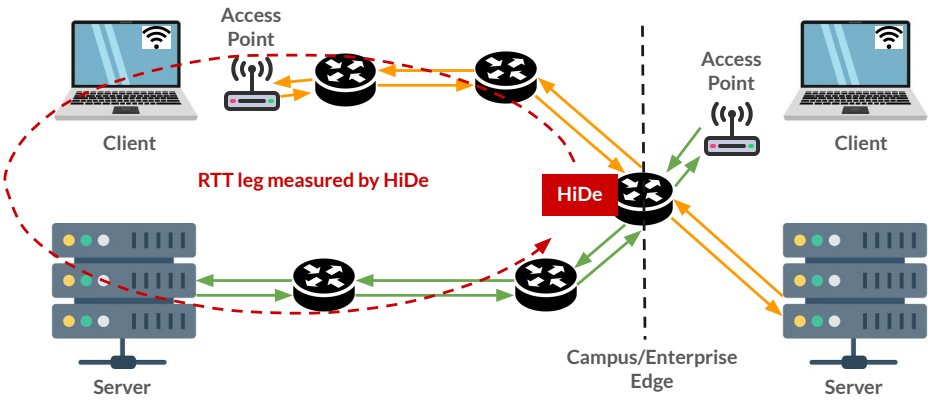


Figure 4.10: *HiDe*—deployed at the edge of a production network—defends servers (associated with vulnerable prefixes) and clients (optionally since associated with less vulnerable prefixes) inside it by measuring the *external leg* of RTT from itself to external hosts.

the internal leg (*HiDe* to internal hosts) [144]. We denote connections with clients inside the defended network as *Client-In-Server-Out* (CISO) and connections with servers inside the network as *Server-In-Client-Out* (SICO). We observe that the primary source of noise in RTTs is typically the access link near the client. For CISO connections, which are associated with the vulnerable prefixes (described earlier in Section 4.5.3), the access link is part of the internal leg and does not affect the monitored RTTs, resulting in less noise. In contrast, SICO connections experience higher noise levels, as the access link is external. In our campus data, we apply a TCP port number-based heuristic to distinguish CISO connections from SICO connections: if the port number used by the campus-internal host is < 1024 and the one used by the external host is > 1024, we consider it a SICO connection, and vice-versa.

4.7 Experimental Setup

We outline our experimental setup: first, to demonstrate live detection of ethically launched interception attacks on controlled *iperf* traffic; second, to collect a 12-hour campus trace highlighting *HiDe*'s low false positive rate and minimal impact on regular operations.

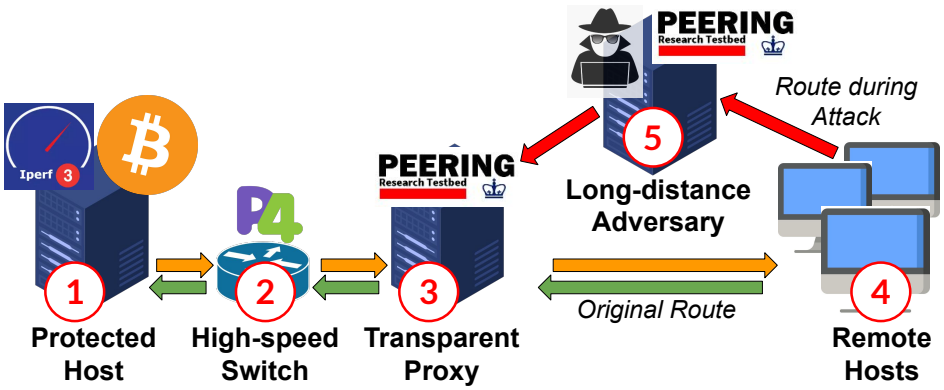


Figure 4.11: Experimental setup for our live experiments. The orange and green arrows indicate the original *protected host* to *remote hosts* route and back, respectively. The return path (green) is intercepted by the *long-distance adversary*—the diverted portion of the route is shown with red arrows.

4.7.1 Passive Capture of Campus Traffic

Data collection. As outlined before, we captured 12 hours of production traffic—covering 1 pm to 1 am local time to include global working hours—at the edge of our US-based campus network using a TAP device near the gateway router. Packets—only TCP headers, anonymized at source in a prefix-preserved manner—from selected subnets were mirrored and recorded on a collection server with *tcpdump*.

Dataset overview. The dataset comprises 1.1TB of trace data representing 5.32TB of packet bytes, encompassing 19 billion packets, 7.5 million flows, and 238 million RTT samples. It includes 12K unique internal IPs and 324K unique external IPs, distributed across 183K external prefixes, 23.2K of which are based in the US.

4.7.2 Live Experiments

Figure 4.11 shows the experimental setup for our live experiments involving active iperf3 traffic.

Deploying *HiDe* to protect experimental traffic. We set up our experiment using three key components: (1) a host on our campus running iperf3, (2) a high-speed

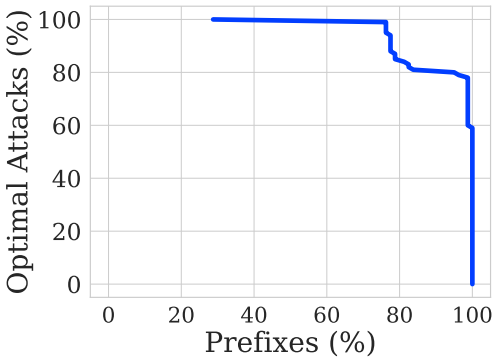
programmable switch on campus where *HiDe* is deployed to monitor traffic, and (3) a transparent TCP proxy on an Amazon AWS instance. The proxy, which doubles as a *PEERING* node, advertises a /24 prefix allocated to our experiment. PEERING provides distributed ASes for controlled, real BGP announcements [132]. We use one IP address from the /24 pool, applying *iptables* rules on components 1 and 3 to masquerade it as the application's IP. This setup enables *HiDe* to monitor all experimental traffic while allowing external adversaries to ethically launch BGP interception attacks on the /24 prefix.

Setting up remote hosts. We deploy AWS instances in geographically diverse locations, including the US east and west coasts, Europe, and Asia. The *iperf3* server runs on the protected host across multiple ports, while clients on the distributed AWS instances connect to the PEERING IP of the transparent proxy, which forwards traffic to the server. The remote hosts are indicated as component 4.

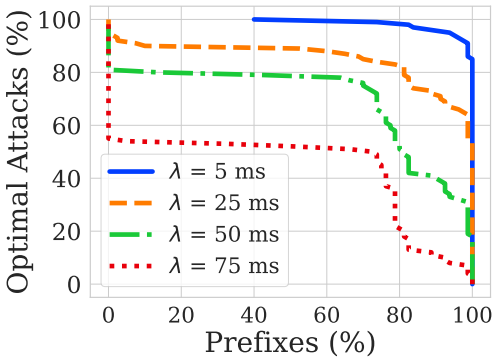
Launching ethical routing attacks. The final step in our setup is launching ethical BGP interception attacks on the PEERING IP. We designate the PEERING node in Amsterdam as the attacker (component 5) and implement a stealthy interception attack using the technique by Birge-Lee et al., which employs BGP communities to control the blast radius of the attack [16]. The attacker advertises the same /24 prefix as the transparent proxy (an *equally-specific* attack), redirecting traffic from nearby nodes to Amsterdam. The attacker then forwards the intercepted traffic to the transparent proxy, leaving both sender and receiver unaware of the attack.

4.8 Evaluation

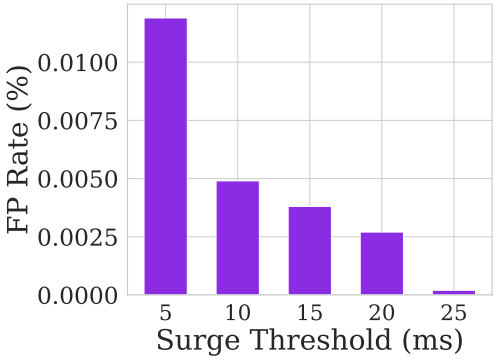
In this section, we present our evaluation results. In the first part (Section 4.8.1), we run simulations of *HiDe* on our campus dataset and report coverage, false positive rate, and downtime due to false positives. In the second part (Section 4.8.2), we present results from



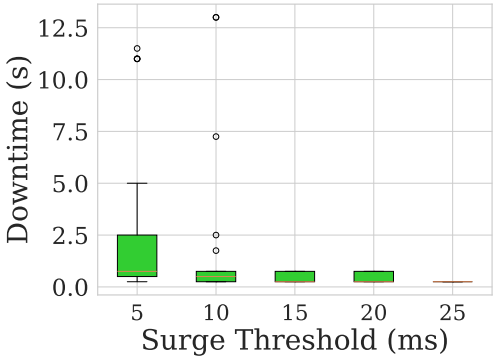
(a) Coverage (lower bound)



(b) Coverage (defendability)



(c) False positive rate



(d) Downtime

Figure 4.12: Faithful simulation on campus data illustrates that *HiDe* can defend most prefixes from optimal attacks from most countries, incurs low false positives ($\leq$0.012%) and low downtime due to false positives (median downtime $\leq$0.75s).

live experiments where the *HiDe* prototype defends the protected host (in Figure 4.11) when a subset of connections are impacted by an ethically conducted interception attack.

4.8.1 Trace-Based Evaluation

We evaluate *HiDe* using three metrics: (1) False positive rate or FPR (measures reliability/usability/practicality), (2) Coverage (measures the trade-off with low FNR and FPR), and (3) Downtime (measures impact of false positives on regular operation). We perform this evaluation using a faithful simulation of *HiDe* written in Python on real latency data obtained from production traffic on our campus (Section 4.7.1). We operate with the goal

of protecting US-based prefixes from long-distance interception attacks from the mainlands of other countries in our dataset. For each prefix, we divide into two equal parts the total time during which the prefix was active: the first half is used for *profiling* while the second half is used for *detection*.

Vulnerable prefixes. In accordance with *HiDe*'s coverage strategy, we only defend vulnerable (CISO) prefixes, i.e., external prefixes associated with a server. As described earlier, we identify such prefixes using a TCP port number-based heuristic. 16.8K US-based external prefixes match this condition in our campus dataset.

Profiled prefixes. From external server prefixes, we further select those that were active for at least 10 minutes out of 12 hours (so we profile on at least 5 mins of data). We determine this by dividing the 12-hour period into buckets of 1 min, and checking which prefixes generated an RTT sample in at least 10 such buckets. 6K US-based external server prefixes are retained after this step. In a real network, the operator could profile prefixes for as long as needed to ensure it covers typical variation of RTT during benign operation—without any excess overhead since the profiling happens directly in the switch. For the following analysis, we make the assumption that the campus data captured during the 12-hour period did not experience any long-distance interception attacks (i.e., no true positives were present), so if *HiDe* detects a prefix it must be a false positive. We report our results based on optimal attacks from all 257 non-US threat countries.

Coverage impact of theoretical lower bound. Some US-based prefixes are not defendable against certain threat regions because during the profiling phase, they exhibit a $min(RTT_{min})$ larger than the corresponding τ^*_{mid} (i.e., measured delay without diversion is always higher than the lower bound with diversion). This could be because the threat region is geographically too close or because the network is always congested. Figure 4.12a shows that, based on this condition, *HiDe* can cover 99% prefixes against 78% attacks and 75% prefixes against 99% attacks. The covered prefix-threat country pairs are considered in

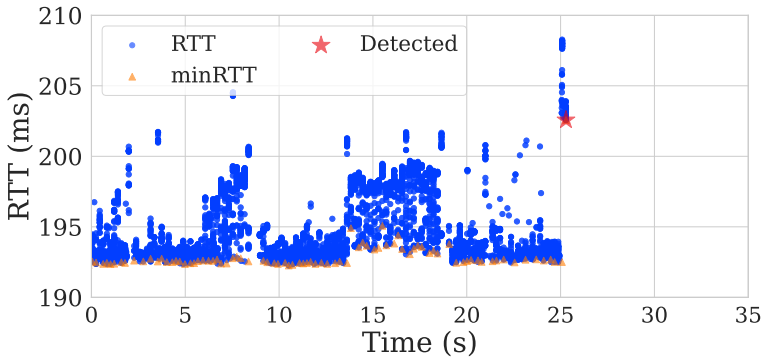


Figure 4.13: *HiDe* (immediately) detects interception attacks ethically launched by us on iperf3 traffic.

subsequent experiments.

Coverage impact of defendability analysis. For different values of the surge threshold (λ), Figure 4.12b shows *HiDe*'s coverage based on defendability—i.e., whether the mid-attack lower bound RTT clears the pre-attack $max(RTT_{min})$ by at least λ ms. At $\lambda = 5$ ms, 25 ms, 50 ms, and 75 ms respectively, *HiDe* can cover 99% prefixes against 91%, 64%, 31%, and 7% attacks, respectively. This illustrates the trade-off between surge threshold and coverage.

False positive rate. By focusing on defendable prefixes, we achieve a false positive rate of approximately 0.012% at worst, as shown in Figure 4.12c. For higher surge thresholds (i.e., 30 ms), it drops to zero.

Downtime due to false positives. We estimate the likely downtime from a false positive by measuring how long (in multiples of time window size) it takes for the minRTT to return to normal for a falsely detected prefix. Since our (optional) active probing sends one probe per window, we expect similar results in reality. The median downtime is only 0.75 seconds.

4.8.2 Live Interception Attack Detection

In this section, we evaluate HiDe on a *live* interception attack to demonstrate the *fidelity* of our hardware prototype, i.e., that HiDe's in-switch RTT monitoring and changepoint

detection behave as intended under real network conditions. This case study is not intended to establish that HiDe detects interception attacks in a fully general setting or to characterize coverage across all geographies. Instead, we focus on a long-distance scenario in which the adversary is sufficiently far away to induce a large RTT increase, yielding a clear delay signal for validating end-to-end correctness. Sections 3 and 8.1 analyze HiDe's coverage more broadly, including settings where the detour is geographically short and the RTT change may be too small to detect reliably.

Setup. We run the iperf3 server on our campus and the transparent proxy in Ireland, who forwards all traffic it receives to our campus via our prototype. The iperf3 clients are in Virginia (2 flows from the same prefix), Ohio, and Mumbai. The prefix in Virginia is hijacked from Amsterdam, causing Ireland to send traffic to Amsterdam instead of Virginia. Amsterdam then forwards the traffic to Virginia. The traffic takes the following round-trip route before the attack: *Virginia (via PEERING infrastructure) to Ireland to our campus to Ireland to Virginia (via PEERING infrastructure)* and the following one during the attack: Virginia (via PEERING infrastructure) to Ireland to our campus to Ireland to Amsterdam (via. PEERING infrastructure) to Virginia (via PEERING). Due to limitations of where we can deploy a Tofino switch on live traffic and a lack of diversity in the PEERING topology, we are restricted to this complex setup. The attack takes effect at 25 seconds, as can be observed from the abrupt rise in RTT (blue dots) in Figure 4.13.

Interception attack detection. Using multiple runs of traceroute, we estimate the lower bound of RTT as approx. 190.5 ms before attack and 199 ms during attack (absolute threshold). We set the window size to 250 ms and the surge threshold to 5 ms. Based on the minimum RTTs (orange triangles), we detect the attack in 500 ms (red star).

4.9 Related Work

Prior work on defending against BGP hijacks broadly falls into *proactive* approaches that aim to prevent hijacks from succeeding in the first place, and *reactive* approaches that detect and mitigate attacks after they begin. Reactive defenses can be further divided into control-plane-based, data-plane-based (using active or passive measurements), and hybrid designs that combine both. In this paragraph, we summarize these lines of work and highlight how HiDe fits into, and complements, this landscape.

4.9.1 Proactive Approaches

A long line of prior work seeks to proactively secure interdomain routing by preventing invalid routes from being accepted in the first place. One direction is cryptographic upgrades to BGP, such as BGPsec, which provide path validation but require widespread deployment and operational support to be effective [10]. Another direction is origin authentication and filtering via RPKI and route-origin validation, which can prevent a subset of attacks (e.g., forged-origin hijacks) but does not address many other attack classes (e.g., interception or path manipulation) [25]. Clean-slate architectures such as SCION provide stronger security properties by design, but similarly face the barrier of incremental deployability at Internet scale [124]. These proactive mechanisms can be highly effective where deployed, but their protection is limited by adoption and the subset of attacks they cover. HiDe is complementary: it is a reactive defense that does not rely on broad Internet-wide adoption, and is therefore readily deployable in today's ecosystem.

4.9.2 Reactive Control-Plane-Based Detection and Mitigation

Reactive defenses commonly analyze BGP updates from public monitors (e.g., RIS and RouteViews), private commercially-operated monitors, and/or an operator's own routers to detect suspicious announcements and trigger mitigations such as prefix deaggregation

(announcing more-specific routes) or filter-based actions. Systems such as ARTEMIS and DFOH illustrate this model, emphasizing fast identification of suspicious announcements and automated mitigation [66, 145]. A key limitation of this class is *visibility*: detection quality depends on what monitors can see and when they see it. Prior work shows that attackers can craft targeted hijacks that restrict propagation toward monitors and limit global visibility, including attacks that manipulate BGP attributes to achieve *stealthy, targeted* interception [14, 107, 181]. A concrete example is the manipulation of BGP communities to surgically steer traffic while limiting visibility at monitors [16].

Even when attacks are eventually detected, detection and mitigation can still be *slow*: updates may take time to reach monitors, for detectors to accumulate sufficient evidence, and for mitigation to kick in, increasing the exposure window during which damage can occur. In fact, meaningful harm can occur within seconds of a hijack taking effect: early-stage website fingerprinting based on statistical properties can succeed using only the first few packets (20 in one study, 100 in another), bursts (3 in one study) or a small fraction (22% in one study) of a page load [45, 148, 151]. Attackers can also mirror and store diverted traffic for offline analysis and potentially cause further harm by, for example, trying to break encryption of payloads. Real incidents likewise show that interception can be monetized immediately; for example, in the 2018 hijack involving Amazon's DNS infrastructure, diverted cryptocurrency traffic enabled attackers to collect credentials and later steal cryptocurrency [125]. These results motivate complementary signals—such as RTT in the data plane (leveraged by HiDe)—that remain observable at the victim even when control-plane visibility is incomplete or slow.

4.9.3 Reactive Data-Plane-Based Detection using Active Measurements.

An alternative is to detect hijacks using *active probing* in the data plane (e.g., ping, traceroute, nmap) to infer unexpected path changes. iSPY is a representative approach that uses

active probing to identify suspicious routing behavior [185]. Defenses based on active measurements can provide direct evidence of path changes and can sometimes localize anomalies, but they come with practical constraints: they require responsive targets, introduce probe traffic overhead that grows with scope, and often require sustained probing to maintain accuracy. These factors complicate always-on deployment at large scale and can limit coverage for unresponsive or rate-limited destinations. HiDe instead relies primarily on *passive* monitoring of RTT, avoiding probing overhead in the common case; it (optionally) uses active probing only as a targeted follow-up to correct potential false positives during mitigation.

4.9.4 Reactive Data-Plane-Based Detection using Passive Measurements

A smaller body of work leverages passively observed performance signals to flag routing anomalies. For example, Hiran et al. use crowd-sourced RTT measurements to detect routing anomalies [64], though their method only addresses detection, not mitigation. Oscilloscope [24] offers advanced hijack detection but relies on emulated data, suffers from high false-positive and false-negative rates, and lacks a hardware implementation, limiting its applicability and scalability. HiDe differs from these approaches in their operational goals and deployability: HiDe is a *real-time* defense for long-distance interception attacks, and is designed for *always-on, line-rate* operation on programmable switches.

4.9.5 Hybrid Approaches Combining Control-Plane and Data-Plane Signals

Hybrid defenses combine BGP-derived signals with data-plane observations to improve detection confidence and reduce false positives. Argus and HEAP exemplify this approach by correlating control-plane anomalies with data-plane signals to strengthen detection [131, 149]. In general, hybrid methods inherit benefits and limitations from both sides: they can be more

robust than either signal alone, but still face challenges with visibility at monitors, and measurement overhead (therefore, scalability) of active probing. HiDe is compatible with the hybrid model *and* strengthens it significantly by providing a fast data-plane signal that does not incur probing overhead and does not require responsive destination IPs.

4.10 Conclusion

We present *HiDe*, a system to detect and mitigate long-distance BGP interception attacks—where an adversary in another country diverts traffic through its own infrastructure to eavesdrop before forwarding it to the victim. By leveraging RTT measurements that attackers cannot conceal, *HiDe* delivers high-accuracy defense at line rate on a Tbps-scale programmable switch. Our analysis of worst-case attacks across 258 countries confirms its effectiveness, and we validate *HiDe*'s fidelity and effectiveness through experiments with anonymized campus traces and ethically conducted real-world hijacks, achieving robust mitigation with low false-positive rates.

Chapter 5

Enabling Passive Measurement of Zoom in Production Networks

Video-conferencing applications impose high loads and stringent performance requirements on the network. To better understand and manage these applications, we need effective ways to measure performance in the wild. For example, these measurements would help network operators in capacity planning, troubleshooting, and setting QoS policies. Unfortunately, large-scale measurements of production networks cannot rely on end-host cooperation, and an in-depth analysis of packet traces requires knowledge of the header formats. Zoom is one of the most sophisticated and popular applications, but it uses a proprietary network protocol. In this chapter, we demystify how Zoom works at the packet level, and design techniques for analyzing Zoom performance from packet traces. We conduct systematic controlled experiments to discover the relevant unencrypted fields in Zoom packets, as well as how to group streams into meetings and how to identify peer-to-peer meetings. We show how to use the header fields to compute metrics like media bit rates, frame sizes and rates, and latency and jitter, and demonstrate the value of these fine-grained metrics on a 12-hour trace of Zoom traffic on our campus network.

5.1 Introduction

Video-conferencing applications have seen an unprecedented surge in popularity over the past few years [29, 52]. Zoom has been at the forefront of this phenomenon, with adoption by many organizations to foster teaching, meetings, and presentations during the COVID-19 pandemic [34, 51, 118].

To keep pace with ever-stringent user performance expectations [54, 184] and increasing resource contention, practitioners and researchers need the ability to measure (and improve) Zoom performance in the wild, *without requiring cooperation from end hosts*. For instance, granting this capability to network operators would enable more targeted capacity planning, problem troubleshooting, and traffic-prioritization policies. Realizing this requires the ability to extract metrics such as media bit rates, delay, frame rates, and frame-level jitter solely via analysis of packet captures of Zoom sessions. These insights, in turn, grant a clear understanding of the inner-workings of a Zoom meeting, and the performance and quality experienced by each of its participants. Taken together, we require *fine-grained* measurements and performance insights for Zoom derived from analyzing passively-collected network traffic *in the wild*.

Unfortunately, existing measurement approaches all fall short of at least one of these goals. Some researchers instrument end hosts to run controlled experiments that study Zoom's rate adaptation and performance [22, 29, 86, 93]. However, controlled experiments are labor-intensive, limited in scope, and do not reveal Zoom performance in the wild. Other researchers conduct measurement studies on production networks [34, 118]. Due to Zoom's proprietary network protocol, these studies collect only coarse-grained statistics such as byte and packet rates, which are insufficient for the use cases outlined above. Lastly, while some performance metrics are available to operators through Zoom's API, this data is also coarse-grained (i.e., not packet level) and measured far from the operator's network (i.e., in Zoom's data centers); consequently, this API data is insufficient for rapid adaptation at on-premise network devices.

In this chapter, we address this void, and enable direct (and systematic) measurements of Zoom by (1) demystifying how Zoom works at the packet level and (2) designing tools and techniques for analyzing Zoom performance from packet traces. The key challenges are twofold. First, Zoom uses a proprietary packet format, encrypted control and media traffic, and closed-source client software [34, 86, 95, 118]. As a result, Zoom cannot be analyzed easily using diagnostic tools for WebRTC [17] or packet-analysis tools like Wireshark [172]. Second, Zoom employs complex control logic whereby both user behavior (e.g., muting audio/video, changing the display configuration, sharing the screen, enabling recording) and network behavior (e.g., packet loss, delay, and throughput) can result in similar changes in application behavior.

To grow our understanding and overcome these hurdles, we first use controlled experiments (solely) to decipher Zoom's protocols, discovering (1) the relevant unencrypted fields in the Zoom packet format, (2) how to group streams into meetings, and (3) how to identify peer-to-peer meetings. We make our packet-parsing logic available as a Wireshark plugin for the community to use. Perhaps surprisingly, we find that valuable information can be gleaned from the unencrypted parts of Zoom traffic. We expect these unencrypted parts of Zoom packets to remain in the clear, since Zoom itself requires these for scalable forwarding of media traffic.

Armed with this knowledge, we show how to estimate Zoom meeting performance from in-network measurements at fine granularity, supporting metrics such as media bit rates, packet latency, jitter, retransmission, and more. To validate our methodology and ability to collect fine-grained performance insights without end-host modifications, we apply our passive measurement techniques to Zoom traffic collected on our campus network. Our study used a scalable Zoom traffic capture system we built in the data plane (using P4 [20] on Intel Tofino switches [3, 68]) to filter and anonymize Zoom packets for analysis on servers.

While our analysis and study focuses on Zoom, many of our techniques should generalize across video-conferencing applications that use the Real-Time Protocol (RTP) which is the

vast majority of such applications, including Google Meet and Microsoft Teams [118]. We describe in detail how we deciphered Zoom's protocols and provide a blueprint for future studies on proprietary network protocols. Finally, the core of our work is the analysis of Zoom and the development of measurement techniques; as such, the measurements reported only serve to demonstrate and validate the accuracy and granularity of our techniques. We leave a full measurement study of Zoom performance in the wild as future work.

Taken together, our chapter makes the following contributions:

1. We demystify the most relevant parts of Zoom's proprietary protocols and shed new light on how Zoom delivers media streams over the network.

2. We demonstrate how understanding Zoom's protocol enables us to extract a wider range of metrics from Zoom network traffic, including packet loss, latency, jitter, media bit rates, frame rate, and frame sizes.

3. We make our artifacts — the Wireshark plugin, the P4 traffic-capture program, and the software-based analysis tools — available to the community [104].

5.2 Video Conferencing Background

Video-conferencing applications (VCAs) are complex distributed systems with many components. Design and implementation choices that must be made when building a VCA fall into three main categories which we will now outline.

Network Protocols and Mechanisms. Most major conferencing applications use the Real Time Protocol (RTP) [137] and its counterpart, the Real Time Control Protocol (RTCP) [137], to transport media. An RTP stream is identified by a unique identifier, the Synchronization Source Identifier (SSRC), which can be used to multiplex several media streams (e.g., audio and video) over a single UDP flow. Each stream can contain several sub-streams to, for example, carry forward error correction (FEC) data. While

RTP headers are usually transmitted in cleartext to allow conferencing servers to forward or modify streams without having to decrypt them, RTP payload is mostly encrypted (see SRTP [119]). Before media can flow between participants, clients negotiate through a signaling mechanism what codecs to use, how to encapsulate media, and where to send the respective streams. The Session Initiation Protocol (SIP) [134] and the Session Description Protocol (SDP) [123] are commonly used for this.

Media Encoding and Rate Adaptation. Over the past years, audio and video codecs (e.g., Opus [163] and H.264/AVC [171]) have become increasingly efficient; to achieve high quality at low bandwidth consumption, video codecs leverage complex intra- and inter-frame prediction schemes [154, 155]. A feature of modern video codecs particularly relevant to video conferencing is Scalable Video Coding (SVC). A scalable video stream contains several layers corresponding to different quality levels where higher-quality layers build upon lower-quality layers. SVC allows the conferencing server to remove higher-quality parts of a video bit stream to rapidly adapt to, for example, varying network conditions or device capabilities, for each participant individually [139].

System Architecture and Topology. Video-conferencing applications may choose different architectures for interconnecting meeting participants. Direct peer-to-peer (P2P) connections between meeting participants result in the lowest latency between two given clients but can be challenging to realize in the presence of firewalls and network address translators (NAT) [11, 53, 76]. Additionally, P2P meetings with many participants become infeasible due to the quadratically growing number of streams between the clients. As a result, most VCAs use P2P connections only for two-party calls (e.g., Zoom) or not at all (e.g., Google Meet) [118]. The alternative is to use an intermediate conferencing server that either transcodes incoming streams to a single outgoing stream per participant or one that selectively replicates and forwards media streams among clients. The server in the latter approach is called a Selective Forwarding Unit (SFU). SFUs enable the use of SVC and

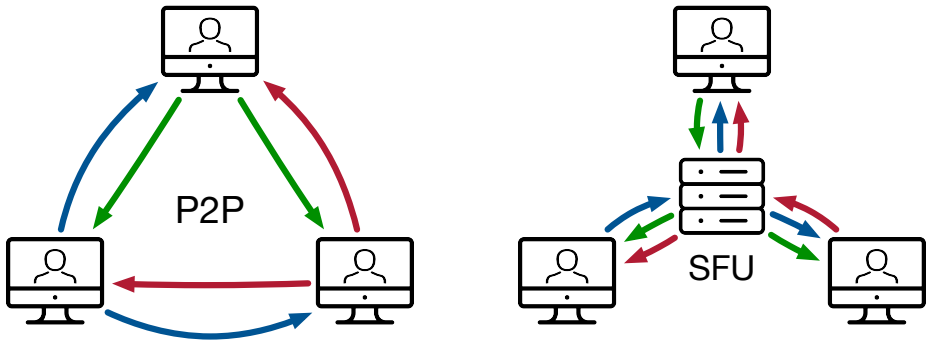


Figure 5.1: Video conferencing architectures: Peer-to-peer (P2P) vs. selective forwarding unit (SFU). Each color represents a participant's video stream.

are the de-facto standard in video conferencing today. Figure 6.1 illustrates the difference between a P2P meeting and one using an SFU.

Finally, as the space of possible implementation choices is vast, WebRTC provides a common framework for implementing real-time collaboration and communication systems [17]. WebRTC provides several APIs, means to coordinate the use of the various protocols, a standard congestion control algorithm (Google CC [27]), and a common set of media codecs.

5.3 What is (Not) Known about Zoom

Prior studies have reported on the flow-level structure of Zoom meetings [29, 34, 86, 118] and shed light on Zoom's rate-adaptation algorithm through controlled experiments [29, 86]. Additionally, Zoom provides high-level documentation about some internals of their system [190, 194, 197]. In contrast to prior results, we are interested in extracting finer-grained, continuous performance metrics, such as frame rates and frame-level jitter, from large-scale passive packet captures. We will now give a brief overview of prior findings, and outline why they fall short in enabling our target use cases for operators and researchers outlined in Section 5.1.

Server Traffic. Zoom publishes the list of IP subnets they use [197] to aid operators with firewall configuration. Prior works have used this list to filter traffic to and from Zoom

servers and analyzed the resulting traffic from a client's perspective. The network traffic observable during a Zoom call consists of several TCP connections to various Zoom servers and between one and three UDP flows to a single Zoom server. The TCP connections use server port 443 and carry TLS-encrypted data [34] which is presumably control traffic. The UDP flows carry the actual media traffic; there is always one flow per media type in use (audio, video, or screen sharing), regardless of the number of participants in the meeting. Prior works have confirmed this by enabling and disabling audio, video, and screen sharing during a meeting and observing the respective flows appear or disappear in their network trace [29, 34, 86, 118]. The media flows use ephemeral port numbers at the client and port 8801 at the server [29, 34].

P2P Traffic. On top of connections to Zoom servers, prior studies have observed that Zoom media traffic switches to a direct, peer-to-peer (P2P) connection between participants for meetings with exactly two participants. In this case, all three media types are sent over the same UDP flow which uses ephemeral port numbers at both peers. When the meeting media traffic switches from being sent through a server to P2P, the new single UDP media flow starts with a new port number [29, 34, 86, 118]. Meetings with a single participant (before any others have joined) start transmitting to a server. Once the second participant joins, the Zoom client then sometimes establishes the direct P2P connection within tens of seconds. As soon as a third participant joins, the meeting reverts back to using a server where it then stays even if the number of participants goes back to two [34]. No prior work has been able to deterministically detect and filter these P2P connections due to the unknown peer IP addresses and ephemeral port numbers, as opposed to the public IP and port information published for Zoom servers.

Header Format. Only Nisticò et al. [118] went beyond flow-level characteristics of Zoom's network traffic by reporting that Zoom uses RTP embedded in a custom, undocumented four-byte header. They observed two different values in this header. Their study does not

go further than this and their publicly available tool to extract RTP headers from Zoom traffic [111] (from May 2020) was not able to extract any headers from our traces collected during 2021 and 2022. We further discuss this in Section 5.4.2.

Rate Adaptation. Finally, prior works on Zoom's session quality and rate adaptation [29, 86] used controlled experiments where they injected a video stream at one side of a two-party meeting and compared a capture at the receiver with the original video. Lee et al. [86] studied Zoom's rate-adaptation algorithm by monitoring its bandwidth use and frame rate (using a timestamp in the source video) while injecting cross-traffic. They find that Zoom media streams adapt to network congestion primarily by adjusting the sender's bit- and frame rate, as opposed to adjusting the stream at the SFU. Furthermore, the paper reports that Zoom uses jitter as opposed to absolute delay for rate adaptation. Chang et al. go further by also comparing the Structural Similarity Index (SSIM) [164] of the sent and received videos to quantify picture quality degradation. The authors find that Zoom's video encoding is sensitive to the type of video with high-motion videos significantly reducing the received video quality. It has been reported that Zoom uses SVC over AVC for rate adaptation and scaling of video streams [65].

Taken together, all prior studies on Zoom report operational details at a *flow level*, e.g., overall packet and data rates of Zoom traffic. Unfortunately, this level of detail is not sufficient to obtain deeper understandings of the performance of a Zoom meeting. For example, previous approaches are unable to deterministically differentiate audio from video packets, quantify packet loss, or infer frame rates or latency from Zoom traffic. Attributes and metrics like these are (at a minimum) required to estimate if an ongoing meeting suffers from poor quality as experienced through, for instance, low frame rates, poor video resolution, or audio lag. In Section 5.4, we outline our approach to understand Zoom's network protocols in more detail and extract such *packet-level* metrics.

5.4 Demystifying Zoom's Protocols

We identified three main questions that must be answered to understand Zoom's network protocols in detail and to estimate Zoom performance:

1. How do we reliably detect all Zoom traffic (including P2P connections) in a network? (Section 5.4.1)

2. What is Zoom's header format and what information can be extracted from individual packets? (Section 5.4.2)

3. How do we group packets belonging to the same meeting together to make meeting-wide inferences? (Section 5.4.3)

We developed a set of methodologies to dissect and analyze the captured traffic and use these methods on data gathered from controlled experiments to answer the questions above step-by-step. In contrast to earlier work, we do not use our controlled experiments to draw conclusions about Zoom performance or rate adaptation specifics, rather, we use them to understand enough about Zoom's protocols to enable large-scale passive measurement studies in the future. Even though only applied to Zoom traffic in this chapter, we believe that our approach is applicable to studying other proprietary, black-box protocols (see Section 6.9).

5.4.1 P2P Connection Detection

While previous work has reported that Zoom uses P2P connections for two-participant meetings [29, 34, 86], no prior work has been able to deterministically detect this traffic due to the use of ephemeral port numbers at both ends and the fact that the IP addresses belong to the clients and are not publicly known (in contrast to Zoom's server addresses).

We observed that before any P2P connection is established, each client exchanges *Session Traversal Utilities for NAT (STUN)* [97] packets with a Zoom server. This exchange determines if a P2P connection is possible and always uses UDP port 3478 (the well-known

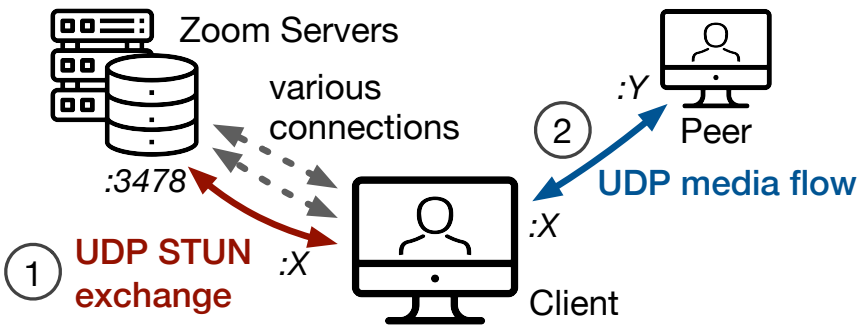


Figure 5.2: Connection establishment in a P2P meeting.

port for STUN) on the server side and the ephemeral port which is later used for the P2P connection on the client side. The packets are a series of STUN binding requests and are transmitted in cleartext. Figure 5.2 illustrates this process with *:X* denoting the UDP port number that is later used for the media flow.

These observations about the P2P connection establishment process allow us to reliably capture not only server-based, but also P2P Zoom traffic within and leaving our campus. To do so, we observe STUN packet exchanges with Zoom servers, store the IP address and ephemeral port number used at the client together with the time of the STUN exchange. If the same client then uses this port number within a configurable timeout again to communicate with another IP address, we treat this traffic as a Zoom P2P media flow. While this method, depending on the timeout used, can lead to false positives due to port reuse, in all traffic that we collected, all resulting P2P flows did actually contain Zoom media traffic. Even if false positives are collected, they can easily be filtered out by inspecting the packet format (see Section 5.4.2).

5.4.2 Entropy-Based Header Analysis

Previous work [118] has reported that Zoom uses RTP within a custom four-byte header but does not go further than this (see Section 5.3). The Zoom traffic that we captured in late 2021 and early 2022 uses a different packet format than the one described in previous work

and, as a result, we could not reproduce the prior findings. We assume that Zoom changed the used header format since the publication of this particular work in 2020.

Our findings are also vulnerable to becoming (partially) invalid if Zoom's protocols change. Many of our techniques, however, are based off widely-used protocols (e.g., RTP, RTCP, and STUN) and we do not expect Zoom to dramatically change the set of standard protocols employed. Of course, the way Zoom uses and encapsulates them may change, which would require slight modifications to our application logic. For this reason, we share in detail our methodology of analyzing Zoom's header format; this methodology can be repeated if the format changes.

5.4.2.1 Finding Unencrypted Header Fields in Zoom Traffic

While Zoom claims that all media traffic is encrypted [193], it does not specify at what granularity encryption is applied and what information may be transmitted in the clear. To find out whether all UDP payload is encrypted or not, we wrote a program that extracts the (binary) values of 8, 16, and 32-bit blocks at various offsets from the beginning of the UDP payload across all packets within a UDP flow. For example, for four bytes of payload, this results in four 8-bit, two 16-bit, and one 32-bit value sequences. We then plotted each such sequence with the packet index on the x-axis and the respective byte range's values on the y-axis to see if the values appear to be uniformly distributed and show maximum entropy, as expected in encrypted data. This approach, illustrated in Figure 5.3, allowed us to quickly and visually inspect Zoom's protocol. Oftentimes, after finding parts of the payload that does not seem to be random (or encrypted), we manually adjusted the block sizes and offsets of the value sequences extracted to further inspect the protocol fields.

We automatically generated hundreds of such plots; they show three different types of value distributions. For illustration, Figure 5.4 depicts (fabricated) examples of these distribution types: values are either entirely randomly distributed (red dots), follow horizontal lines (green squares), or follow angled lines (blue triangles). Several such lines, often with dif-

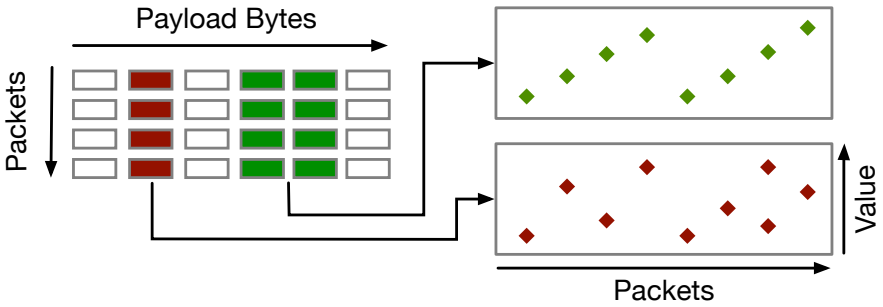


Figure 5.3: Entropy-based packet header analysis.

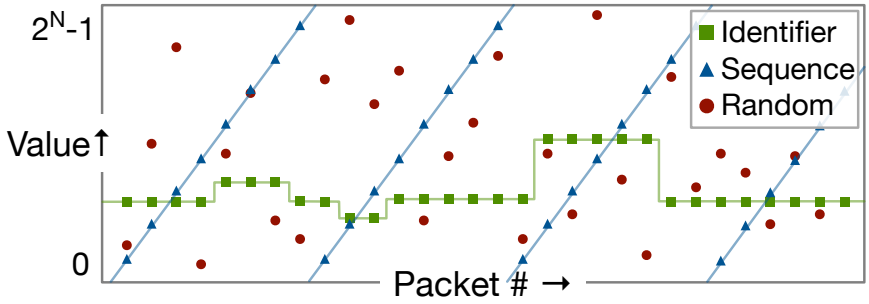
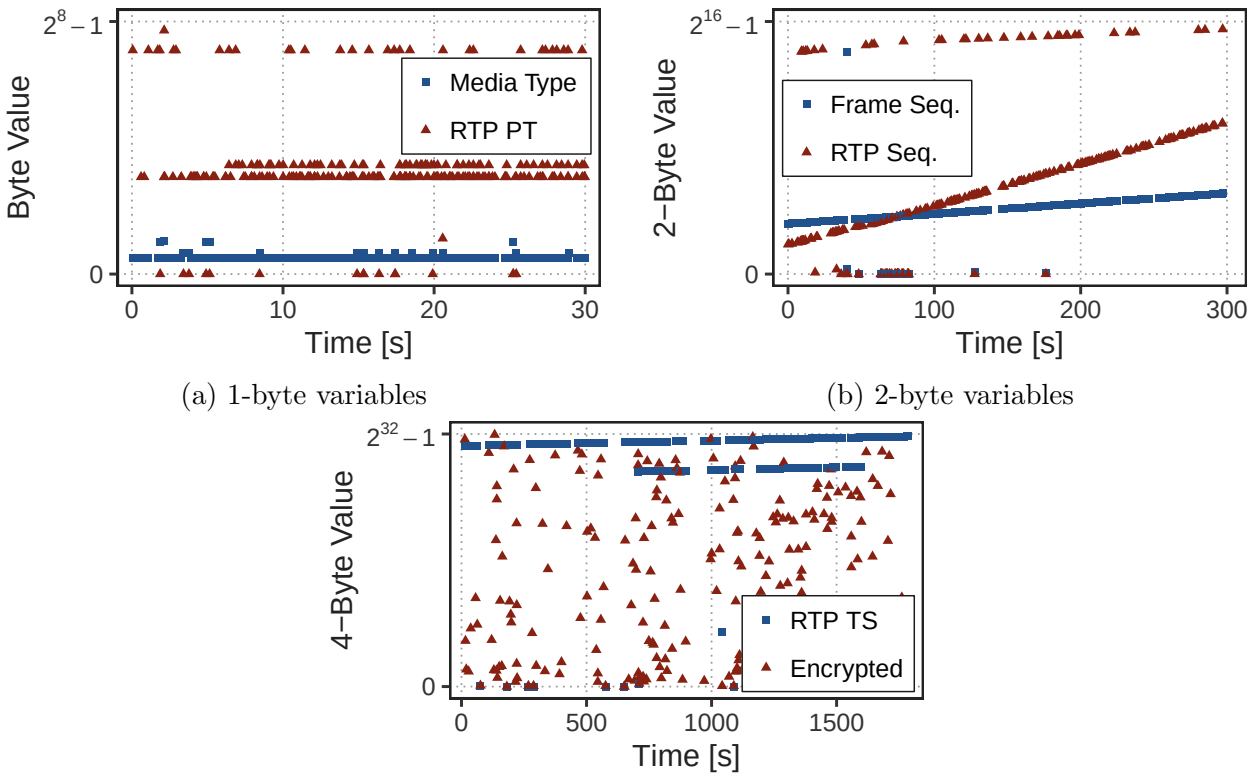


Figure 5.4: Patterns observed in packet header analysis.

ferent slopes, usually overlap at the level of a UDP flow; angled lines commonly wrap around. We suspect that randomly distributed points, especially when covering the entire value space, indeed belong to encrypted portions of the header. Data points following horizontal lines are likely either identifiers (e.g., a stream identifier) or a bitmask (e.g., as in TCP flags). Data points following angled lines are probably either sequence numbers, timestamps, or counters, depending on the range covered and whether the values are monotonically increasing or not.

Figure 5.5 shows actual examples of such plots from Zoom packets that we collected during controlled experiments. Each plot is labeled with the two different header field types that we inferred. Note that not necessarily every data point is the inferred type as other, non-RTP/RTCP packets are typically interleaved with media packets. Figure 5.5a shows two different 1-byte-wide value sequences of a single UDP stream over 30 seconds (250 randomly sampled points); the sequences correspond to a superset of bits that contain the Zoom media type and the RTP payload type, respectively. Figure 5.5b shows the same type of plot with two 2-byte-wide fields, corresponding to the frame sequence number and the RTP (packet)



(a) 1-byte variables

(b) 2-byte variables

(c) 4-byte variables

Figure 5.5: Examples of extracted 1, 2, and 4-byte ranges from a single Zoom UDP flow and their inferred variable type.

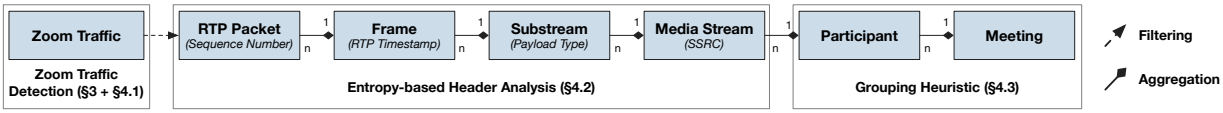


Figure 5.6: Aggregation levels within Zoom meetings with used methodology.

sequence number. Lastly, Figure 5.5c shows sequences corresponding to the RTP timestamp and to four bytes of encrypted payload.

As we were expecting to find RTP headers, we started looking for the most discernible pattern within the RTP header, which is a two-byte sequence-type field (RTP sequence number), followed by a four-byte sequence-type field (RTP timestamp), followed by a four-byte identifier-type field (SSRC). Using this methodology, we are able to detect RTP headers at various offsets in most packets within our trace. We could confirm the presence of RTP headers also by checking the other header values for compliance with the protocol specifica-

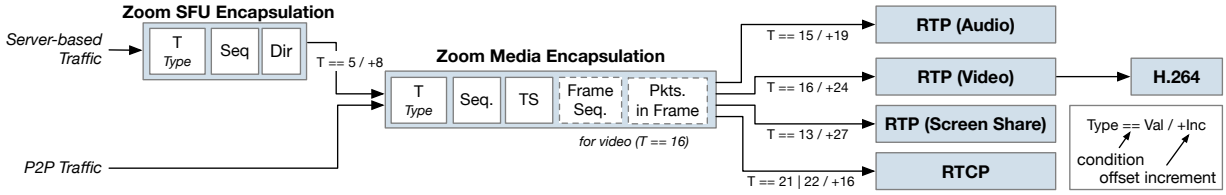


Figure 5.7: Simplified structure of Zoom's custom headers.

tion [137]; for example, the first two bits of the RTP header, the version field, must contain the value `10`.

Finally, suspecting that Zoom also uses RTP's counterpart, the RTP Control Protocol (RTCP), we searched all remaining payloads (where we did not see RTP headers) for the set of SSRC values seen in RTP packets. This is based on the insight that RTCP packets always refer to one or more specific SSRCs and that these values are also carried in the RTCP header. Using this method, we were indeed able to find RTCP *sender reports (SR)* containing timestamps and packet counters being emitted from each sender for each media stream at every second. We did not find any RTCP *receiver reports (RR)* that would contain performance-related information, such as jitter and lost packets. Later on, we will show how these metrics can also be calculated from analyzing the RTP packets alone.

5.4.2.2 Identifying Different Types of Zoom Media Packets

After finding RTP and RTCP headers at different offsets, we needed a recipe to determine where the headers for a given packet start. To do so, we analyzed the payload before the RTP or RTCP header. We took a group of packets with the same RTP header offset and compared them with groups of packets with a different offset. This method allows us to see if there are any header fields before the RTP header whose values are consistently the same within one group but differ between groups. Such a field could be an identifier for the type of packet.

We used this method on the eight different groups of RTP/RTCP header offsets in our traces. Using this methodology, we determined that there is a variable-length header before

Field Name	Byte Range	Comment
Zoom SFU Encapsulation		
– Type	0	`0x05` for 98.4% of packets
– Sequence #	1–2	
– Direction	7	`0x00`/`0x04` – to/from SFU
Zoom Media Encapsulation		
– Type	0	media type or RTCP
– Sequence #	9–10	
– Timestamp	11–14	
– Frame seq. #	21–22	only in video packets
– # Packets/frame	23	only in video packets

Table 5.1: Select header fields in cleartext

Value	Packet Type	Offset	% Pkts.	% Bytes
16	RTP: Video	24	59.48	80.67
15	RTP: Audio	19	24.77	8.89
13	RTP: Screen Share	27	4.62	4.90
34	RTCP: SR + SDES	16	0.89	0.09
33	RTCP: SR	16	0.27	0.02
		Sum:	**90.03**	**94.57**

Table 5.2: Zoom media encapsulation type values

the respective RTP or RTCP header where the first byte indicates the type of packet which also determines where the RTP/RTCP header starts; we will refer to this header as *Zoom Media Encapsulation*. While P2P traffic starts with this header directly (after the UDP header), server-based traffic first has another fixed-length, 8-byte header; we refer to this header as *Zoom SFU Encapsulation*. The overall structure of these headers is depicted in Figure 5.7. Both headers start with a one-byte identifier; we refer to them as *type* fields. The vast majority of SFU encapsulation headers (98.4% of Zoom server-based UDP packets in our trace) start with the type value 5. We found that this value indicates that a Zoom media encapsulation header is following. The remaining header fields we identified and their respective locations within the headers are listed in Table 5.1.

The vast majority (90.03%) of media encapsulation headers in our trace start with the values 13, 15, 16, 33, and 34. Types 13, 15, and 16 are used for RTP media packets and indicate the media type while packets of types 33 and 34 contain RTCP headers. Table 5.2

Media Type	RTP PT	Description	% Pkts.	% Bytes
Video (16)	98	main stream	62.00	79.27
Audio (15)	112	speaking mode	22.04	7.92
Video (16)	110	*FEC*	6.14	7.47
Screen Share (13)	99	main stream	3.59	3.72
Audio (15)	113	mode unknown	2.96	0.89
Audio (15)	99	silent mode	2.60	0.56
Audio (15)	110	*FEC*	0.62	0.13
		Sum:	**99.98**	**99.99**

Table 5.3: RTP payload types values in trace.

shows the mapping between type values, corresponding payload, and offset (from end of the UDP header) where the encapsulated payload starts.

The table also includes the percentage of packets or bytes that contain the specific type in our trace (see more in Section 5.6); we were able to decode over 90% of all Zoom packets as media-carrying packets (94.5% of bytes). We conjecture that the remaining less than 10% of packets carry other control information, e.g., congestion control packets. While we did see some sequence numbers in such packets, we did not further analyze their payload.

5.4.2.3 How Zoom Uses RTP and RTCP

RTP follows a structure of aggregation levels to map a packet to a media stream (see Section 5.2). The levels are depicted in the center part of Figure 5.6 and Zoom uses them in the following way:

- To identify **media streams** (i.e., a participant's audio or video), Zoom uses a limited set of *synchronization source identifiers (SSRC)*. These are unique within a meeting, yet neither globally unique nor appear to be randomly sampled as specified in the relevant RFC [137].

- Each Zoom media stream carries between one and three **sub-streams** identified by an RTP *payload type (PT)*. For audio and video streams, we see the combination of PTs 99, 110, 112, and 113, and 98 and 110, respectively. The sub-streams with PT 110 are not always present and generally make up the minority of packets in a stream. If present, they

use the same timestamps but different sequence numbers than the other sub-stream. We suspect that these streams carry forward error correction data (FEC). In audio streams, we either see packets of types 99 and 112 interleaved or type 113 exclusively. Through controlled experiments, we found that sub-stream 112 carries audio packets when the respective participant is talking (or emitting any significant sound). During periods of silence or only background noise, Zoom uses fixed-size (40B of RTP payload) packets of type 99. This enables us to quantify how much and when a participant actually talks during a meeting when not muted. When type 113 is used, we cannot tell if the participant talks or not; we saw type 113 used when joining a Zoom meeting from the mobile app. Screen sharing always uses type 99. Our trace occasionally contained packets with other payload types where we do not understand their meaning; however,these packets made up less than 0.02% of the over 1.5 billion media packets in our trace. Table 5.3 shows the RTP payload types we understand and their relative frequency.

- Each sub-stream carries **frames** identified by a *timestamp* corresponding to the time when the media was sampled. These timestamps are not expressed in wall-clock time but depend on the sampling rate (see [137]).

- Each frame can be spread over several **packets**. Each packet within a sub-stream is uniquely identified by a *sequence number*. The last packet of a frame generally has RTP's *marker* bit set.

Further Observations. The *contributing source count (CSRC count)* in Zoom RTP packets is always zero indicating that every RTP packet only carries a single media source. This suggests that Zoom indeed uses an SFU architecture and not a multipoint control unit (MCU) where media is transcoded and resulting frames effectively carry several signals (sources).

The RTP header in media packets includes RTP extensions and is followed by a *H.264 fragmentation unit (FU) network abstraction layer (NAL)* header in case of video packets.

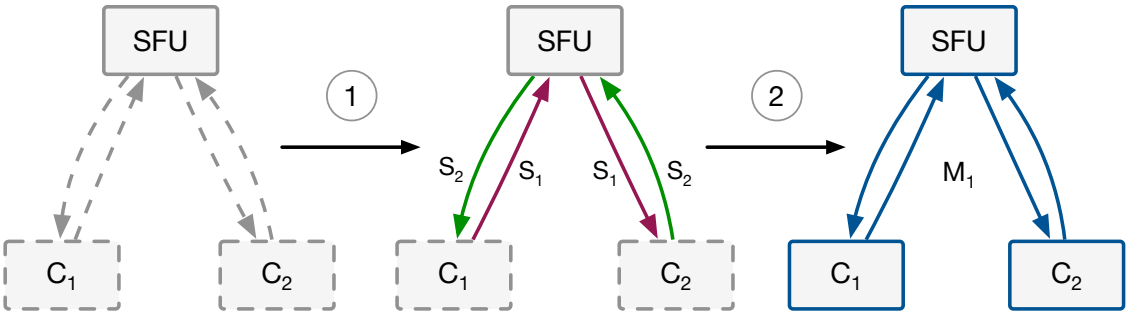


Figure 5.8: Process for grouping streams into meetings.

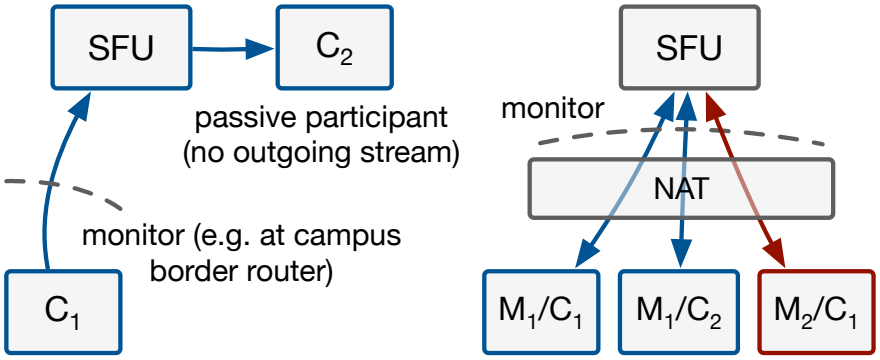


Figure 5.9: Limitations of grouping heuristic.

After this header, the remaining payload appears to be encrypted. We did not further investigate the payload of audio packets.

The RTCP packets that we see are only *sender reports (SR)*. RTCP sender reports accompany a media stream and are used to periodically synchronize wall-clock time with RTP timestamps by carrying an NTP timestamp. This mechanism ensures that receivers play media at the right speed and that different streams from the same source (e.g., audio and video) are synchronized. Some SRs also include a *source description (SDES)* which is, however, always empty.

Wireshark Plugin. We wrote a Wireshark dissector plugin incorporating all our findings. Using this plugin, which we make available to the community, it is easily possible to analyze Zoom traffic.

5.4.3 Grouping Streams into Meetings

Finally, we developed a heuristic to group media streams belonging to a single meeting together. This enables us to (1) calculate round-trip-time (RTT) between our monitor and the SFU by comparing copies of the same stream while going to and coming from the SFU and (2) judge whether only a single participant is affected by poor meeting performance or if the meeting in general suffers from problems. During our experiments and campus-level traffic analysis, we did not find a meeting identifier in Zoom's packet headers, and as a result, we need to rely on other flow properties and header fields including IP addresses, port numbers, SSRC, RTP timestamps, and sequence numbers to group streams.

5.4.3.1 Challenges Associated with Grouping Streams.

Developing such a heuristic, however, is challenging for several reasons: (1) Locally used port numbers and server/peer IP addresses and port numbers change when a meeting switches between SFU and P2P modes. As a result, a heuristic cannot easily associate an IP address with a meeting and must account for changes in the meeting mode. (2) SSRCs used by Zoom are unique within a meeting but not globally unique nor randomly chosen (as specified in the RTP RFC), making it difficult to detect duplicates of a particular meeting stream (i.e., after stream replication by the SFU) solely through its SSRC. (3) A participant that mutes their microphone and has their camera turned off also does not emit any media streams despite being a participant in the meeting. As network-level heuristics for this task rely on observing media streams, depending on the vantage point, such passive participants may be invisible. This point, in particular, compromises the accuracy of any such heuristic and makes it difficult to validate their accuracy. Fortunately, to estimate performance, we are only interested in active participants as there is no stream performance to measure when there is no media stream.

While the above-mentioned limitations can compromise the accuracy of the participant count and meeting duration, one of the main purposes of this heuristic for our study is to

enable fine-grained RTT estimation. For this, we look at *copies* of a media stream observable from our vantage point. This occurs when an on-campus participant sends a media stream which is then replicated by the SFU and sent "back", through our monitor, to another on-campus participant. We explain this method in more detail in Section 5.5.3. Detecting stream copies, which is the only part of the heuristic required for RTT estimation, is based on four different features (time, SSRC, RTP sequence number, and RTP timestamp) that all need to match, making it relatively robust.

5.4.3.2 Heuristically Grouping Streams into Meetings.

Despite these limitations, we developed a heuristic to provide an estimate of the number of meetings and participants in a meeting. As an example, consider an SFU-based, audio-only Zoom meeting with two participants (C_1 and C_2) as illustrated in Figure 5.8. Each participant sends their audio stream (S_1 and S_2) to the SFU which forwards it to the other participant. Our heuristic consists of two steps.

Step 1: Finding Duplicate Streams. The first step reads RTP packet records and groups them into media streams identified by IP 5-tuple and SSRC. Whenever a new stream is created (i.e., when the stream key does not exist yet), it checks if there is an existing stream with the same SSRC (but different 5-tuple) where the most recently seen RTP timestamp is within a small range of the first RTP timestamp of the new stream. This is based on the insight that during a transition between P2P and SFU modes the IP 5-tuple of the flow changes, but not any of the RTP-level information (e.g., SSRC). Also, Zoom's SFU does not translate timestamps or sequence numbers and, as a result, an outgoing media stream that is sent back to a different client within a campus (i.e., within the monitor's vantage) will have the same RTP-level information but appear slightly later (RTT to SFU plus processing time). After this step, every stream that carries the same media (e.g., a single participant's audio or video) is assigned a unique identifier (here S_1 and S_2).

Metric	Requires Headers	Available in Z. Client	Validated
Overall Bit Rate (§5.5.1)			
Media Bit Rate (§5.5.1)	•		
Frame Rate (§5.5.2)	•	•	• (Fig. 5.10a)
Frame Size (§5.5.2)	•		
Latency (§5.5.3)	•	•	• (Fig. 5.10b)
Jitter (§5.5.4)	•	•	• (Fig. 5.10c)

Table 5.4: Key Zoom performance and quality metrics

Step 2: Assigning Streams to Meetings. The second step operates on stream records, including SSRC, IP 5-tuple, stream start and end time, and, most importantly, the unique identifier from step one. This identifier (based on RTP header values) greatly increases the accuracy of the following algorithm. The algorithm starts by assigning the first stream to a new meeting. For every subsequent record, it then decides whether to assign the stream to an existing meeting or to create a new meeting for the stream. The heuristic maintains mappings from the unique stream id, the client's IP, and client's IP and port combination to meeting identifiers. If a lookup in this data returns at least one match, the stream is assigned to the respective meeting. If there are several matches with different meeting ids, the matched meetings are merged. If there is no match, a new meeting is created.

This heuristic works well in most cases but its efficacy is constrained, as mentioned above, by the vantage point where packets are captured. The two key issues of our approach are depicted in Figure 5.9. The left side of the figure shows the above mentioned case of a passive meeting participant whose media stream is not visible at the monitor. The right side shows the issue that arises when NAT is used within the campus (e.g., by connecting a personal hotspot) or when the monitor is placed outside of a large-scale NAT. Here, meetings M_1 and M_2 could be merged as they appear to the monitor as using the same IP address.

5.5 Estimating Performance Metrics

Understanding parts of Zoom's packet headers, enables us to extract and analyze a wide range of protocol fields from media traffic. However, extracting the fields does not, by itself, provide useful insights into meeting performance. In this section, we discuss our methods of deriving various network-level and performance metrics from Zoom traffic and show how we validate our methods using statistics provided by the Zoom client application. Table 5.4 provides an overview of the most important metrics discussed below.

Validation of Metrics. Zoom provides session performance statistics to users and operators in three different ways: (1) in the Zoom client application's *Statistics* window, (2) through a dashboard and REST API for operators [195], and (3) programmatically through an SDK [196]. The available metrics are latency, jitter, packet loss, audio frequency, video resolution, and video frame rate. Each metric is available for each of the three media types. Both the Zoom client and the SDK seem to update their data roughly once per second while the REST API provides updated data once per minute [192].

Using the GUI would require us to automatically take frequent screenshots of different tabs in the Settings window and process the images later. The API's update rate of once per minute is too coarse-grained. Consequently, we decided to use the SDK to validate our methods and compare the accuracy of our metrics to the ground truth provided by Zoom. We wrote a custom Zoom client application using the macOS SDK in Objective-C based on the example code provided by Zoom [196]. The application opens a standard Zoom meeting window and session. The traffic generated by this client looks exactly like the traffic generated by the public Zoom client. We instrumented the code to log all available performance metrics for both audio and video once per second (the finest granularity that the API supports).

Using this setup, we performed a series of controlled experiments where we collected both the Zoom-provided data and all network traffic for later analysis using our tools. We ran

several 5–6 minute-long two-person meetings where we introduced cross-traffic twice during each call by running a network bandwidth test for 10–20 seconds each.

5.5.1 Overall and Per-Media Bit Rates

Flow-level Bit Rate. The overall data rate of Zoom flows (per IP 5-tuple) can be easily measured from network traffic captures and does not require parsing any Zoom headers. Data rate has been used to (1) characterize Zoom's bandwidth requirements, (2) estimate session quality, and (3) differentiate between audio and video streams (via the relative difference in data rate between flows) [34, 86, 93, 118].

While suitable for quantifying bandwidth requirements, the overall data rate does not correctly characterize session performance or quality. Zoom's rate-adaptation algorithm adjusts the sending picture quality and frame rate in response to network conditions, user interactions, and device capabilities. Recall from Section 5.4 that the following user interactions and circumstances affect the overall bit rate:

- Moving a little or a lot while on camera causes the video data rate to change rapidly.
- Resizing the video display (e.g., to thumbnails during screen sharing or using "speaker-only" view) at the receiver can reduce the frame rate by half.
- In Zoom, mobile devices use different aspect ratios and video resolutions than desktop computers, making comparisons between overall data rates difficult.

Furthermore, a single UDP flow may carry several individual media streams. Thus, overall flow-level bit rate is insufficient to estimate meeting quality as low bit rates can be caused by other factors. Moreover, differentiating between media types based on relative bit rate of a flow is inaccurate as a stream with low frame rate and low resolution (as possible in thumbnail mode or sometimes during screen sharing) can have a similarly low overall bit rate as an audio stream (more in Section 5.6). Lastly, in our trace, roughly 10% of Zoom

UDP packets carry no media data (Table 5.2), causing inferences about media bit rate from overall flow bit rate to be inherently inaccurate.

Per-media Bit Rate. Our deeper understanding of Zoom's protocol, including the ability to determine the media type and sender (via SSRC) for each packet, allows us to put packet and bit rates into context. Further, knowing exactly which packets carry media and where the media payload starts, enables us to compute the actual media bit rate.

5.5.2 Frame Rate and Frame Size

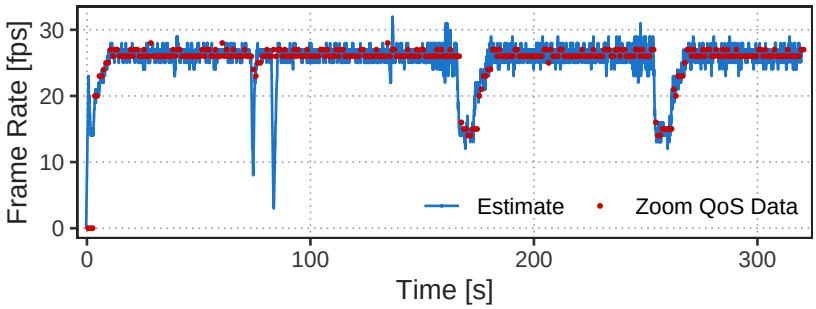
Frame Rate. The frame rate of a Zoom video or screen-sharing stream can precisely be calculated in two ways. As each frame is identified by an RTP timestamp for the decoder to know when to play back the frame, frame rate can be estimated simply by counting the number of unique timestamps seen in a 1-second period. The second approach is to derive the encoder's frame rate from the increments of the RTP timestamp in conjunction with the stream's sampling rate. The results of this approach can be different from the first method in the presence of network congestion; we explain the difference in more detail after describing both methods.

Method 1: To compute the frame rate, we use a circular buffer to store all frames that were *completely* delivered within the last second. We discovered a field in the Zoom Media Encapsulation header that contains the number N of packets in a given frame (Section 5.4.2). We consider a frame complete when we see N distinct (per sequence number) RTP packets with the same RTP timestamp. Whenever a new frame completes, we check if the head of the buffer still falls within a 1-second interval from the current time and remove it if not. The current frame rate is then simply the occupancy of this buffer and can be computed at any time (e.g., for each frame). Using this method, we can also calculate how long it takes to deliver a given frame which can then be compared to the time the frame covers in the media stream, the *packetization time* [137] (more in Section 5.5.5).

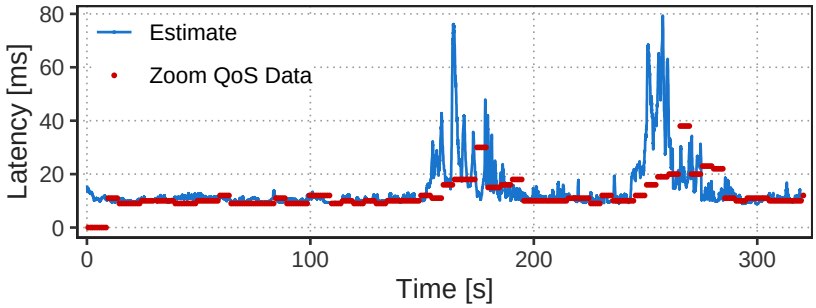
Method 2: A second approach to calculate frame rate leverages the RTP timestamp to exactly calculate the encoder's (i.e., the "intended") frame rate but requires knowledge of the stream's sampling rate. Through a simple parameter sweep and comparing the result with data obtained through the method above, we found that Zoom's video streams use a sampling rate of 90 kHz, which also happens to be the recommended value for sending conferencing video over RTP [136]. The frame rate FR at sampling rate SR for each frame can then be calculated as $FR = SR/\Delta RTP$ where ΔRTP is the RTP timestamp increment from the last frame. The packetization time is then given by FR^{-1}. Note that the encoder's frame rate is not necessarily the rate of successfully delivered frames over the network as computed using the first method; it is rather the frame rate the encoder is currently sending at. In the presence of congestion, the two numbers can temporarily diverge before the encoder adjusts the frame rate, indicating a network problem.

We validated our frame rate estimation based on the first method using the statistics provided by the Zoom SDK. Figure 5.10a shows the results of one such experiment; our estimate closely matches the data provided by Zoom. The frame rate mostly fluctuated between 26 and 28 fps and dropped temporarily during the competing download. Our data also shows the frame rate dropping twice, but Zoom's data reflects only the first drop, because of its relatively low refresh rate and a possible smoothing mechanism.

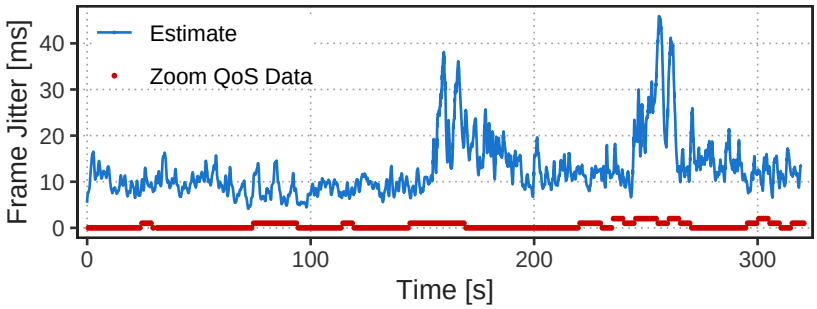
Frame Size. Finally, knowing which packets belong to a particular frame, how many packets are expected in a given frame, and where the RTP payload starts, allows us to exactly calculate the size (in bytes) of a media frame. Even though frame size does not account for user interaction (e.g., reduced frame size due to thumbnail mode), together with frame rate, it gives a more accurate estimate of the size, resolution, and quality of the currently displayed picture than the overall flow bit rate.



(a) Frame rate estimation accuracy.



(b) Latency estimation accuracy.



(c) Frame-level jitter estimation accuracy.

Figure 5.10: Estimation accuracies from single experiment.

5.5.3 Latency

End-to-end latency between participants is a key performance metric in video conferencing. The ITU found that users begin noticing difficulty having conversations when the mouth-to-ear latency (i.e., the time from recording an audio signal to playing it back at the receiver) exceeds 200 ms [70]. While we cannot accurately measure mouth-to-ear latency, we developed techniques to (1) estimate the end-to-end latency (through the SFU) between pairs of on-campus participants in the same meeting and (2) estimate the round-trip time between an individual participant and the SFU.

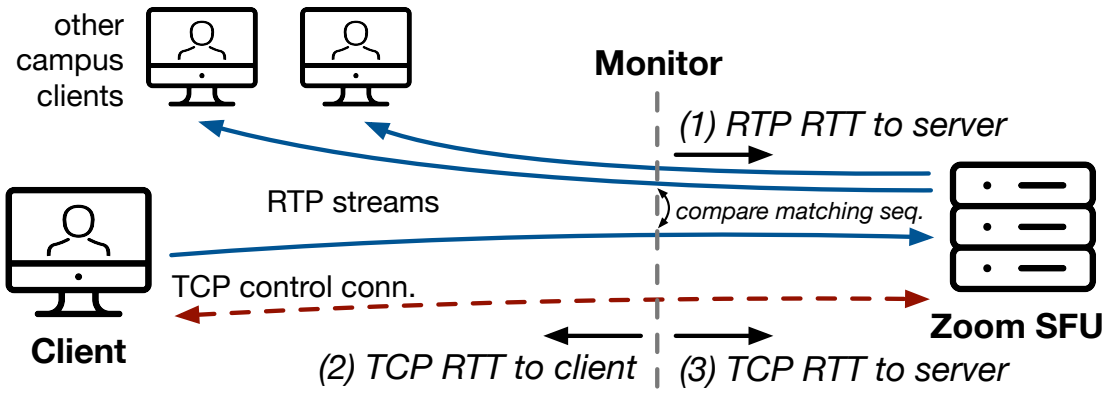


Figure 5.11: Methods for measuring session latency.

Method 1 (latency via RTP sequence numbers): The first approach to estimate session latency leverages the media stream itself. RTP packets carry sequence numbers and the SFU simply forwards these packets to all meeting participants. As a result, if we monitor a meeting with several participants and have RTP streams carrying the same media grouped together, the RTT between the point where packets are captured and the Zoom SFU can be measured by comparing the egress and ingress timestamps of RTP packets with matching sequence numbers (see blue, solid lines in Figure 5.11). Depending on media type and quality, each stream produces tens and up to hundreds of packets per second, making this method a way to obtain very frequent RTT probes.

Method 2 (latency via TCP as a proxy): Even if our monitor sees only one of the meeting participants communicating with the SFU, we can leverage the client's TCP control connection (Section 5.3) to estimate the RTT from the monitor to the client and to the SFU. This is shown by the red dashed line in Figure 5.11. TCP RTTs can be measured by matching TCP sequence numbers of outgoing packets with acknowledgment numbers of incoming packets [32, 144]. That is, we use TCP RTTs as a *proxy* for the latency of real-time media [6, 109]. Despite providing fewer samples than the RTP-based method due to the comparatively lower packet rate of the control connection, it does help us measure the latency from our vantage point both to the Zoom SFU and the client. The difference between

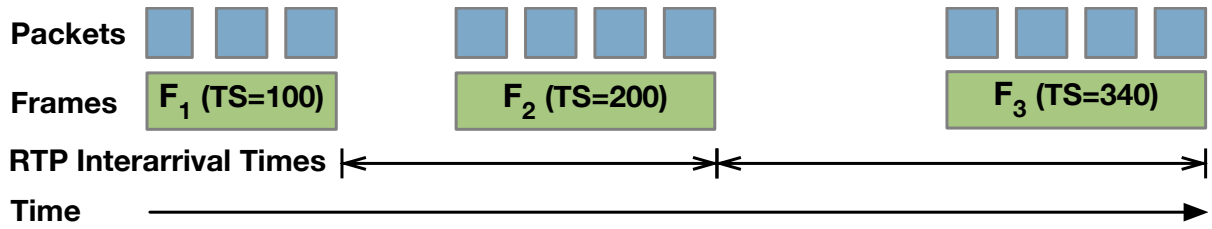


Figure 5.12: Frame-level interarrival time calculation.

the two can be used to pinpoint whether congestion is located upstream or downstream from the measurement device, e.g., inside vs. outside our campus network.

Figure 5.10b shows the accuracy of our latency estimation based on the first method for the same experiment and video stream as in Figure 5.10a. We can see that Zoom only updates its latency estimate every five seconds. Without cross traffic, our estimate matches the data provided by Zoom but yields significantly more data points as we can calculate latency for every single RTP packet. As a result, our method more clearly highlights fluctuations in latency, especially during periods of rapid variation in network condition.

5.5.4 Jitter

Jitter, defined as the "statistical variance of the RTP data packet interarrival time", provides a short-term measure for network congestion and may indicate congestion before loss occurs [137]. Of the metrics outlined in Table 5.4, it is the most direct estimator for network quality and therefore important for reasoning whether, for example, low frame rate is caused by the network or by the user's behavior as explained above.

Simply computing the variance in interarrival time between packets is not useful in the context of RTP streams for two reasons. First, a UDP video-conferencing stream can carry several media streams which can then carry multiple sub-streams. The notion of packet order (via RTP sequence numbers) is only valid within each such sub-stream, requiring parsing the respective RTP headers. Second, RTP traffic at the packet level is bursty by nature, as each frame generally spans several packets which are transmitted back-to-back. As a result, we usually see short bursts of packets belonging to a frame, followed by a pause before

the next burst, as illustrated in Figure 5.12. Moreover, the packetization time (i.e., the time a frame covers in the media signal) is not necessarily constant throughout the stream which requires correcting the jitter computation by what the interarrival time for any two frames *should* be. In fact, Zoom uses variable packetization intervals as indicated by variable increments between RTP timestamps in Zoom traffic. We use the formulas outlined in the RTP RFC [137] to compute this adjusted time difference and subsequently the frame-level jitter. Jitter can either be computed in terms of RTP time or as wall-clock time by first converting between them using the stream's sampling rate.

In our experiments, Zoom always reported very low jitter which never exceeded 2ms, even in the presence of congestion. As shown in Figure 5.10c, our jitter estimate does not match Zoom's estimate but appears more in line with the fluctuation in latency during the same experiment which was also reported by Zoom, especially during the two congestion events (Figure 5.10b) which even caused Zoom to adjust the video frame rate (Figure 5.10a). We are surprised by this result as the significant fluctuation in latency depicted in Figure 5.10b should also be reflected in the (resulting) jitter metric in Figure 5.10c. We hypothesize that Zoom computes jitter differently, perhaps taking forward error correction into account, or using a very long smoothing interval. We use the jitter computation method recommended for RTP by the corresponding RFC [137].

5.5.5 Other Metrics

Loss and Retransmissions. While computing the number of lost packets, retransmissions, and out-of-order deliveries for TCP connections is relatively straightforward using TCP sequence numbers, this is ordinarily not possible for UDP traffic. Our ability to parse RTP headers in Zoom traffic, however, allows us to estimate these metrics over an RTP stream. In our controlled experiments, we found that Zoom retransmits lost packets up to 2 times. As a result, we rarely see entirely lost packets in our trace but rather duplicates. Since we are unable to find any explicit loss signals in the traffic, we must rely on analysis of the seen

sequence numbers to estimate reordering, loss, and retransmissions. This method, however, can be inaccurate as we are not able to differentiate between retransmissions and regular packets (apart from observing elevated jitter). Some observed reorderings might also be due to retransmissions. In conclusion, Zoom's use of retransmissions makes it fundamentally difficult to infer loss and packet reordering from in-network measurements and sequence numbers alone and we require other metrics (e.g., jitter) to assess network condition.

Frame Delay. Finally, we can measure the time between the first packet of a frame and the time the frame is fully delivered; we call this time *frame delay*. High frame delay (in comparison to other frames in the stream) indicates that retransmissions took place to fully deliver the frame. In those cases, we observed that the frame delay is elevated by at least the current RTT to the SFU plus a timeout that appears to be 100ms. As retransmissions in Zoom use the same RTP sequence numbers as the originally lost packet, it is not straightforward to detect lost packets if the packet was dropped before the vantage point. Observing a packet with suspiciously high delay (i.e., 100ms + RTT) delivered out-of-order, however, is a strong indicator that the respective packet was retransmitted in response to loss.

Also, we can compare a frame's packetization time with its delay. If the delay is larger than the packetization time over the course of several frames, the jitter buffer gets drained and the video will eventually stall. We leave the detection and deeper analysis of audio and video stalls based on this metric for future work.

5.6 Analyzing Zoom Campus Traffic

We now show how our methods can be applied to a large dataset by first describing our capture system and then discussing performance metrics computed over a 12-hour trace collected at our campus network. The trace contains 1.8 billion Zoom packets and 59,020 RTP media streams.

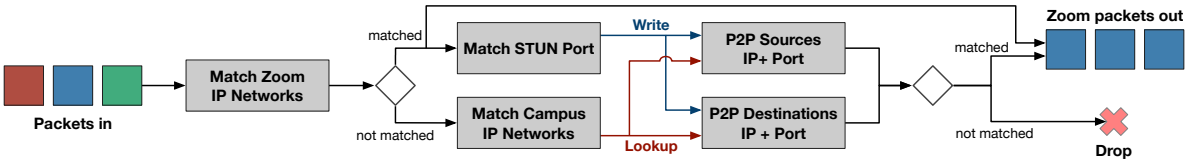


Figure 5.13: Zoom packet capture program implemented in P4 for the Intel Tofino programmable switch.

5.6.1 Scalable P4-based Zoom traffic capture.

To study video-conferencing systems, we must first identify and capture their traffic. A commonly employed method is capturing a large volume of on-campus traffic using a tool like *tcpdump* and then extracting packets of interest—Zoom packets, in this case—in a post-processing step. However, this approach does not scale when the traffic rate ranges from several Gbps to tens of Gbps, as is the case on our campus. Bottlenecks exist at links from our packet broker system to our collection server where *tcpdump* runs, as well as packet and disk I/O at the server. Storage space also becomes an issue. Fortunately, our campus traffic capture setup is equipped with a high-speed programmable switch, namely the Intel Tofino switch [68]. The switch sits between the packet broker system and the server where *tcpdump* operates. This allows us to deploy a data-plane program (written in the P4 [20] language) that takes all campus packets as input and only allows Zoom packets to pass through to *tcpdump*.

Our campus is connected to the Internet via two separate gateways. At both gateways and in both directions each we have taps installed that passively capture all packets and forward them to our packet broker system. To manage overall capture volume, we do exclude some of our campus' internal subnets from the capture at the broker. These excluded subnets mostly belong to research computing facilities that run large bulk transfers to the Internet but are unlikely to contain significant amounts of Zoom traffic. Zoom clients in these unmonitored subnets look to us like external (off-campus) clients. For these clients, we see outgoing streams only if they are being forwarded to a client that is within one of the monitored subnets; we never see the SFU to client leg of incoming streams to those clients. For all

Resource Type	Zoom IP Match	P2P Detection	Anonymization [81]
Stages	2	7	11
TCAM	0.7%	1.0%	1.4%
SRAM	0.1%	10.9%	1.1%
Instructions	1.3%	3.4%	5.2%
Hash Units	0.0%	16.7%	8.3%

Table 5.5: Hardware resource usage of the Tofino-based capture program (divided by functional component).

other (monitored) subnets, we do see every Zoom packet to and from the Internet. As a result, every stream that we do see is complete but we may miss a limited number of streams entirely which can affect the number of meeting participants that we report.

Figure 5.13 illustrates the design of our Zoom packet filter system. For TCP and server-based UDP traffic, it suffices to check in a stateless manner whether one of the source or destination IP addresses matches the list of IPs published by Zoom [197]. Detecting P2P traffic, however, requires a more sophisticated stateful approach. As shown in the figure, whenever we *see* a Zoom STUN packet, we write the campus peer's address (IP address and port number) to these registers (see Section 5.4.1). Subsequently, for all future non-server UDP packets, we extract the campus-side address and look it up against our hash tables. The P4 program also performs anonymization for all outgoing packets using an existing system [81].

The resource consumption (by resource type and functional component) of our system on the Intel Tofino Programmable Switching ASIC [68] is shown in Table 5.5. The percentages in the table refer to the fraction required of the respective resource type available on the Tofino. The table shows that the most complex operation we perform is anonymization, which may be optional in certain production environments. We conclude that our program is lightweight as it uses less than 15% of most of the resource types available on our switch; it can therefore easily and practically be combined with other data-plane processing logic. Our capture system, while designed for Zoom, can be extended to support other applications with known *signatures*, e.g., other video-conferencing applications.

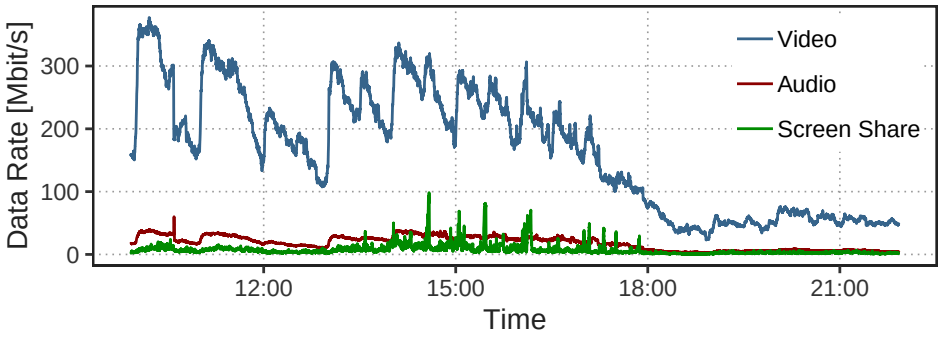


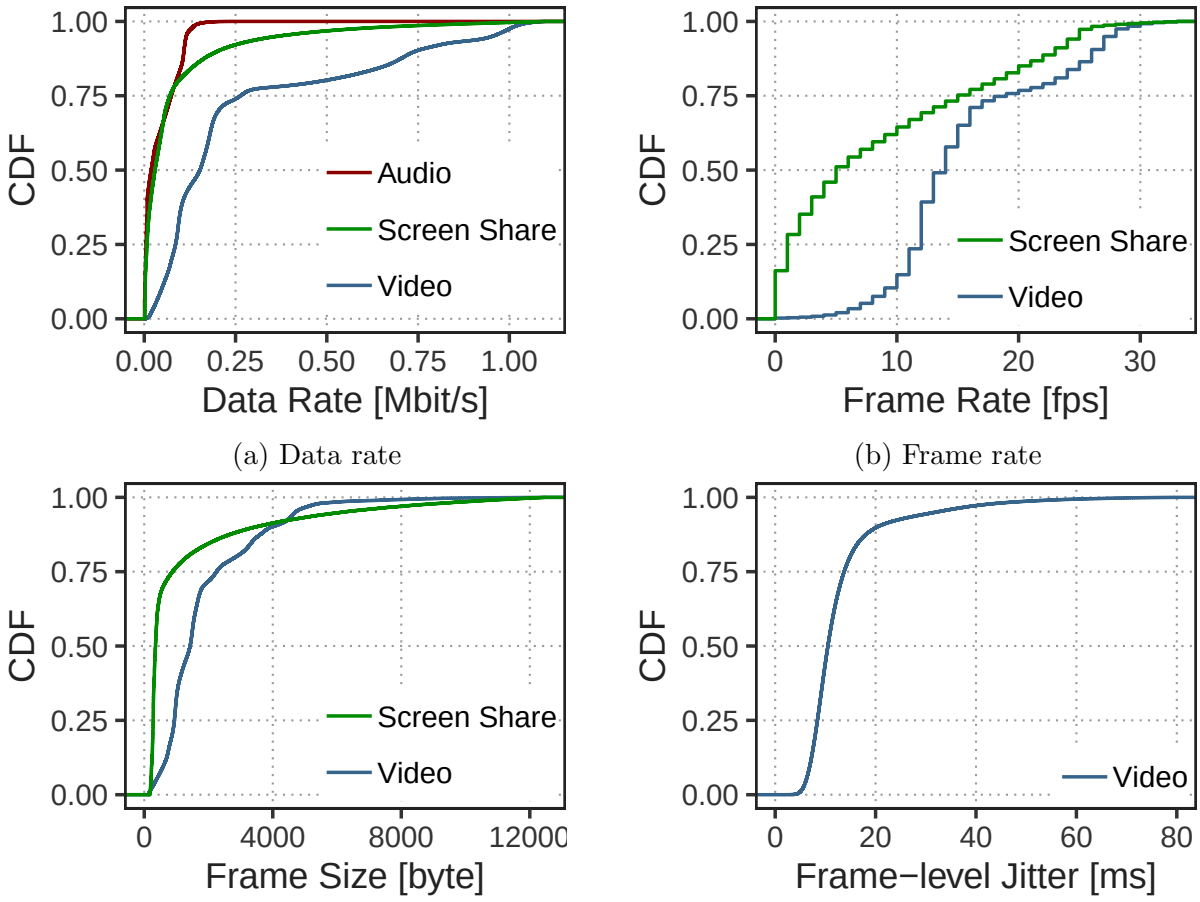
Figure 5.14: Data rate per media type in campus trace.

5.6.2 Zoom Performance Metrics in the Wild

We calculated various performance metrics for each video, screen share, and audio stream in 1-second bins over our entire trace, resulting in roughly 33 million data points for each metric.

Media Bit Rate. Figure 5.14 shows the total media bit rate for all Zoom streams per media type over local time. Video traffic makes up the vast majority of data and we can clearly see spikes in bit rate at each full hour and (to a lesser extent) every 30 minutes as meetings presumably begin during those times. There is a dip during lunchtime and significantly less activity after the end of work day. The distribution of media bit rate per media type is depicted in Figure 5.15a. Interestingly, the bit rate distribution of screen sharing traffic is much closer to that of audio traffic as opposed to video traffic. This illustrates that it is inaccurate to differentiate different stream types based on relative bit rates.

Frame Rate. The distribution of frame rates between screen sharing and video traffic, depicted in Figure 5.15b, shows that Zoom uses a very fine-grained encoding scheme for screen-sharing traffic where no new frames are generated, presumably when the picture does not change frequently, as is often the case in presentation slides. In fact, roughly 15% of frame rate samples for screen sharing showed a frame rate of zero; approximately half of the samples had a frame rate of five or less with the remainder of the samples being relatively evenly distributed. In contrast, video frame rates are under ten frames per second (fps) only in 10% of the data points with a lot of the probability mass being centered around



(a) Data rate

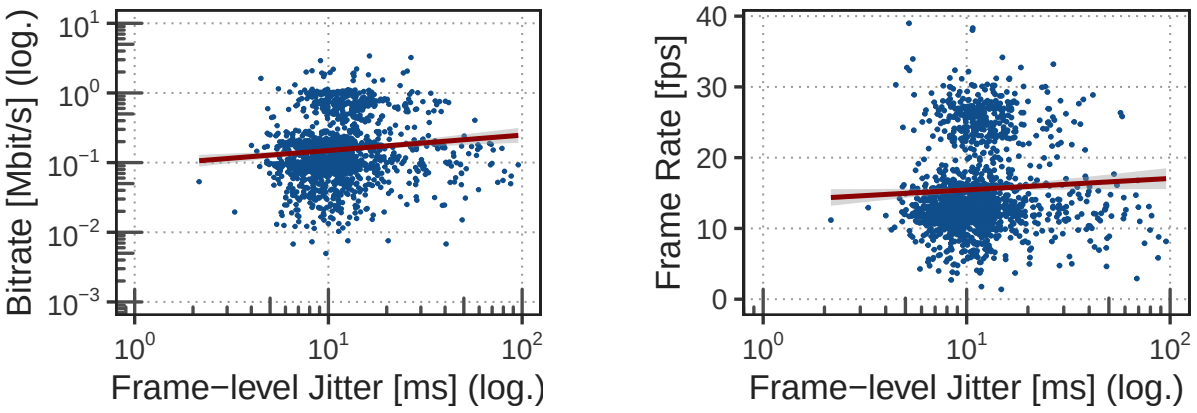
(b) Frame rate

(c) Frame size

(d) Frame-level jitter

Figure 5.15: Distribution of performance metrics per media type in campus trace.

11–14 fps. As mentioned earlier, in controlled experiments we observed that Zoom usually tries to achieve a frame rate around 28 fps and in cases of video thumbnails or massive network congestion reduces the rate abruptly to around 14 fps. Our data reflects this and shows that the majority of video streams have frame rates in this range, meaning that a lot of video is transmitted in this "reduced-fps mode". We assume that the samples in the range of 20–25 fps (approximately 10–15%) are due to short-term adaptations of frame rate or partially delivered frames in the presence of network events (see Section 5.5.2). This data suggest that low frame rates are usually caused by user interaction rather than network instability as there is a disparity between the frequency of low frame rate below 20 fps (almost 75% of cases) and high jitter of more than 20 ms (less than 20%).



(a) vs. bit rate

(b) vs. frame rate

Figure 5.16: Lack of correlation between jitter and other performance metrics.

Frame Size. The frame size distribution in Figure 5.15c also shows differences between screen sharing and video streams. We see that over half of screen-sharing frames are smaller than 500 bytes but the distribution has a long tail. We assume that the character of the media (need for high resolution but little movement in the image) causes this distribution. Frames that contain a lot of new information (e.g., initial frames, changing slides, etc.) take more data than usual video frames and after that only small, incremental changes are required. In contrast, the majority of video frames are smaller than 2000 bytes and only a few are larger than 5000 bytes.

Frame-level Jitter. Since we are not certain about the sampling rate of audio and screen-sharing streams, we only include the results of our frame-level jitter computation for video traffic, where we determined the sampling rate to be 90 kHz (Section 5.5.4). Overall, most samples have a frame-level jitter below 20 ms but the distribution's tail is long. Zoom recommends jitter below 40 ms [191]; roughly 5% of samples in our trace show jitter greater than this.

Causes of Low Performance Metrics. Lastly, we illustrate how seemingly poor metrics are not necessarily a result of poor network performance. We mentioned before that low frame rate is often caused by user interactions and meeting characteristics. As an example,

Figure 5.16 shows 1,500 randomly chosen samples from our data set of performance metrics in 1s-bins where we plot video bit rate and frame rate, respectively, on the y-axis and frame-level jitter on the x-axis. Jitter, here, serves as a metric that is mostly influenced by the network whereas the other two metrics are influenced by a variety of factors. We can see that there is no direct correlation between jitter and bit rate, or between jitter and frame rate, meaning that bit rate and frame rate adaptations are in many cases not a result of poor network conditions. Figure 5.16b also clearly shows the two aforementioned frame rate modes as clusters centered around 14 fps and 28 fps, respectively. As a result, relying on a single metric, like bit rate, alone is not sufficient to estimate the quality of an ongoing meeting; several fine-grained performance metrics must be evaluated in conjunction, which is what our work enables.

5.7 Related Work

Demystifying Black-box Protocols. Research in reverse engineering of network protocols has a long history, with Skype P2P audio call analysis as one of the earliest works for real-time network applications [11]. Our methodology and techniques are built upon this large body work in protocol reverse engineering [18, 38, 92, 113, 162], including inferring protocol structure from network traces [26, 40, 173]. These works focus on determining boundaries between header fields but do not use entropy-based analysis to infer the semantics of those fields. Additionally, we are the first to use this systematic approach in the context of video conferencing.

Measuring Performance of VCAs. Nisticò et al. performed a controlled experiment on 13 popular Real-Time Communication (RTC) applications to understand how they operate and consume bandwidth [118]. MacMillan et al. analyzed Zoom, Google Meet, and MS Teams to measure their performance and network utilization [93]. Both works provide key insights about how VCAs operate and perform. Yet, they focus on bit rate and link

utilization, and how VCAs compare with each other. Our work focuses on Zoom and dives deep into its protocol to extract performance metrics to better understand user experience. Chang et al. measured and analyzed Zoom, Webex, and Meet based on data from over 700 VCA sessions [29]. The video QoE metrics are measured at the Zoom clients, applying Peak Signal to Noise Ratio (PSNR) and structural similarity index (SSIM) analysis on actual video feeds. Our work is about *inferring* the performance and quality of Zoom calls using passive measurements collected in the network data plane.

Zoom Traffic Analysis. Several prior works [29, 34, 86, 93] studied Zoom among other VCAs to uncover implementation details and characterize its performance using controlled experiments. Their findings on Zoom characteristics are summarized in Section 5.3. Unlike these prior works, we go further by inferring performance metrics from packet traces that are passively collected from a large production network. A second line of work on Zoom focuses on security, instead of performance. Mahr et al. performed a detailed forensic analysis of Zoom, primarily using disk, network, and memory forensics [94]. It demonstrates that it is possible to find users' critical information, such as chat messages, names, email addresses, and passwords, in plain and/or encrypted text. Similarly, a CitizenLab blog post [95] provides insights into the privacy and general security properties of Zoom.

5.8 Discussion

Generalizability. While our work focuses on Zoom, we believe our header and protocol analysis methods can be used to demystify other black-box systems. Also, if Zoom changes its protocol in the future, these techniques can be applied again. Our tools and techniques to study Zoom performance are also largely applicable to other video-conferencing systems that employ RTP for media transfer. As RTP is used in the vast majority of these systems as reported in Table 2 in [118], our techniques for estimating (among others) latency, frame rate, jitter, and bit rate are applicable to a wide range of applications, including Google

Meet, Microsoft Teams, Cisco Webex, and Apple FaceTime. Of course, other aspects such as P2P connection detection, mapping of payload types, or other parts of Zoom's behavior reported in this chapter are specific to Zoom. We, however, did share our methodology in detail to facilitate similar future studies on Zoom, if required, or on other proprietary protocols. Consequently, even if Zoom were to make their protocol and header format public in the future, our performance measurement techniques will continue to remain relevant and novel.

Limitations. Two of our techniques are fundamentally limited. First, for quantifying loss and retransmissions, we cannot disambiguate with certainty between fresh packets and their retransmitted copies. A heuristic to detect retransmissions could analyze frame delay (Section 5.5.5). If the delivery of a frame (normally consisting of packets sent back-to-back) takes longer than the connection's RTT, at least one retransmission likely happened within this frame. Second, due to our vantage point, we do not see media streams of off-campus participants that do not send any media (i.e., are completely passive). This is a problem for quantifying the number of overall meeting participants as their media streams are transported entirely outside of our campus and we do not have access to the respective packets. For this reason we are not able to measure the performance of the media streams sent to these participants.

In-Network Monitoring and Control. There will be extra benefits if our performance analysis can run in a high-speed programmable switch at line rate (e.g., Intel Tofino [3, 68]). In particular, the switch's proximity to the client would enable it to take immediate actions in response to degraded Zoom performance. This would benefit both the campus network operators and Zoom. We can already identify and parse Zoom headers in the data plane; the computations of our performance metrics can be implemented in a streaming fashion and are amenable to data-plane implementation. The space constraints of high-speed programmable switches may require approximate data structures limiting overall accuracy.

Control actions that may be performed on a switch include selectively forwarding layers in an SVC stream or annotating packets (e.g., using DSCP) based on their type, relative importance, or dynamically in response to congestion. We leave this as future work.

Labeled Datasets for ML-based QoE Inference. While we report performance metrics that affect meeting quality, we do not use these to infer the quality-of-experience (QoE). Our metrics can, however, be used as *features* in a QoE ML inference model where labels can be created by collecting opinions from viewers. To this end, our system can help automatically generate large, feature-rich data sets from real-world traffic. While applied to on-demand video streaming as opposed to real-time conferencing, Bronzino et al. presented a system inferring QoE from network traffic using ML [22]; this approach could be accelerated using our methods.

5.9 Conclusion

Zoom is at the forefront of the recent unprecedented surge of video-conferencing traffic. Zoom's proprietary header format and encrypted traffic, however, make it hard for network operators and researchers to understand how Zoom actually operates and performs in the wild. To this end, we demystify Zoom far beyond existing studies by digging deep into its protocol and header format. We show how to extract metrics that closely relate to the quality of a Zoom call, such as media bit rates, frame rate, and frame-level jitter. Our method achieves this by solely inspecting passively collected packet traces, *without any coordination from Zoom clients or servers*. We also create open-source software artifacts to analyze Zoom packet traces, including a Wireshark Zoom plugin [104]. We believe our work paves the way for enabling network operators and researchers to conduct in-depth measurements of Zoom performance in production networks.

Chapter 6

Scalable Video Conferencing Using SDN Principles

Video-conferencing applications face an unwavering surge in traffic, stressing their underlying infrastructure in unprecedented ways. This chapter rethinks the key building block for conferencing infrastructures — selective forwarding units (SFUs). SFUs relay and adapt media streams between participants and, today, run in software on general-purpose servers. Our main insight, discerned from dissecting the operation of production SFU servers, is that SFUs largely mimic traditional packet-processing operations such as dropping and forwarding. Guided by this, we present Scallop, an SDN-inspired SFU that decouples video-conferencing applications into a hardware-based data plane for latency-sensitive and frequent media operations, and a software control plane for the (infrequent) remaining tasks, such as analyzing feedback signals and session management. Scallop is a general design that is suitable for a variety of hardware platforms, including programmable switches and SmartNICs. Our Tofino-based implementation fully supports WebRTC and delivers 7-422× improved scaling over a 32-core commodity server, while reaping performance improvements by cutting forwarding-induced latency by 26×. We also present an implementation of Scallop on the BlueField-3 SmartNIC.

6.1 Introduction

Video-conferencing applications (VCAs) such as Google Meet and Zoom have become essential for remote work, education, and social interactions. The past decade has seen substantial efforts to improve these applications, e.g., via more efficient codecs [54, 139, 184], rate-adaptation algorithms [27, 43, 50, 187], and measurement studies [22, 29, 93, 103]. Though effective, prior work has primarily focused on end users, with the scaling challenges that VCA *operators* face to support exploding traffic rates [52, 96] being far less explored.

At the core of VCA infrastructure are selective forwarding units (SFUs) [60, 75, 91, 102]. These servers are tasked not only with relaying media streams among meeting participants, but also with monitoring and adapting media signals to match the time-varying network capabilities of users. Unfortunately, their deployment on general-purpose servers—the status quo today—makes scaling very difficult, particularly given several fundamental properties of VCAs (Section 6.2):

1. The workload of an SFU is *hard to predict and can change rapidly.* Beyond diurnal variation, the number of streams that an SFU must handle grows quadratically with the number of participants in a meeting, i.e., even a single new participant in a meeting introduces substantial load since their media streams must be relayed to all other participants, and they must receive all media streams from all other participants.

2. And yet, the replication and forwarding that SFUs perform on media packets is on the *latency-critical path of user interactions.* At hundreds of packets/sec./stream, operating-system delays for software packet processing (e.g., scheduling, context switches, interrupts, socket-buffer copying) can lead to significant user-perceived jitter and latency, especially in the face of under-provisioned resources.

As a result, VCA operators are left with two options today: massively over-provision SFU server infrastructure to handle peak loads, which is costly and wasteful, or (reactively)

auto-scale those resources using traditional mechanisms [128, 165], which risks harming QoE for users.

In this chapter, we forego ephemeral SFU scaling enhancements (e.g., improved software packet processing or provisioning mechanisms) in favor of a fundamental rethink of VCA infrastructure that can support long-term traffic forecasts. Guided by our detailed study of production SFUs and real campus VCA traces (Section 6.3), our key insight is that, despite the large semantic gap, the forwarding and adapting of media signals at SFUs—the most frequent tasks that account for the lion's share of scaling overheads—is *strikingly similar to traditional packet-processing tasks*. Indeed, relaying and adapting media signals in today's SFUs can be distilled down to primitives such as packet replication and selective dropping that network hardware is optimized for.

Fueled by this insight, we present **Scallop**[1], a new hardware/software co-designed SFU that is built on top of the WebRTC standard. Drawing inspiration from SDN principles, Scallop decouples SFUs into (1) an efficient data plane that adapts and relays high-volume media streams using line-rate network switching hardware, and (2) a two-tier software control plane that handles the remaining infrequent tasks, e.g., session management, periodic feedback handling, and signaling. The potential benefits are significant, with promises of 7-422× improved packet-processing performance over general-purpose servers at similar cost with fixed per-packet delays to eliminate SFU-induced jitter.

Yet, decoupling SFUs in this manner requires a rethink of their design and introduces three challenges, which we describe next.

First, since not all SFU functions are amenable to processing in hardware, we decouple SFUs into a three-tier architecture that reflects the different time scales and latency requirements of SFU operations as well as the locality of their data. Specifically, (a) session management is complex but occurs at low frequency and benefits from a global view of the system, making it suitable for centralized processing in software; (b) network feedback pro-

[1]Wordplay on "scale up".

cessing and rate estimation occur at medium frequency and are local to each SFU instance, requiring a responsive—but not ultra-low-latency—control loop in software on each Scallop network device; and (c) media forwarding occurs at hundreds of packets per second per stream and critically depends on low latency in the data plane. Decoupling these tasks in practice, however, is challenging while preserving all application semantics and the correctness of the forwarding and adaptation logic. To address this, we present a detailed analysis of the SFU workload and then place all of the many different SFU functions on one or more of these tiers while maximizing performance. Moreover, we describe a series of methods for filtering, routing, and processing the many feedback messages in VCAs to maintain faithful SFU operation in this redesigned architecture.

Second, *Scallop*'s data plane must implement the media-replication and forwarding tasks of an SFU in hardware. Most packet-processing platforms have packet *mirroring* capabilities; however, unlike mirroring where only one packet replica is created, VCAs require as many replicas as there are receivers. To address this mismatch, we present a general but hardware-amenable solution that can be implemented on a wide range of packet-processing platforms and accelerators, including switches, SmartNICs, or eBPF. From this, we derive two platform-optimized designs for programmable switches (here the Intel Tofino2) and SmartNICs (here the NVIDIA BlueField-3). These designs play to the strengths of the respective platforms while accommodating their constraints.

Third, to build a system that is practical and deployable, we follow WebRTC, a widely adopted standard for real-time communication. To do so, we must ensure that our SFU implementation is transparent to unmodified WebRTC clients running in browsers or on mobile devices. We do this in two ways. First, owing to the peer-to-peer (P2P) design of video-conferencing protocols like WebRTC, SFUs traditionally operate as *split proxies* that terminate and spawn new client connections. However, this design would burden the SFU with tasks typically handled by end hosts and unsuited for network hardware (e.g., packet de/re-encryption), ultimately increasing control-plane overheads and reducing scalability.

Instead, we aim to run SFUs as *true proxies* by capitalizing on the observation that most of these functions already run individually at clients and need not be replicated at SFUs. Doing so, while remaining faithful to all SFU semantics, requires a redesign of the way WebRTC establishes sessions and handles feedback signals at each participant. Second, our proxy SFU design must ensure that all traffic (media and control) appears to clients exactly as if it were sent from another client, without intermediate processing. Among other challenges discussed in the chapter, this is especially difficult when adjusting the media rate to adapt to network conditions, which, if done naively, creates sequence gaps that WebRTC receivers interpret as network losses, triggering unnecessary retransmissions. To maintain the P2P illusion, the SFU must rewrite sequence numbers to mask intentional gaps in the stream. However, such rewriting is inherently difficult, especially in the presence of network-induced loss and reordering—even software implementations cannot do this perfectly. Our experiments reveal that leaving extra gaps is preferable to masking legitimate ones: missing sequence numbers trigger packet retransmissions, while incorrect rewrites break the decoder's state, leading to a permanent freeze that can only be recovered by a new key frame, causing severe video freezing. Based on this finding, we design a hardware-friendly sequence-number rewriting heuristic that minimizes retransmissions while preserving stream continuity, even under high loss or reordering. We further discuss possible changes to the WebRTC protocols and reference implementation that would enable more efficient in-network processing of media streams to improve overall system scalability.

We implemented Scallop using P4 on a 12.8 Tbit/s Intel Tofino2 ASIC and a NVIDIA BlueField-3 SmartNIC. In experiments replaying campus-scale Zoom traces, Scallop handles 96.5% of all packets and 99.7% of bytes entirely in the hardware-based data plane. On the Tofino, this enables Scallop to support up to 128,000 concurrent meetings on a single switch, a 7-422× improvement over a 32-core commodity server running existing SFUs [102]. Further, Scallop reduces the latency introduced by SFUs by a factor of 26.8, improving QoE for all participants. We will publicly release our Scallop implementation post publication.

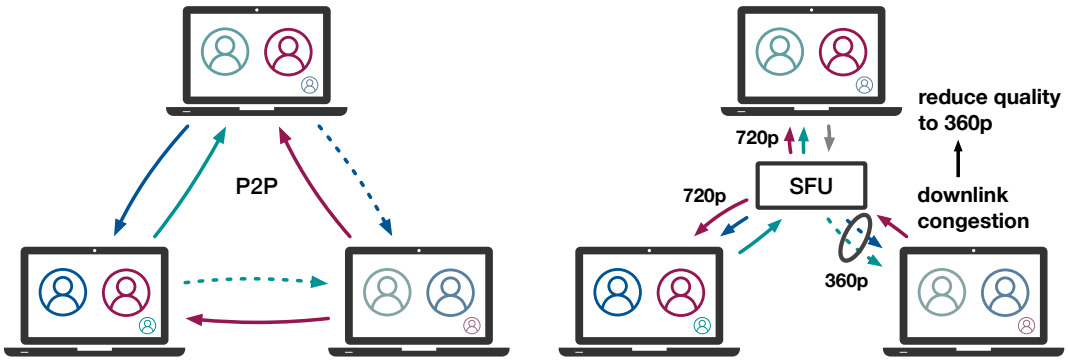


Figure 6.1: VCA architectures: P2P vs. SFU

6.2 SFU Scaling Challenges

6.2.1 Meeting Topologies and SFUs

Need for SFUs. Modern VCAs use SFUs as the de-facto standard to connect participants for two main reasons: First, while P2P connections are possible for meetings with more than two participants, they are impractical due to the need for each participant to encode and send media to every other participant (Figure 6.1, left). This results in significant computational overhead and requires substantial uplink bandwidth which is often unavailable in residential and wireless Internet settings. Second, an intermediate (publicly routable) server solves challenges associated with network address translators (NATs), making SFUs useful even for two-party calls. For example, Google Meet always uses an SFU for two-party calls.

SFU Scaling Properties. While the use of SFUs solves the problem of constrained uplink bandwidth and reduces the required CPU resources at clients, SFUs do not entirely solve the scalability problem in multi-party video conferencing as sometimes suggested [71]. The number of streams required in an SFU scenario still grows quadratically with the number of participants. There are N^2 media streams for N participants (per media type, i.e., video, audio, or screen share) in an SFU topology (see Figure 6.1) which even grows slightly faster compared to the $N(N-1)/2$ streams in a P2P topology. The key difference is that all these streams are now sent to or received by the SFU, effectively moving the bottleneck

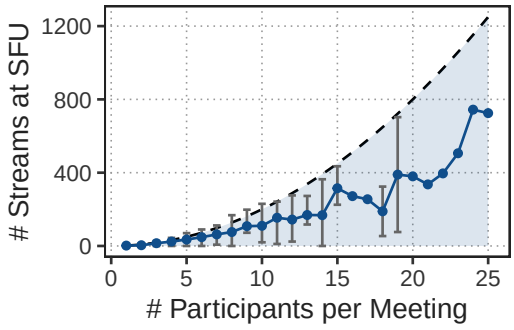


Figure 6.2: Number of media streams per meeting in campus trace.

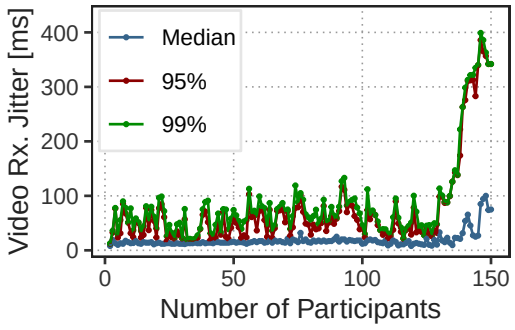


Figure 6.3: Video jitter while adding participants to the SFU.

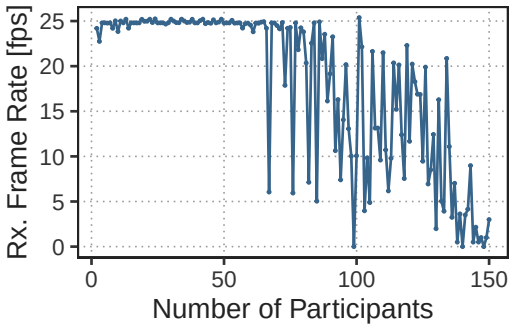


Figure 6.4: Video frame rate while adding participants to the SFU.

from the clients to the SFU. The second important observation is that the amount of work that an SFU needs to perform is determined by the number of media streams and not by the number of participants which therefore leads to quadratic scaling behavior at the SFU. This is due to the fact that every incoming stream needs to be replicated for each of the downstream participants and then sent out. Figure 6.1 shows an example of this with three participants where P2P connections result in a total of six streams while the SFU handles nine streams.

SFU Load in Campus Trace. Of course, meeting participants do not share both audio and video at all times. To get an understanding of the actual number of streams per meeting in a real-world video-conferencing application, we analyzed Zoom usage data taken from the Zoom Account API [189] on our University campus. The data was collected over the course of two weeks in October 2022 and contains 19,704 meetings. Figure 6.2 shows the range

(gray bars) and median number (blue dots) of media streams at the SFU for each meeting as a function of the maximum number of participants within each meeting on our campus. A media stream was counted when it was active for at least 10% of the meeting's duration. The dashed line indicates the upper bound of streams possible if every participant shares both audio and video which can be exceeded in practice when participants also share their screen, as seen in this figure. Even for meetings with 10 participants, the SFU already handles up to 200 media streams. Meetings with 25 participants, a typical classroom size, generate over 700 streams at the SFU in our data set and can theoretically produce up to 1250 streams.

6.2.2 Consequences of Under-Provisioning

SFU Performance Implications. In contrast to signaling and rate adaptation, media distribution is latency-sensitive. SFUs must touch each of these media packets, and as such, any delay introduced by the SFU is added to the end-to-end delay a user experiences, directly impacting QoE. Consequently, it is crucial that the forwarding delay and induced jitter are minimal. However, software packet processing is subject to OS-level delay artifacts due to scheduling, context switches, interrupts, etc., adding to the forwarding delay. At hundreds of packets/sec./stream (typically between 800 and 1400 Bytes in size for video and around 200 Bytes in size for audio), SFU operations require copying significant amounts of data among socket buffers for receiving packets and before sending them out, resulting in high CPU load and frequent context switching. These delays are hard to predict and impair session quality o the point where the VCA becomes unusable.

QoE Degradation under High Load. To confirm the suspected quality impacts of under-provisioning SFUs, we conducted an experiment using the Mediasoup open-source SFU [102] which we deployed on a server with a 40-core Intel Xeon Silver 4114 CPU and 96GB of RAM in our testbed (Section 6.8). A second emulated clients using headless Chrome and was directly connected to the SFU via a 10Gbit/s Ethernet link. We pinned the Mediasoup server to a single CPU and incrementally built up to 15 meetings with 10

participants each, adding one participant every ten seconds. We measured the quality of the first meeting using the WebRTC statistics API [168] as we added more participants to the SFU. The server reached 100% CPU utilization at around 80 participants. Figure 6.3 shows the receive jitter. Tail jitter is high throughout the experiment before exceeding 100 ms, causing significant mouth-to-ear delay and freezes, as depicted in Figure 6.3. Figure 6.4 shows that the video frame rate starts dropping at around 60 participants and making the session effectively unusable beyond 100 participants.

Takeaways. In summary, SFUs share scaling properties (e.g., diurnal usage patterns) with other user-facing services. Additionally, however, they exhibit unique scaling challenges due to the quadratically growing amount of media streams to be forwarded. Dynamics and unpredictability within meetings, for example, due to participants joining or leaving or starting or stopping to share a particular media type (e.g., video, audio, or screen), further exacerbate this problem. At the same time, under-provisioned SFUs can rapidly hit high utilization levels, which have a direct, noticeable, and sometimes prohibitive impact on the session quality. Taken together, VCA operators either vastly over-provision their infrastructure to accommodate such dynamics or they jeopardize QoE.

6.3 SFUs as Packet Processors

Before presenting Scallop, we provide an overview of SFU operations and the key insights guiding our design.

SFU Design Choices. As opposed to earlier generations of intermediate servers, SFUs do not mix or transcode media streams but instead operate on *packetized* media. WebRTC is the only widely adopted standard and open-source framework for video conferencing, yet it does not provide any guidance on how SFUs should be implemented or how they should handle RTP streams. The simplest way to implement an SFU is to maintain separate P2P connections and distinct RTP streams between the SFU and each participant. This

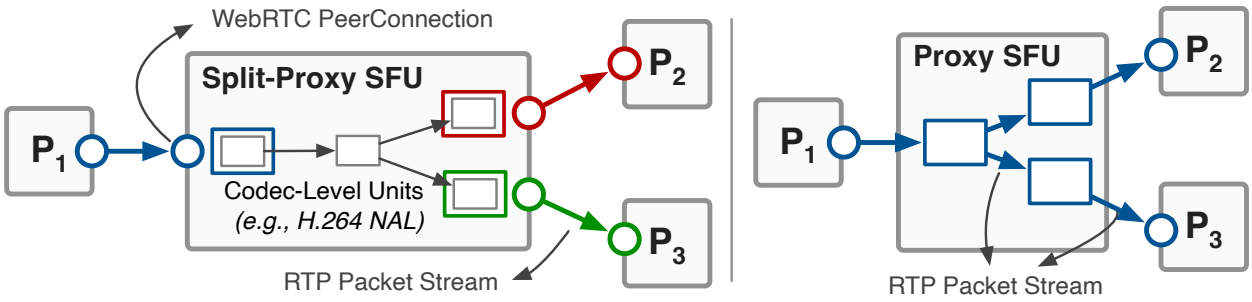


Figure 6.5: SFU design choices.

approach is similar to a *split-proxy* design, a term we will use going forward and is illustrated on the left side of Figure 6.5. Existing WebRTC SFUs (e.g., MediaSoup [102]) operate in this way.

Alternative Approaches. In contrast to the aforementioned *split-proxy* design, we design Scallop using a powerful insight that makes the design of SFUs computationally simpler: SFU functionality can be accomplished by replicating packets and rewriting header fields. These tasks are supported by traditional packet processors (e.g., switches) and do not require software processing. We look for evidence of such a design in the real-world by collecting and analyzing packet-level traces of meetings on our campus. In accordance with previous studies [65, 103], we find that the SFU servers used by Zoom—one of the market leaders in video conferencing—send exact copies (except for rewritten IP addresses and port numbers) of the incoming RTP packets to all downstream participants. While we do not know Zoom's full architecture or how their SFUs are implemented, these findings lend credibility to Scallop's design.

Rate Adaptation in Zoom's SFUs. Rate adaptation becomes necessary when the network conditions of a participant change, e.g., due to network congestion or if it is not necessary to forward high-resolution video to a participant due to device characteristics, such as a smaller screen on a cell phone. Without rate adaptation at the SFU, media senders would all have to reduce their sending rate to relieve congestion, resulting in lower quality for all participants, even those unaffected by congestion (see Section 6.5.3). Realizing

this functionality (without transcoding media) requires using a scalable media stream, for example using Scalable Video Coding (SVC). In SVC, video is encoded in multiple layers, each with a different bitrate. The media stream is packetized so that a given packet always belongs to exactly one layer. As a result, reducing the media resolution or frame rate can be achieved by dropping a specific subset of packets. This insight is the second foundation of our design. Our analysis of Zoom's SFU reveals that it uses a combination of Simulcast and SVC to achieve rate adaptation, providing further credibility to our design.

Takeaways. Taken together, the core work of SFUs can be implemented by replicating packets and sending copies out to all receiving participants. Furthermore, if SVC is used, adapting a media stream reduces to forwarding a clearly defined subset of packets to a given participant.

6.4 Introducing Scallop

Based on the insights from Section 6.3, we introduce Scallop, a novel SFU design leveraging SDN principles and programmable networking hardware to improve the scalability and performance of video-conferencing infrastructure. Scallop offloads all media replication, forwarding, and rate-adaptation tasks to high-speed hardware, yet several operations SFUs perform are not amenable for such an implementation. Consequently, we require a split of functionality where we leave as many tasks as possible inside the data plane and only carefully leave operations in software when absolutely necessary.

A Taxonomy of SFU Operations. Scallop's control/data-plane split is driven by the latency requirements, computational complexity, and frequency of the tasks an SFU performs. Along these axes, we classify SFU tasks into 3 categories as shown on the left side of Figure 6.6: (1) infrequent tasks that are not latency-sensitive, including session management and signaling; (2) latency-sensitive tasks (on the order of tens of milliseconds), including deciding the correct target sending rate of a media stream based on feedback signals, as well

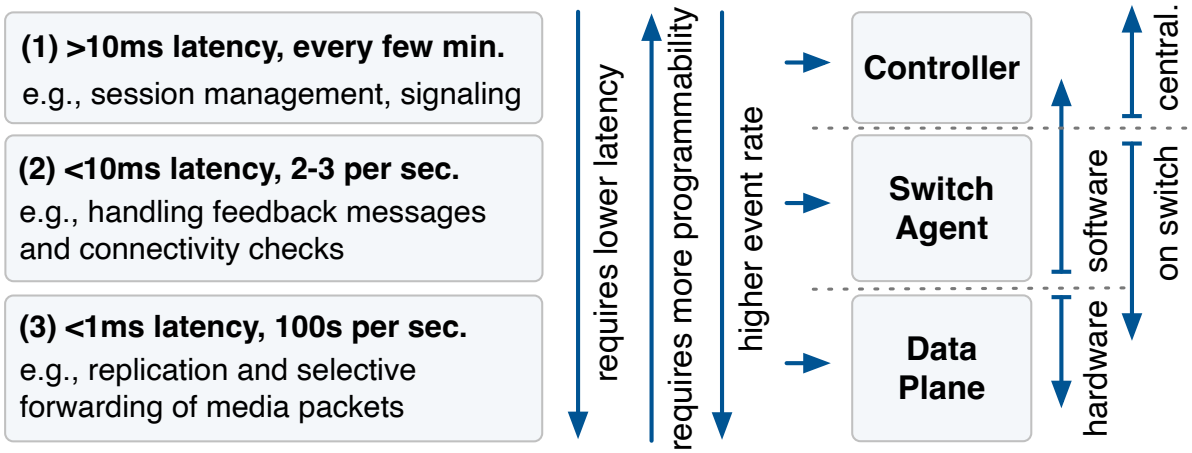


Figure 6.6: Latency and programmability requirements of key SFU responsibilities and resulting placement.

as handling connectivity checks performed by the STUN protocol [97]; (3) ultra-low-latency tasks (sub-millisecond), including the actual handling of media packets, i.e., forwarding and dropping them if necessary.

Three-Tier SFU Design. The resulting architecture (right side of Figure 6.6) is inspired by the design of SDN systems but goes a step further by introducing a third plane between a centralized controller and the switch data plane for the aforementioned latency-sensitive tasks. Here, the central controller is only involved when (1) a new session is created, (2) a participant joins or leaves a meeting, or (3) a participant starts or stops sharing a particular media type (i.e., audio, video, or screen). The controller is centralized as it is designed to manage multiple distributed Scallop instances to enable more efficient forwarding topologies (Section 6.9). Finally, the data plane handles truly latency-sensitive tasks on the critical path for QoE.

Scallop's Switch Agent. All remaining tasks that are either (1) not amenable to implementation in the data plane due to their complexity, or (2) fall in the category of latency-sensitive tasks that not on the critical path for QoE, run in software directly on the switch (i.e., the switch's CPU and operating system). This, for example, includes processing feedback signals that are subsequently used for rate adaptation. We call this

intermediate component the *switch agent*. Importantly, the switch agent is not involved in media forwarding, and none of the above-mentioned tasks require any media or feedback to be sent back from the switch agent; rather, the switch agent only receives copies of packets, analyzes them, and reconfigures the data plane if required. Its location on the switch enables a low-latency control loop.

Deployment Setting. Today, production-scale SFUs are deployed on data-center servers of cloud providers. Increasingly, these servers are connected using *programmable* top-of-the-rack switches. Besides, these servers are often equipped with SmartNICs to offload networking tasks. While *Scallop* is not tied to a data-center setting, data-center switches, and SmartNICs are natural and readily available deployment settings for *Scallop*. Going further, switches implementing *Scallop* can entirely replace multiple SFU servers, increasing traffic served per rack at comparable operating costs.

Support on Packet Processors. Modern programmable switches have higher capacity than SmartNICs (Tbps vs. hundreds of Gbps) and more advanced capabilities (e.g., scalable replication, register arrays) in the hardware data plane. *Scallop* requires three main capabilities in the hardware: deep header parsing, scalable replication (Section 6.6), and stateful sequence-number rewriting (Section 6.7). Consequently, *Scallop*'s potential can be fully realized on switches such as the Intel Tofino2, which is our evaluation platform (Section 6.8). We also implement *Scallop* on the hardware pipeline of a representative SmartNIC (NVIDIA BlueField-3).

6.5 Control-Plane Prototype

Scallop's control plane must handle two key tasks: (1) establishing and managing WebRTC sessions between participants such that the SFU is inserted as a proxy between them and (2) handling the infrequent but important remaining tasks that cannot run in the data plane. The first set of tasks are handled by a centralized controller (Section 6.5.1) while the second

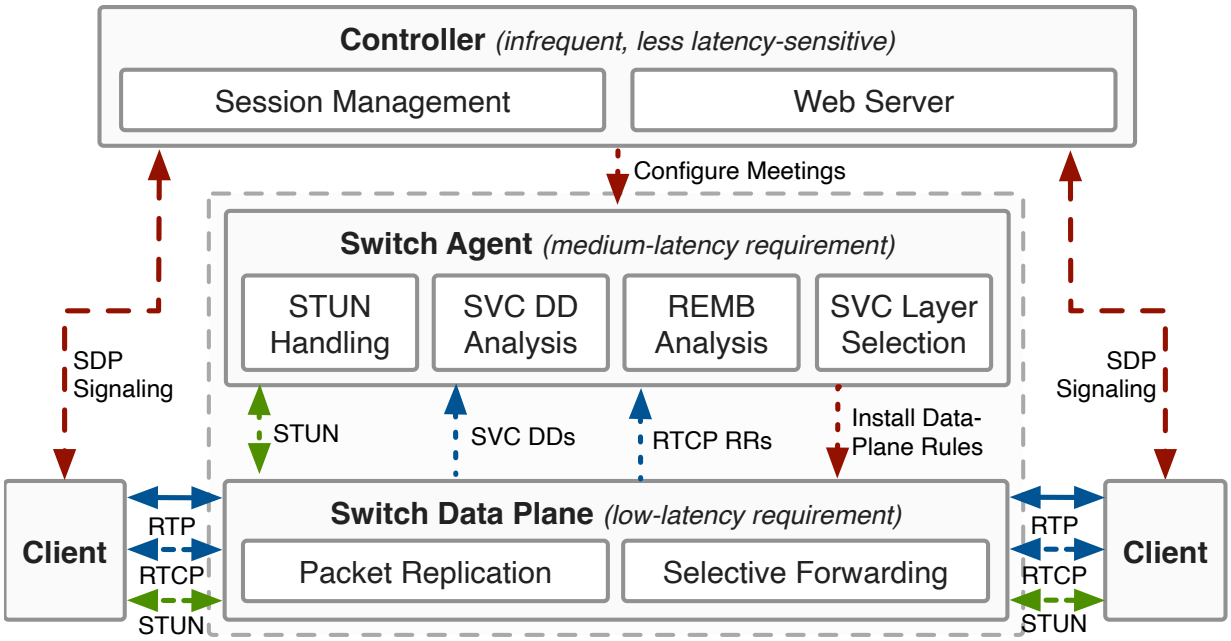


Figure 6.7: Scallop's 3-tiered architecture

set of tasks are handled by the switch agent, a lightweight control program on the switch CPU (Section 6.5.2-6.5.5). Both applications are written in C++ and communicate with each other using remote procedure calls. The switch agent can additionally exchange packets with the data plane via the switch CPU port. Figure 6.7 shows the resulting architecture.

6.5.1 Session and Connectivity Management

The core task of the central controller is to manage WebRTC sessions and connectivity between participants such that all media traffic is sent to and can be received from Scallop as opposed to another client, realizing our proxy architecture.

WebRTC Signaling. WebRTC uses the *Session Description Protocol (SDP)* to negotiate media-session parameters between participants, including codecs, their parameters, and IP addresses and ports [110, 138]. This negotiation, known as *signaling*, is initiated by participants whenever a new media stream is created. Scallop's controller acts as the signaling server that exchanges SDP messages between participants and maintains state about participants and their media streams to correctly configure the data plane.

Controlling Signaling to Create Proxy Topology. Each SDP message includes a list of *connection candidates*, which convey the IP addresses and ports RTP media is being sent from or can be received at. Using this information, we can *insert* the SFU as an intermediate entity between participants while appearing to meeting participants as their sole peer. *Scallop* achieves this by intercepting SDP messages and modifying the connection candidates on the fly.

STUN and Connectivity Management. WebRTC uses periodic *Session Traversal Utilities for NAT (STUN)* [97] packets to continuously check reachability between participants. Scallop handles STUN in the switch agent as processing STUN packets is too complex for the data plane due to their header format. This is fine since STUN packets are not classified as *latency critical*. A connection is only deemed interrupted after multiple consecutive connectivity checks fail.

6.5.2 Bandwidth Estimation

Rate Adaptation in SFU Architectures. Continuous and timely bandwidth estimation and rate adaptation are critical tasks in video conferencing as trying to send media at a higher bitrate than the network can support rapidly leads to high latency and, ultimately, loss, severely degrading QoE. When an SFU is used, this task is split into two parts: (1) each sender sends at the highest rate any of the receivers can receive at (Section 6.5.2-Section 6.5.3), and (2) for receivers that can only support lower rates than the highest rate, the SFU scales down the stream by selectively dropping packets (Section 6.5.4). WebRTC uses Google Congestion Control (GCC) [27] to estimate link capacity, which is then used to adjust media bitrate.

GCC Modes. GCC can operate in two ways. In sender-driven mode, the sender computes available bandwidth based on *frequent feedback* from receivers. Scallop uses receiver-driven

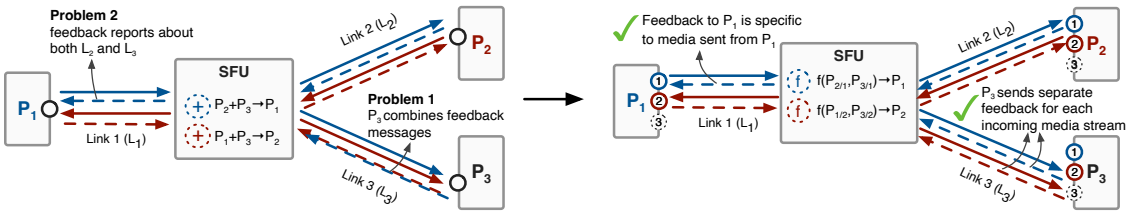


Figure 6.8: Splitting WebRTC connections per participant and forwarding feedback messages of the best-performing downlink only preserves feedback semantics and ensures effective rate adaptation.

mode where each receiver independently estimates bandwidth based on packet arrival-time variation and then *periodically* informs the sender of the new bandwidth estimate using *Receiver-Estimated Bandwidth (REMB)* messages. Although REMB messages are still too complicated for the data plane to process, the frequency of them is proportional to the frequency of the link capacity changing, unlike in sender-driven mode, where the receiver sends a message every 10-20 media packets. Scallop intercepts REMB messages and adapts media bit rates based on the reported estimates.

6.5.3 Preserving Feedback Semantics

Bandwidth Estimation in a Proxy Architecture. Realizing correct bandwidth estimation in our proxy-based Scallop architecture, however, is challenging. This is because forwarding feedback among participants, rather than using individual control loops (as in a split proxy), mixes signals, incorporating uplink measurements from all senders instead of reflecting a specific downlink. As a result, feedback converges on the lowest-bandwidth receiver, forcing the sender to lower the media bitrate unnecessarily for all participants.

Mixed Feedback Signals. To illustrate the problem, consider a 3-party meeting as depicted in Figure 6.8. Participant 1 (P_1, blue) and 2 (P_2, red) share video while participant 3 (P_3) only receives. Solid lines depict media streams and dashed lines depict Real-Time Transport Control Protocol (RTCP) feedback messages. Done naively, two problems arise. First, RTCP combines feedback messages concerning multiple media streams into a single

RTCP packet which is too complex to parse, analyze, and correctly forward in the data plane. Second, they also mix feedback from different media streams in a way that feedback is not actionable anymore. For example, the bandwidth estimated by P_3 now is based on feedback from both P_1's and P_2's uplinks (to SFU) in addition to P_3's downlink (from SFU). This estimate is irrelevant for the SFU since it cannot do anything about uplink capacity. More importantly, it leads to the problem that when naively forwarding the combination of all feedback to P_1, P_1's media will be encoded at a bitrate adapted to P_2's uplink performance. Put differently, all send rates will converge to the lowest capacity of all uplinks in the meeting which defeats the purpose of even having rate adaptation in the SFU.

Solution Part 1: Split Connections. We solve these problems using two techniques (Figure 6.8). First, we split each WebRTC UDP stream (the scope for bandwidth estimation) by participant, such that every receiver's RTCP feedback pertains to exactly one sender. This eliminates the need to parse and filter combined RTCP messages carrying reports for multiple streams—Scallop simply forwards each receiver's feedback directly to its corresponding sender.

Solution Part 2: Selectively Forward Feedback. In our example, P_3 now has a dedicated stream to receive media from P_1 and a second stream to receive media from P_2. Consequently, P_3's REMB messages only include information about the path between P_3 and P_1 or P_2, respectively, and not about the combination of all paths. A filter function f at the SFU selects the best-performing downlink per sender and configures the data plane to only forward these messages to the respective sender. The selection is done by computing an EWMA over each receiver's bandwidth estimates and periodically selecting the maximum out of the EWMAs. The rationale for this is that each sender should send at the highest rate allowed by its uplink and the best downlink. As all packets traverse the sender's uplink, its

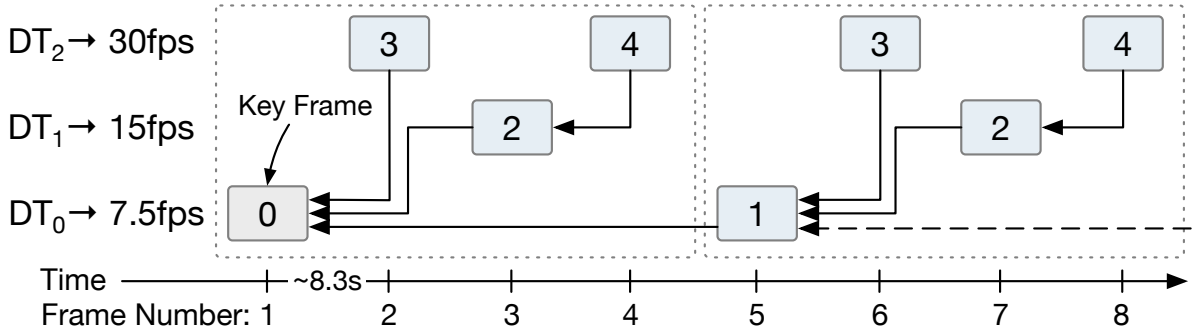


Figure 6.9: Frame dependencies in AV1 L_1T_3 SVC.

performance is inherently accounted for in the feedback. The SFU then handles adaptation for lower-bandwidth downlinks, explained next.

6.5.4 SVC Analysis and Layer Selection

Scalable Video Coding (SVC). *Scallop* leverages *SVC* [139] with the AV1 codec's L_1T_3 profile [44, 160] to adapt media streams to network capacity. This allows the SFU to choose among three temporal layers (frame rates) for video streams. While the data plane handles the actual forwarding, the control plane decides the quality level to send to each participant. The SVC structure can change with each key frame (sent when a stream starts or the resolution changes), requiring the SFU to adapt the data plane's forwarding rules accordingly. Figure 6.9 shows an L_1T_3 SVC stream's dependencies.

AV1 Extension and Dependency Descriptor. In AV1, each RTP packet contains an RTP AV1 extension header indicating its layer through a unique *template id*. Key frames additionally contain a *dependency descriptor* that carries the semantics of the SVC structure [160]. In our example, template ids 0 and 1 represent the base layer (7.5 fps), id 2 the first enhancement layer (15 fps), and ids 3 and 4 the second enhancement layer (30 fps). Dropping frame ids 3 and 4 would reduce the frame rate from 30 fps to 15 fps. The data plane parses the AV1 extension header (primarily its template id) to decide whether the packet should be dropped or not.

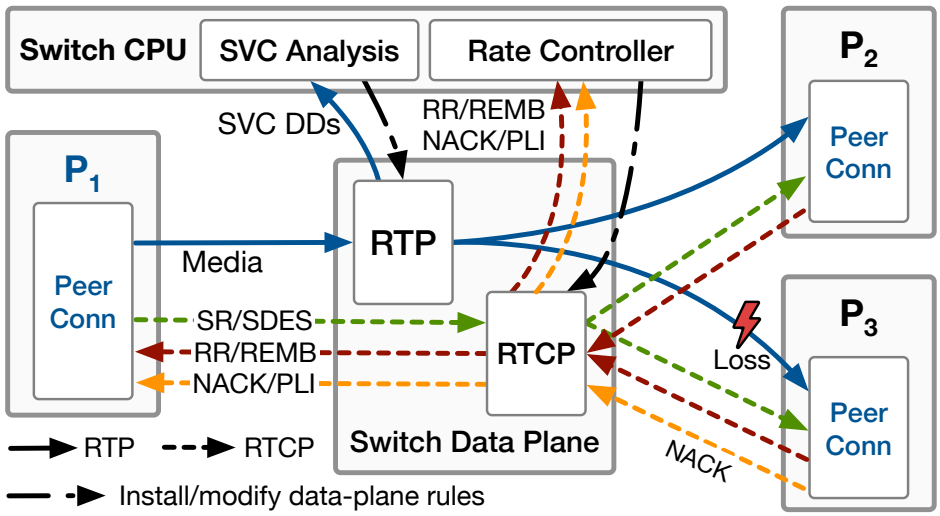


Figure 6.10: Flow of all types of media and control packets in a Scallop 3-party conference.

Selecting a Quality Layer. Parsing the dependency descriptor is beyond the data plane's capabilities. Thus, *Scallop* sends key frames to the control plane for analysis. Whenever a bandwidth estimate (i.e., an REMB message) is received, the switch agent invokes a function that can be defined by adopters of Scallop. This function is declared as follows:

`selectDecodeTarget(currDT, estHist, newEst)` $\rightarrow$ `newDT`.

The function takes as input the current decode target (currDT), a history of past estimates (estHist), and the new estimate (newEst) from the REMB message; it returns the new decode target. If the returned decode target is different from the previous one, the switch agent reconfigures the data plane. Importantly, while we implemented a simple heuristic that switches between quality levels based on fixed capacity-estimate thresholds, using this model, arbitrary rate-adaptation algorithms can be implemented.

6.5.5 Handling other RTCP Messages

Besides REMB, additional feedback messages are delivered through RTCP. From the receiver side, negative acknowledgments (NACKs) request the retx. of a specific media packet, and picture-loss indication messages (PLI) notify the sender to send an intra-coded video frame. NACK and PLI messages are sent from a receiver experiencing loss to the respective sender

(Figure 6.10). While PLIs, NACKs, and REMB messages are forwarded to their intended destinations through the data plane without delay, the data plane also creates copies of them on-the-fly, sending them to the switch CPU for further analysis (e.g., for the filter function described earlier).

6.6 Scalable Media Replication

Background and Challenges. *Scallop* must enable scalable replication of media packets in hardware to support thousands of simultaneous meetings. This entails two core challenges: First, for each meeting, *Scallop* stores both the list of active participants and each participant's rate-adaptation status. On every incoming media packet, this state determines which downstream recipients receive a replica. Since memory is limited on packet-processing hardware, supporting a large number of concurrent meetings becomes difficult. Second, once the recipients are known, *Scallop* must generate and forward replicas to them efficiently. Although modern packet-processing platforms offer hardware-assisted replication, their features vary widely, and no prior work demonstrates how to map SFU packet replication onto them. *Scallop* addresses these challenges in two parts. First, we introduce a general, memory-efficient algorithm for SFU replication on hardware packet-processing platforms. Second, we adapt and optimize this algorithm for two representative platforms—programmable switches (e.g., Intel Tofino2) and SmartNICs (e.g., NVIDIA BlueField-3).

6.6.1 General, Memory-Efficient Replication

For our platform-agnostic design, we rely on three basic primitives available on all programmable packet-processing platforms: match-action (MA) tables populated by the control plane (in *Scallop*, the switch agent), a replication (mirroring) function that produces a single copy of an input packet, and a recirculation function that feeds packets back into the ingress pipeline for further iterations. Our replication algorithm then proceeds in two parts: (1)

Algorithm 1: General Packet Replication in *Scallop* (RA-driven logic highlighted in green)

```
1  Sender2Meeting: Map{Participant: Meeting}
2  Meeting2Participants: Map{{Meeting, ReplicaID}: Participant}
3  Template2Layer: Map{{Meeting, Template}: Layer}
4  ExcludedReceivers: Map{{Meeting, Layer, Participant}: Bool}
5  ExcludedSenderReceivers: Map{{Meeting, Layer, Participant, Participant}: Bool}
6  Function scallop_replicate (Packet pkt):
7      Participant sender := {pkt.ip.src, pkt.udp.src, pkt.rtp.ssrc}
8      Meeting meeting := Sender2Meeting[sender]
9      Layer layer := Template2Layer[{meeting, pkt.av1.template_id}]
10     ReplicaID replica_id := 1
11     while Meeting2Participants[meeting, replica_id].hit do
12         receiver := Meeting2Participants[meeting, replica_id]
13         if receiver != sender then
14             if ExcludedReceivers[{meeting, layer, receiver}].miss &&
                  ExcludedSenderReceivers[{meeting, layer, sender, receiver}].miss then
15                 Packet pkt2 := replicate(pkt)
16                 send_packet_to(pkt2, receiver)
17         replica_id := replica_id + 1
```

Participant-driven, which replicates each packet according to the meeting-to-participants mapping, and (2) *Rate Adaptation (RA)-driven*, which excludes replication for a subset of rate-adapted participants.

Solution Part 1: Participant-Driven Replication. Consider a meeting M with N participants. *Scallop* must replicate each incoming packet from any participant (sender) to the other N–1 participants (receivers). In a naive approach, we could assign a separate *replica ID* (RID) to every sender-receiver pair in M and store them as MA-table records of the form: key={sender, RID}, value={receiver}. When a packet arrives from a sender, we create a replica for its receiver with RID=1, recirculate the packet and create a replica for its receiver with RID=2, and so on. While this technique works, it requires $N(N–1)$ or $\mathcal{O}(N^2)$ records per meeting. A more efficient strategy (outlined in Algorithm 1) is to assign a global replica ID to each participant in M (line 2). This reduces the memory requirement to N records ($\mathcal{O}(N)$). However, now when *Scallop* iterates over the replica IDs (lines 1, 7-8), it would encounter the sender itself. Fortunately, the sender is trivially known from

the packet headers (line 7), so we skip that iteration (line 13). Once created, the replica is addressed to the receiver's IP and UDP port, and forwarded (line 16). Since this logic is sufficient for meetings not subject to RA, we also call it the *non-rate-adapted (NRA)* technique.

Solution Part 2: RA-Driven Replication. In some meetings, individual receivers might have lower receive bandwidth than other participants, requiring RA. To support RA, *Scallop* must replicate only the necessary quality layers for each affected receiver. For example, if a receiver can handle only 7.5 fps instead of the full 30 fps, *Scallop* should replicate the base layer but omit the two enhancement layers (see Section 6.5.4). *Scallop* identifies each packet's quality layer by looking up its AV1 *template ID* in an MA table, which uses constant memory per meeting (lines 3, 9). Once the layer is known, *Scallop* must determine which participants should receive replicas. Because RA is often not required, we record only the *excluded* receivers rather than the included ones. Exclusions fall into two categories: (1) *Receiver-driven RA (RA-R)*, where a receiver receives the same reduced quality (e.g., 7.5 fps) from all senders, and (2) *Sender–Receiver-driven RA (RA-SR)*, where a receiver receives different qualities from different senders. RA-SR happens when some senders send media at a higher bitrate than others. *Scallop* implements exclusions using two disjoint MA tables for RA-R and RA-SR (lines 4-5). In the worst case, RA-R consumes $\mathcal{O}(N)$ records and RA-SR consumes $\mathcal{O}(N^2)$, respectively, although actual usage is much lower. During each iteration, *Scallop* checks both tables and skips any excluded receiver before replicating the packet (line 14).

Optimizing Bandwidth and Latency. Our general solution runs entirely in hardware and is memory-efficient—a significant improvement over software SFUs. However, the *while* loop in Algorithm 1 (lines 11–17) has two drawbacks. First, it recirculates the original packet N–1 times, consuming recirculation capacity. Second, it produces the final replica only after N–1 recirculations, incurring $\mathcal{O}(N)$ latency. Fortunately, some packet-processing platforms, such as Intel Tofino and Juniper Trio [176] programmable switches, provide hardware primitives

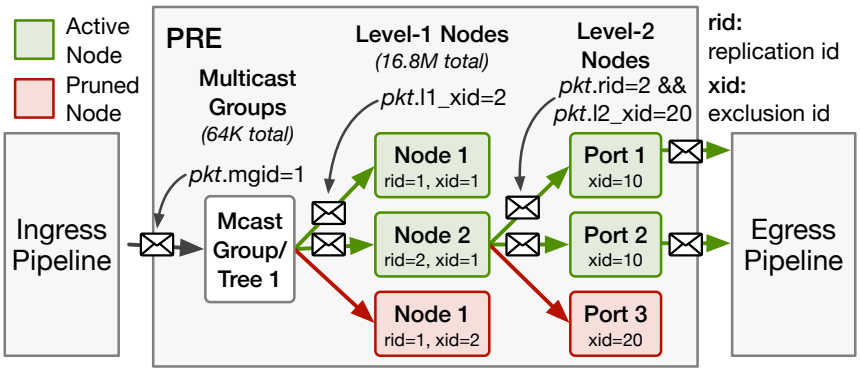


Figure 6.11: Tofino's Packet Replication Engine (PRE)

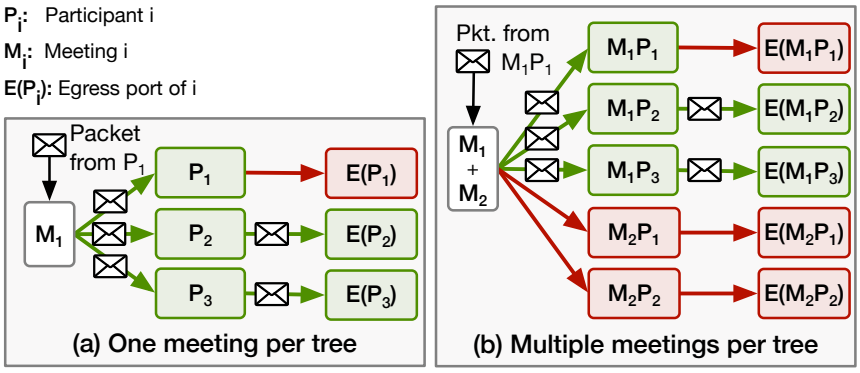


Figure 6.12: Constructing efficient replication trees by aggregating meetings and using dynamic pruning.

to generate hundreds of packet replicas at once. In Section 6.6.2, we describe how we leverage one such platform (Intel Tofino2) to address both drawbacks. On platforms without bulk-replication support (e.g., NVIDIA BlueField-3), we replace the while loop with a recursive, perfect binary-tree routine that produces all replicas within $\lceil \log_2 N \rceil$ iterations and therefore a uniform $\mathcal{O}(\log N)$ latency (Section 6.6.3).

6.6.2 Scalable replication on the Tofino2

Background. The Tofino *Packet Replication Engine* (PRE) is a specialized hardware module designed for efficient multicast through a hierarchical, three-level structure called a *multicast group* or a *multicast tree* (Figure 6.11). The PRE sits in the *Traffic Manager* between the ingress and the egress pipelines of the switch, allowing the ingress pipeline to invoke replication on a packet, the PRE to perform the replication, and the egress pipeline to process

each replica before forwarding it. The control plane configures the PRE at runtime in three steps: (1) Allocate level-2 (L2) nodes, each associated with one egress port ; (2) Allocate level-1 (L1) nodes, each identified by a *node ID* (unique across the PRE), a *replica ID* or RID (unique across a multicast tree), and a *set* of allocated L2 nodes; (3) Create multicast trees, each with a unique *multicast group ID* (MGID) and an associated set of allocated L1 nodes.

Replication: The ingress pipeline sets a packet's *mgid* metadata field to map it to an existing multicast tree. When the packet arrives at the *root* of that tree, the PRE stores the packet in a buffer and creates a *pointer*. Then, it replicates this pointer to L1 nodes, and further to L2 nodes. At each L2 node, a replica is created from the pointer, attached to the associated egress port, and forwarded to the egress pipeline.

Pruning: The PRE also supports branch-*pruning*. The control plane configures pruning by assigning an L1 XID to each L1 node, and an L2 XID to each egress port. The data plane's ingress invokes pruning by setting the packet's *l1_xid* (for L1-pruning), and *rid* and *l2_xid* (for L2-pruning) metadata.

Opportunity and Challenges. Due to its specialized design, the PRE neither requires recirculations nor does it pass around packets. Consequently, it can replicate a packet hundreds of times with minimal latency. *Scallop* can benefit greatly from the PRE, but it must address two challenges. First, mapping VCA entities (meetings, senders, receivers) onto the PRE's tree hierarchy (root, L1 nodes, L2 nodes) is not obvious. No prior work has used the PRE (on Tofino or similar switches) for such purposes, requiring us to explore the design space from the ground up. Second, the PRE comes with resource constraints. While it can support up to 2^{24} L1 nodes, 2^{16} RIDs per tree, and all of the switch's egress ports per L1 node, it can only support $\mathcal{T}$=64K multicast trees [3]. Furthermore, only one L1-XID and L2-XID can be set per packet, limiting pruning flexibility. *Scallop* must adapt its solution to the PRE and maximize its utilization under these constraints.

Two-Party Meetings. Meetings with only two participants (60% in our campus dataset) do not require replication. Therefore, we do not allocate multicast trees for them, thereby saving PRE resources.

NRA Design. For meetings with $N > 2$ and not requiring RA, *Scallop* aggregates all N meeting participants into a single replication tree (Fig. 6.12a). The root represents the entire meeting, each L1 node represents a participant and each L2 node maps a participant to their egress port. This design supports $\mathcal{T}$ concurrent meetings and consumes $\mathcal{O}(N)$ L1 nodes. We use L2-pruning to prevent the sender from receiving its own packet. To further increase efficiency, *Scallop* aggregates multiple (m) meetings into a single replication tree (Fig. 6.12b), supporting $m\mathcal{T}$ concurrent meetings. However, in addition to L2-pruning, this approach requires L1-pruning to ensure that packets of one meeting are not received by participants of another. For example, in Figure 6.12b, replication to $M2$'s participants is suppressed for a packet from $M1$, and vice-versa. The PRE's pruning limitations force m to be 2. *Scallop* can support 128K concurrent meetings using this design (Section 6.8.2).

RA Design. Since both L1- and L2-pruning are consumed in the NRA design, we need alternatives to handle RA. For RA-R, *Scallop* creates one tree (following NRA design) per SVC layer. When a rate-adapted receiver should not receive a particular layer, it is removed from that layer's tree. With q=3 SVC layers, *Scallop* can support up to $m\mathcal{T}/q$ concurrent meetings, which evaluates to 42.7K (Section 6.8.2). For RA-SR, *Scallop* cannot do better than aggregating two senders (and their receivers) per SVC layer into a single tree. *Scallop* consumes $2\mathcal{T}$ L1 nodes to support $2\mathcal{T}/qN$ concurrent RA-SR meetings, which evaluates to 4.3K, compared to 192 on a 32-core server (assuming 10-party meetings, all sending video and audio).

Dynamic Migration across Designs. Since RA (especially RA-SR) is not always required for a meeting, *Scallop* dynamically and seamlessly migrates each meeting across two-party, NRA, RA-R, and RA-SR designs as needed. This maximizes the PRE's utilization, ensuring

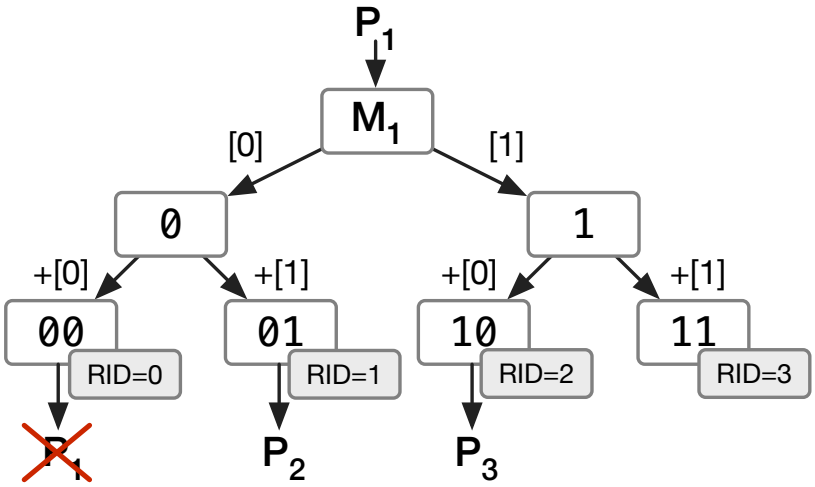


Figure 6.13: Packet replication using a perfect binary tree.

the bottleneck remains the switch's bandwidth rather than its replication capacity, whenever possible.

6.6.3 Scalable Replication on the Bluefield-3

Unlike Tofino2, some packet-processing platforms, such as the BlueField-3 SmartNIC, cannot replicate in bulk and require recirculation to create multiple replicas. However, we can improve the replication latency on such platforms using a perfect binary tree design (as shown in Figure 6.13). During meeting setup, the switch agent populates three MA tables: one mapping each participant to its meeting ID, another mapping every (meeting ID, replica ID) pair back to a participant, and a third recording, for each meeting, the *maximum tree depth* $\lceil \log_2 N \rceil$ (where N is the number of participants). This depth ensures the tree will have at least N leaves.

Figure 6.13 shows an example meeting with N=3 (and so, maximum depth = 2). When a media packet from sender P_1 arrives in the data plane, we encode two fields in its header: a running replica ID (RID) and the current depth, both initialized to zero. The SmartNIC hardware mirrors the packet, then recirculates both copies. Conceptually, one copy traverses the left subtree and the other the right. Each time a mirrored packet returns, we append

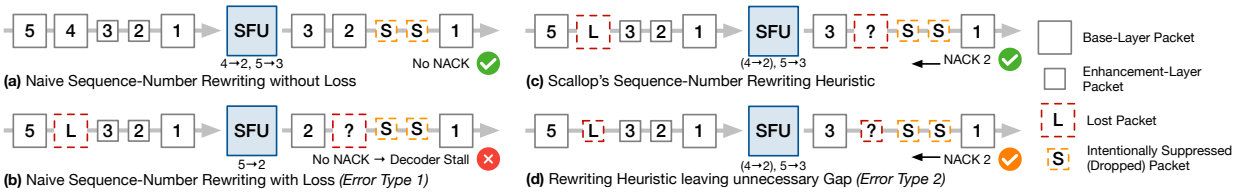


Figure 6.14: Example of sequence-number rewriting and associated error types in Scallop.

the bit value 0 (for left) or 1 (for right) to its RID, increment its depth, mirror it again, and recirculate. This process repeats until the packet's depth equals the precomputed maximum depth for that meeting.

At the end, each packet carries a full-length RID. In our example, we end up with RIDs 0 (00), 1 (01), 2 (10), and 3 (11). RID 0 corresponds to the sender, and RID 3 has no matching participant, so both are dropped. The remaining two replicas with RIDs 1 and 2 map to P_2 and P_3 respectively, and are forwarded accordingly. We then layer our RA schemes (RA-R and RA-SR) on top of this replication mechanism, following the logic described in Algorithm 1. This method can be further optimized by preemptively checking whether a downstream subtree will yield valid replicas and skipping its mirroring if not; we leave this as future work.

6.7 Transparent Rate Adaptation

Background and Challenges. Under WebRTC's P2P model in a split-proxy architecture, each receiver expects a continuous, unmodified media stream from the SFU. However, in our true-proxy architecture, *Scallop* performs RA by suppressing packets associated with a specific quality layer to match network conditions. If not handled explicitly, this suppression would create gaps in RTP sequence numbers, which receivers would interpret as packet losses, triggering unnecessary retransmissions (retxs.). To preserve a continuous RTP packet stream, *Scallop* rewrites sequence numbers after replication. However, rewriting sequence numbers *perfectly*—when intentional suppression coincides with network-induced loss and reordering—is impossible without buffering packets.

Naive Solution. To illustrate, consider the example in Figure 6.14. Here, the sender sends packets with sequence numbers 1 to 5. Packets 1, 4, and 5 are base-layer packets while packets 2 and 3 are enhancement-layer packets to be suppressed. In a straw-man design (Fig. 6.14a), *Scallop* would simply increment a counter every time it sends out a packet to a receiver. This approach would preserve a continuous stream of sequence numbers and the receiver would not trigger negative acknowledgments (NACKs) for missing packets, thus avoiding retx. of intentionally suppressed packets. This design, however, would also mask network-induced packet losses, leading to the receiver's decoder state breaking and the video freezing until a picture-loss indication (PLI) to reset the decoder is triggered (Fig. 6.14b). This situation, which we refer to as an error of *type 1*, would lead to an unacceptable stall for the affected receiver.

Acceptable Heuristic Error. Instead, we design a hardware-amenable heuristic that uses a combination of per-meeting, per-stream, and per-packet data to infer whether a sequence number skipped due to a loss corresponds to a packet that was supposed to be suppressed anyway. If so, *Scallop* increments the sequence number by one. If the packet belongs to a base layer (required for the decoder), *Scallop* will preserve the sequence gap to trigger a NACK and retx. (Fig. 6.14c). There are, however, cases where *no* online algorithm for this problem can exactly determine which layer the packet with the skipped sequence number belongs to. Consider the last case (Fig. 6.14d) where packet 4 carries the enhancement layer which is supposed to be suppressed. Assuming *Scallop* cannot infer that 4 is a candidate for suppression, it preserves the sequence-number gap. This leads to a NACK for packet 2 (originally packet 4), causing its retx. which is unnecessary. Our experiments show that, while this retx. wastes some bandwidth, it does not cause any QoE degradation. Therefore, this error, which we call error *type 2*, is acceptable but slightly expensive. We design our heuristic to *never* commit type-1 errors, and to *minimize* type-2 errors. We find that the overhead of unnecessary retxs. from type-2 errors grows with the loss rate but is low even

at significant loss (e.g., under 5% overhead at 10% loss).

Heuristic Design. To rewrite sequence numbers correctly, *Scallop* must determine an accurate *offset* (difference between original and rewritten sequence numbers) for each replicated packet in each rate-adapted sender-receiver media stream. This is challenging because the rewriting must be done post-replication, i.e., in the egress pipeline, where suppressed packets are never "seen". The task is further complicated by network loss and reordering, as described above. Our main insight is that the offset's accuracy depends on how accurately we reconstruct the state of the media frame the replica belongs to, and the one immediately preceding it. For this, we rely on (1) per-packet state: current frame number, sequence number, frame start and end markers from headers; (2) per-stream state: the offset, max. frame number seen, max. sequence number seen of the max. frame, a flag indicating whether the max. frame's end marker was seen, the *cadence* of frames suppressed by RA (e.g., every second frame) populated by the control plane, and a history of the offsets for the last few unsuppressed frames. For each incoming packet, first, we check whether the offset should be updated according to the following cases, and then we rewrite its sequence number based on the offset:

1. If current frame equals max. frame, don't update offset;
2. If current frame succeeds max. frame, add to the offset the best estimate of the no. of suppressed packets in between;
3. If current frame precedes max. frame (late reordered packet), rewrite using offset in history if found, else drop.

Accuracy vs. Memory Efficiency. If the loss (and reordering) rate is low, a short history of offsets suffices; otherwise, we need a longer history. Thus, there is a trade-off between accuracy (i.e., unnecessary retxs.) and per-stream memory, with the optimal balance dictated by the loss rate. We implement several variants of our heuristic to explore the design space. Below, we describe two:

Seq. Rewriting-Low Memory (S-LM) stores no history of frames beyond the max.

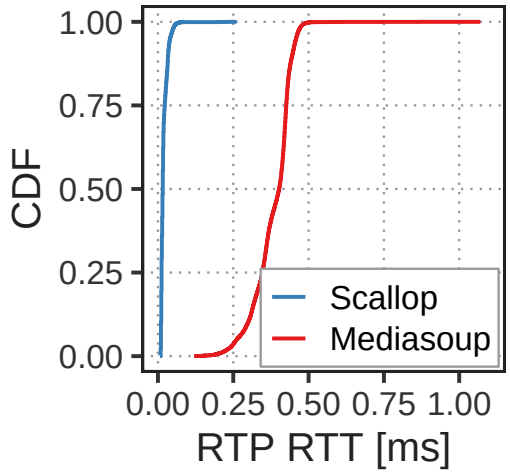


Figure 6.15: Forwarding latency in *Scallop*.

frame, minimizing memory usage.

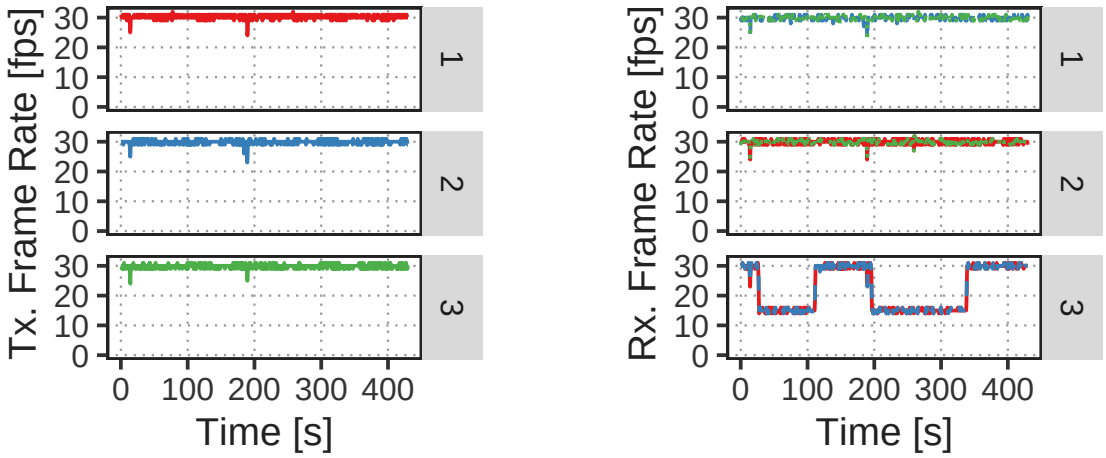
Seq. Rewriting-Low Retx. (S-LR) stores the history of three additional frames to accommodate 150 ms of suppressed (15 fps) video, the max. recommended RTC end-to-end delay [70]. This reduces unnecessary retxs. under higher loss rates.

6.8 Evaluation

Experimental Setup. We deploy the data plane of *Scallop* on a 12.8 Tbit/s Intel Tofino2 hardware switch. The switch agent runs on the CPU of this switch, an 8-core Intel Pentium with 8GB of RAM. The controller is deployed on a 40-core Intel Xeon server with 96 GB memory. We use another server with the same configuration for both *Scallop* and MediaSoup (where applicable) clients.

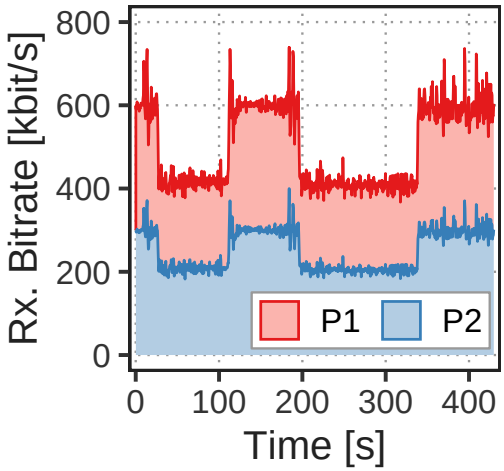
6.8.1 Control Plane

We first analyze the amount of packets and bytes that our controller needs to process compared with the amount of packets that stay entirely in the data plane. This ratio demonstrates the feasibility of *Scallop*'s control/data-plane split. We collect a packet-level trace of a real three-party meeting using *Scallop* where participants send audio and 720p AV1 SVC



(a) Send frame rate

(b) Receive frame rate



(c) Receive bit rate at P3

Figure 6.16: *Scallop* rate adaptation: Participant 3's receive bit rate is reduced twice.

video. We then determine the number of packets and bytes that can be processed in the data plane. The experiment ran for ten minutes with a total of 180,718 packets.

Table 6.1 shows the results. 94.5% of these packets were RTP packets which can be handled in the data plane with the exception of five RTP packets containing an AV1 dependency descriptor. RTCP accounted for 5.06% of all packets and 0.48% of all bytes, out of which our switch agent uses RTCP receiver reports and REMB messages to control the RA logic. These packets accounted for 3.41% of overall packets. Finally, STUN packets accounted for 0.38% of all packets and also need to be processed in software. In summary, 96.46% of all packets and 99.65% of all bytes can be processed in the data plane, showing that this work-

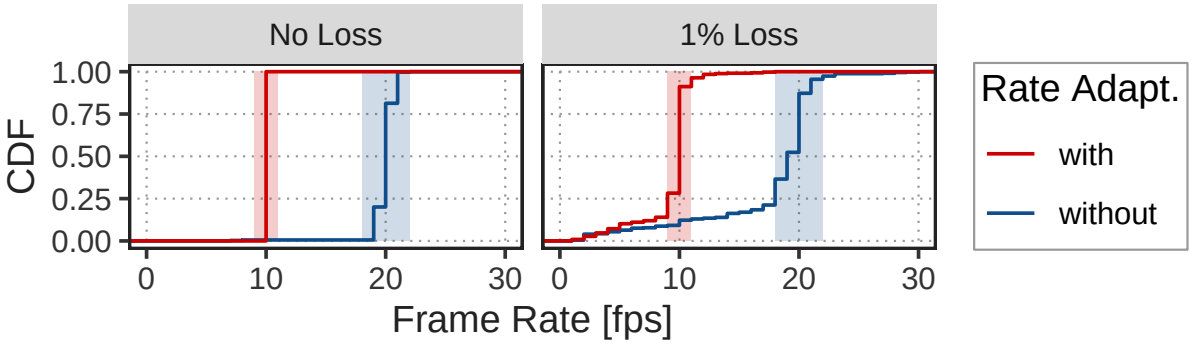


Figure 6.17: Frame rate under loss and rate adaptation.

Proto./Type	Packets	Pct.	Per sec.	KBytes	Pct.
RTP	170,870	94.5	284.30	166,762	99.47
- Audio	29,746	16.46	49.49	3,826	2.28
- Video	141,124	78.09	234.81	162,935	97.19
- AV1 DS*	5		0.008	6	
RTCP	9,153	5.06	15.22	801	0.48
- SR/SDES	3,456	1.91	5.75	304	0.18
- RR*	240	0.39	0.13	15	0.01
- RR/REMB*	5,457	3.02	9.07	482	0.29
STUN*	695	0.38	1.15	89	0.05
Ctrl. Plane	6397	**3.54**	10.64	593	**0.35**
Data Plane	174,326	**96.46**	290.06	167,066	**99.65**
Total	180,718	100	300.69	167,653	100

Table 6.1: Packets per participant sent to the SFU (10 min.)

load is well-suited for a control/data-plane split. More importantly, the remaining packets, except for a few STUN packets at the start of a session, are not blocking.

6.8.2 Data Plane

Tofino Resource Utilization. We implement *Scallop*'s data plane using ∼2000 lines of $P4_{16}$ code on the Tofino2. Table 6.2 summarizes the resource utilization of the Tofino2. We categorize resource types by how they scale with the number of concurrent participants supported by *Scallop*—constant, linear, or quadratic. For the non-constant types, we report the average utilization under peak campus traffic load, as observed in our campus dataset. The high ingress parsing depth is evidence of the deep, flexible parsing required to extract the SVC quality layer information. The number of stages consumed by *Scallop* falls squarely within the bounds of the maximum available on the Tofino2. For the rest of the components

Resource type	**Scaling behavior**	**Usage under peak campus load (avg.)**
Parsing depth	Constant	Ing. 27, Eg. 7
No. of stages	Constant	Ing. 7, Eg. 5
PHV containers	Constant	17.9%
Exact xbars	Constant	5.66%
Ternary xbars	Constant	2.52%
Hash bits	Constant	4.62%
Hash dist. units	Constant	6.94%
VLIW instr.	Constant	7.29%
Logical table ID	Constant	21.87%
SRAM	Linear	6.77%
TCAM	Linear	1.38%
Egress Tput.	Quadratic	1.2 Gb/s

Table 6.2: Resource usage of the Tofino2 hardware prototype.

with constant scaling behavior, we report the average utilization across all stages. This analysis shows that *Scallop*'s resource usage is low enough such that other network applications can be supported simultaneously on the switch.

6.8.3 Latency and Impact on Session Quality

Latency. We demonstrate that leveraging a hardware-based data plane significantly reduces SFU-induced delay. This is shown by comparing per-packet RTTs of RTP media packets in a two-party call. Two participants are connected either through *Scallop*'s Tofino or the Mediasoup's SFU server. Figure 6.15 shows that *Scallop* achieves 26.8× lower median latency while cutting 99%ile latency by 8.5×.

Session Quality during Rate Adaptation. To show that *Scallop* is faithful to the core SFU functionality, we validate that our SVC-based rate adaptation effectively reduces bit rate without causing freezes or other QoE degradation. We conduct two experiments. First, we start a three-party call, where all participants send and receive video. We collect WebRTC performance statistics [168] from Google Chrome, including receive frame rate, stall time, video resolution, and more; they correspond to the media stream after decoding and, thus, are an accurate representation of actual playback quality. Figure 6.16 confirms that *Scallop*

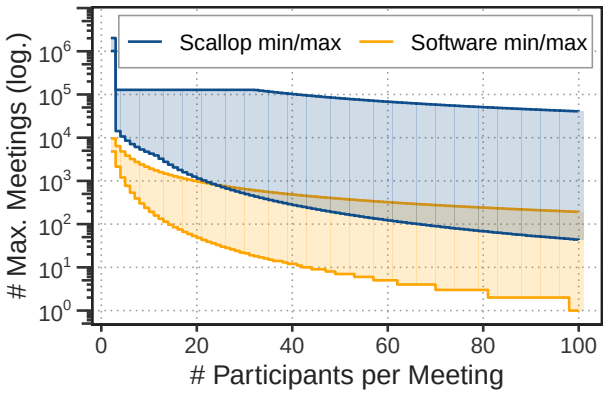


Figure 6.18: Best-case and worst-case performance.

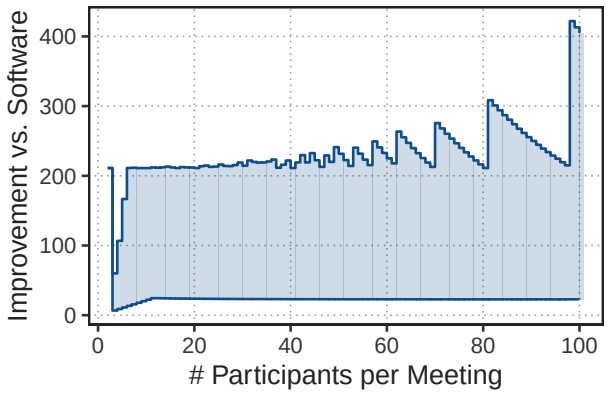


Figure 6.19: *Scallop* scalability gain over software.

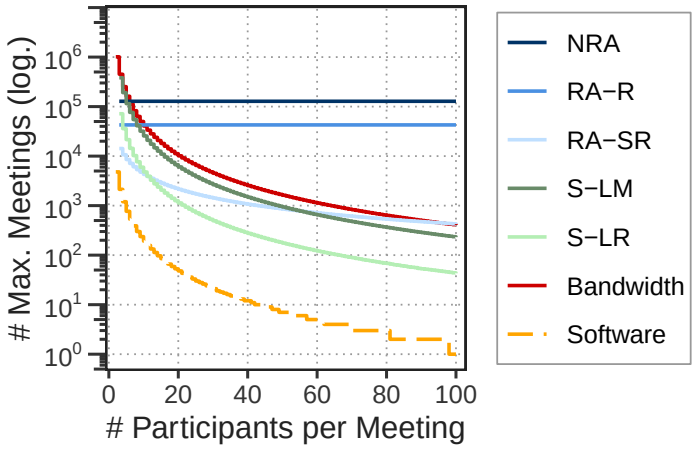


Figure 6.20: Compared to software, worst-case performance of *Scallop* by meeting configuration and implementation choice.

successfully reduces the frame rate from 30 to 15 fps for the bandwidth-constrained participant 3 while maintaining a decodable media stream without incurring otherwise lower QoE (e.g., via freezes). Second, we conduct four 5-minute experiments covering all combinations of 1% packet loss and rate adaptation (RA). Figure 6.17 shows that under loss, even without rate adaptation (and sequence-number rewriting), we see many samples of low frame rates (i.e., short freezes). This is consistent with prior observations and ITU recommendations [55, 69]. Scallop's RA does not change the shape of the distribution but moves the median frame rate, here, from 20 fps to 10 fps as intended. In fact, the fraction of fps readings that fall out of a 10% margin of the target frame rate (shaded areas) is slightly

lower under RA (14% vs. 21%), showing that *Scallop*'s RA does not further reduce QoE, even under loss.

6.8.4 Scalability

First, we perform a faithful simulation of *Scallop*'s data-plane components and subject them to a wide range of meeting configurations (described below) and loss rates. Second, we record MediaSoup's performance across experiments on varying workloads (similar to Section 6.2.2) and extrapolate it to the same configurations on a 32-core server. Finally, for each configuration, we compare the two systems.

Best-case vs. Worst-case Performance. We exhaustively simulate all possible combinations of loss rates (0 to 5% in 0.1% increments), number of participants (N) per meeting (1 to 100), number of senders per meeting (1 to N), the RA status per sender-receiver pair and its associated media quality level (low@1, medium@1.5, and high@3 Mbps, determined from experiments), replication-tree designs (NRA, RA-R, RA-SR), and the sequence-number rewriting heuristic used (S-LM vs. S-LR). For each configuration, we compute the min. and max. number of meetings supported by *Scallop* and MediaSoup (Software), respectively. In Figure 6.18, we show the range between Software/min and Software/max in orange, and that between *Scallop*/min and *Scallop*/max in blue. For any given N (x-axis), *Scallop*/min is always higher than Software/min and *Scallop*/max is always higher than Software/max. This shows that *Scallop* always supports many more meetings than software irrespective of the configuration. For N=2, *Scallop*/max is massive since the PRE is not needed. Thereafter, the numbers are determined by the complex interplay among the simulation configurations. For example, with N=3, without RA, *Scallop* supports 3× more meetings with just one sender vs. three senders, since the bandwidth used is only 9 Mbps in the former case (3 streams × 3 Mbps) but 27 Mbps (9 streams × 3 Mbps) in the latter. However, if RA results in low-quality media in the latter case, the bandwidth reduces to 9 Mbps. We observe that in most cases, the *Scallop*/min corresponds to the memory bottleneck

encountered when S-LR is used.

Scalability Improvement over Software. For each configuration, we compute the ratio between the number of meetings supported by *Scallop* and Mediasoup. Figure 6.19 shows the lowest and highest such ratio per N with the intermediate range shaded. The range shows that *Scallop* can support between 7-422× more meetings than software. Toward the right of the plot, the highest ratio shows a sawtooth pattern because the denominator is small yet discrete.

Software vs. *Scallop* Bottlenecks. The different *Scallop* designs (NRA/RA-R/RA-SR and S-LM/S-LR) have varying trade-offs. Figure 6.20 shows Scallop's performance assuming the respective design was the bottleneck with all participants sending media. For example, NRA supports more meetings than RA-R, but both are constant, whereas RA-SR supports fewer meetings and decreases with N. The overall performance corresponds to the minimum of all the designs. We plot the performance of MediaSoup (orange) for comparison.

6.9 Discussion

6.9.1 Making WebRTC Hardware-Amenable

WebRTC is a widely adopted framework that enables browsers to establish secure and efficient real-time multimedia sessions, even across NATs and firewalls. Originally designed for peer-to-peer communication between two endpoints, its limitations become apparent in multi-party use cases where SFUs become required (Section 6.2). These limitations are associated with the way WebRTC enables secure communication and, secondly, with the design of its wire protocols. We explain both aspects next before making recommendations on how to modify WebRTC for hardware-friendliness.

Encryption and message authentication. In WebRTC, RTP headers are HMAC-protected while payloads are also AES-encrypted via SRTP-DTLS [119]. In *Scallop*, neither

mechanism is currently implemented. The main challenge is key distribution: WebRTC's split-proxy design uses a separate key for each P2P connection, exchanged between SFU and client. This is incompatible with our proxy redesign, where a single sender's packets are replicated to many receivers, necessitating one-to-many key distribution, for example, via centrally distributed keys (as previously done in WebRTC via SDES [37] and done today in Zoom [65]). Since the SFU does not need to touch payloads, it would then operate on encrypted packets. Unlike in WebRTC's current encryption mechanism and key-distribution scheme, doing so would also enable the use of end-to-end encryption (E2EE). Prior work demonstrates that ciphers (e.g., AES) and cryptographic hashes (e.g., SipHash) can be computed on programmable data-plane devices [31, 120, 133, 135, 169, 179], and modern SmartNICs provide even more capabilities [80, 83, 157]. Rewriting header fields requires recomputing HMACs over the short RTP header, which is feasible in programmable hardware.

Hardware-amenable wire protocols. The WebRTC framework and its protocols, which most video-conferencing applications follow, are not hardware-friendly. This is because the protocol was initially designed as a P2P protocol, where software with complex algorithms is designed to run on the end hosts. Furthermore, codec and protocol designers emphasize efficiency, making every bit count and aiming to minimize the bandwidth usage between participants. This is a noble goal, but it makes the protocol inherently challenging to parse and process in hardware (e.g., network devices).

WebRTC design considerations for scalability. Designing SFUs as true proxies for WebRTC adds significant overheads (Section 6.3), while a more efficient and hardware-friendly use, as demonstrated in this chapter, comes with some limitations outlined above or requires some added complexity in managing feedback messages (Section 6.5.3) or realizing rate adaptation (Section 6.7). We find that none of these limitations are fundamental to WebRTC and argue that WebRTC already provides the necessary building blocks to support

more hardware-amenable, lightweight SFU designs. With minor modifications WebRTC can support SFUs that are better suited for hardware offload:

- As outlined above, centralized key distribution would free the SFU from having to decrypt and re-encrypt packets, while still allowing for standard encryption and message authentication mechanisms. If end-to-end encryption of payloads is desired, a double encryption scheme with keys directly exchanged among clients can be used [167, 194]. This approach would allow the SFU to operate on encrypted payloads such that the SFU does not need to be trusted with payloads, which can be desirable.
- If RTCP did not use packets that combine reports across multiple streams, the SFU could process and forward RTCP packets in hardware more easily without needing to parse deep into packets or splitting packets in the data plane.
- If the SFU could indicate via the AV1 dependency descriptor which decode target should currently be active for a given stream, the receiver could ignore all packets (even in the presence of sequence skips) not part of the current decode target. This would eliminate the need for sequence rewriting.
- Compression and dynamic variable-length fields can be difficult to handle in hardware. Following the trend of hardware offloading and co-designing software and hardware, it is worth re-exploring the design and implementation of more hardware-amenable protocols (e.g., for AV1 and RTCP), shifting the balance towards offloading computation and less on bandwidth optimization. In fact, the community has seen ideas in the past, for example using fixed-length header fields in BGP routing [82].

6.9.2 Scallop for Commercial Deployment

Extended VCA Use Cases. Video-conferencing applications often implement additional features that need direct media access; these include live transcription and visual effects (e.g., virtual backgrounds or "funny hats") [166]. In most cases, including Zoom, such

functions are implemented at SFUs, and this requires decryption of client's media stream at the SFU. Because *Scallop* operates on encrypted and encoded media, these features cannot be implemented in *Scallop*'s data plane. We argue, however, that these features should be implemented at clients, not SFUs, to preserve end-to-end encryption. WebRTC provides APIs for this [39, 167].

SFU cascading. SFU cascading is a technique of deploying SFUs in a hierarchical manner to improve the scalability of VCA infrastructure by aggregation of media streams at higher-level SFUs. We argue our system is not an alternative to or a competing solution with SFU cascading. In fact, the approaches are independent and could be combined to improve the scalability further. Our control/data plane split has the potential to simplify deploying many SFU data planes under the management of a single controller. Our current system is already designed in this way and would provide the architectural framework to enable such SFU topologies.

6.10 Related Work

Studies on Video-Conferencing Applications. Baset and Schulzrinne's analysis of Skype [11] provided first insights about RTC systems. In addition to earlier studies on VCAs [118, 86], recent papers conducted extensive QoE-centric measurement studies of different VCAs such as Zoom, Meet, Teams, and WebEx [29, 93]. In the context of this work, these studies shed light on the infrastructure, geographic location, latency, bit rate and network utilization, and their impact on QoE. Choi et al. did a more longitudinal analysis of Zoom [34] while Michel et al. did an in-depth study of Zoom in a production network, demystifying Zoom's packet format [103].

Handling Video-Conferencing Traffic in the Data Plane. Edwards and Ciarleglio showcased a programmable data plane that can perform "clean" video switching of uncompressed video flows based on RTP timestamps [49]. This demo solely focuses on showing the

capability of parsing RTP headers and making forwarding decisions based on RTP timestamp values, rather than offloading any SFU functionality. Other work showed that programmable data planes and eBPF/XDP programs running on servers can help with the NAT traversal functionality in VCAs [77, 88]. Our work goes way beyond this and enables the data plane to also perform packet replication and selective forwarding of actual RTP media traffic. The perhaps closest work [161] builds an SFU in software (bmv2) using P4 but does not actually implement a functional VCA. It generates dummy packets with port numbers distinguishing media layers and the design ignores all challenges related to feedback.

6.11 Conclusion

Taken together, our SDN-inspired SFU-switch design is driven by the key insight that most SFU tasks are, in fact, replicating and dropping media packets. Unlike traditional infrastructure where an SFU server takes care of everything, our prototype comprises an efficient programmable data plane that processes media packets at line rate and a software control plane that handles infrequent tasks such as signaling, quality monitoring, and rate adaptation. Our prototype is built with a real programmable switch (Intel Tofino2 [3]) and delivers 7–422× improved scalability over a 32-core SFU server.

Chapter 7

Conclusion

Modern Internet applications place demanding requirements on the network. Users expect responsiveness, reliability, security, and scalability. Yet application outcomes are shaped by events inside the network and by hop-by-hop decisions made under strict speed and resource constraints. This dissertation argues that the gap between what applications need and what networks natively expose can be narrowed by treating the network as an active participant in delivering outcomes, enabled by high-speed programmable packet processors and careful hardware-aware design.

Networks can contribute along two complementary axes. First, they can deeply observe traffic by extracting signals that matter for application performance, security, and scalability—continuously and at production speeds. Second, they can intervene meaningfully by acting on those signals quickly and in a way that remains compatible with real deployments and real protocol behavior. Together, these capabilities provide a practical foundation for *network-boosted applications*.

7.1 Summary of Contributions

This dissertation applies the lens of network-boosted applications to the two most prevalent classes of Internet traffic—on-demand and interactive.

On-demand traffic. This dissertation develops mechanisms to continuously extract actionable signals, namely RTT, from live traffic through passive measurement at line rate, despite the challenges posed by real protocol behavior and device constraints (Chapter 3). It then shows how these signals can drive fast, in-network defenses against long-distance routing attacks, reducing how long user traffic is exposed and limiting harm (Chapter 4).

Interactive traffic. This dissertation characterizes the structure and performance drivers of real-world video conferencing traffic from a popular application, namely Zoom, from a network vantage point, establishing what can be inferred reliably from packet traces (Chapter 5). Building on these insights, it shows how to push high-volume, latency-sensitive packet-processing work into programmable hardware while retaining a lean software control plane for slower, more complex decisions, substantially improving scalability without violating tight latency and jitter requirements (Chapter 6).

7.2 Future Directions

I plan to extend the concepts presented throughout this dissertation into new directions that broaden the scope of network-boosted applications.

Comprehensive closed-loop control for emerging applications. A natural next step is to extend closed-loop, application-aware control to emerging domains such as AR/VR and AI-driven applications, including chatbots like ChatGPT, voice assistants, and coding copilots. These applications have traffic patterns, performance bottlenecks, and failure modes that remain poorly understood. Addressing them will require new forms of deep observability and fast intervention tailored to their needs, similar to what this dissertation develops for on-demand and interactive media traffic.

Comprehensive closed-loop control across protocol layers. A second direction is to broaden closed-loop control across protocol layers, expanding visibility downward into access networks such as cellular and Wi-Fi [177, 178]. The goal is to build end-to-end control loops that reason jointly across these layers, rather than treating each layer in isolation, so that the network can better diagnose problems and apply the right intervention at the right layer.

Cross-platform realization on heterogeneous programmable infrastructure. A third direction is to realize network functionality across devices with very different performance and programmability profiles, including switches, SmartNICs, FPGAs, programmable hosts, and servers. Fast but resource-constrained devices can handle simple, high-volume tasks, while more flexible platforms can perform richer stateful analysis and precise enforcement. The key challenge is to make this decomposition principled and automatic: given a high-level objective and a collection of available platforms, the system should generate a layered design that respects each platform's constraints while achieving the desired functionality and robustness.

AI for network control with safety, interpretability, and generalization. The final direction is to bring AI into the network control loop in a way that operators can trust. Programmable packet processors create an opportunity to generate richer features at line rate, across diverse traffic and network conditions, which can improve both training, testing, and inference of learning-based controllers. At the same time, safe deployment requires explicit safeguards. The system must continuously verify whether an AI-driven action is improving metrics such as latency, loss, or utilization, and if those metrics degrade beyond predefined limits, it must automatically return to a known, safe rule-based control policy. More broadly, the controller's actions should be bounded by mechanisms in the network itself, so that no single bad decision can cause large or persistent harm. Making this practical will require integrating learning tightly with continuous measurement and safety constraints, rather than treating AI as an opaque decision module.

7.3 Final Remarks

This dissertation's central message is that networks no longer need to be treated as passive conduits that forward packets on a best-effort basis. With modern programmable data planes, the network can observe and act at the timescales where many problems originate, complementing endpoints and end-to-end protocols. The long-term opportunity is to turn this capability into a reliable, repeatable practice, one that supports new application domains, scales across heterogeneous infrastructure, and preserves the performance, safety, and robustness required by production networks.

Bibliography

[1] NVIDIA Mellanox NIC's performance report with DPDK 21.05. *http://fast.dpdk.org/doc/perf/DPDK_21_05_Mellanox_NIC_performance_report.pdf*, 2021.

[2] Aftab Siddiqui. Public DNS in Taiwan the latest victim to BGP hijack. `https://manrs.org/2019/05/public-dns-in-taiwan-the-latest-victim-to-bgp-hijack/`.

[3] Anurag Agrawal and Changhoon Kim. Intel Tofino2: A 12.9 Tbps P4-programmable Ethernet switch. In *IEEE Hot Chips Symposium (HCS)*, pages 1–32, New York, NY, USA, 2020. IEEE Computer Society, IEEE.

[4] Aditya Akella, Jeffrey Pang, Bruce Maggs, Srinivasan Seshan, and Anees Shaikh. A comparison of overlay routing and multihoming route control. *ACM SIGCOMM Computer Communication Review*, 34(4):93–106, 2004.

[5] Andree Toonk. How hacking team helped Italian special operations group with routing hijack. `https://bgpmon.net/how-hacking-team-helped-italian-special-operations-group-with-bgp-routing-hijack/`.

[6] Maria Apostolaki, Ankit Singla, and Laurent Vanbever. Performance-driven Internet path selection. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 41–53, New York, NY, USA, 2021. ACM.

[7] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 93–109, 2020.

[8] Axel Arnbak and Sharon Goldberg. Loopholes for circumventing the constitution: Unrestricted bulk surveillance on americans by collecting network traffic abroad. *Michigan Telecommunications and Technology Law Review*, 21:317, 2014.

[9] Axel Arnbak and Sharon Goldberg. Loopholes for circumventing the constitution: Unrestrained bulk surveillance on americans by collecting network traffic abroad. *Michigan Telecommunications and Technology Law Review*, Jan 2015.

[10] Rob Austein, Steven Bellovin, Russ Housley, Stephen Kent, Warren Kumari, Doug Montgomery, Chris Morrow, Sandy Murphy, Keyur Patel, John Scudder, Samuel Weiler, Matthew Lepinski, and Kotikalapudi Sriram. BGPsec protocol specification. RFC 8205, IETF, 2017.

[11] Salman A. Baset and Henning G. Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. In *IEEE INFOCOM 2006*, pages 1–11, New York, NY, USA, 2006. IEEE.

[12] $83k in Bitcoins 'stolen' through BGP hijack. `https://www.virusbulletin.com/blog/2014/08/83k-bitcoins-stolen-through-bgp-hijack/`.

[13] Debopam Bhattacherjee, Muhammad Tirmazi, and Ankit Singla. A cloud-based content gathering network. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2017.

[14] Henry Birge-Lee, Maria Apostolaki, and Jennifer Rexford. Global BGP attacks that evade route monitoring. In *Passive and Active Measurement Conference*, pages 335–357. Springer, 2025.

[15] Henry Birge-Lee, Yixin Sun, Anne Edmundson, Jennifer Rexford, and Prateek Mittal. Bamboozling certificate authorities with BGP. In *USENIX Security Symposium*, pages 833–849, 2018.

[16] Henry Birge-Lee, Liang Wang, Jennifer Rexford, and Prateek Mittal. Sico: Surgical interception attacks by manipulating BGP communities. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 431–448, 2019.

[17] Niklas Blum, Serge Lachapelle, and Harald Alvestrand. WebRTC: Real-time communication for the open web platform. *Communications of the ACM*, 64(8):50–54, 2021.

[18] Dario Bonfiglio, Marco Mellia, Michela Meo, Dario Rossi, and Paolo Tofanelli. Revealing Skype traffic: When randomness plays with you. In *ACM SIGCOMM*, pages 37–48, New York, NY, USA, 2007. ACM.

[19] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. TCP extensions for high performance. RFC 7323, RFC Editor, Sep 2014.

[20] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[21] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. In *ACM SIGMETRICS*, pages 1–25, 2019.

[22] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. *ACM SIGMETRICS Performance Evaluation Review*, 48(1):27–28, 2020.

[23] Mihai Budiu and Chris Dodd. The p416 programming language. *ACM SIGOPS Operating Systems Review*, 51(1):5–14, 2017.

[24] Tobias Bühler, Alexandros Milolidakis, Romain Jacob, Marco Chiesa, Stefano Vissicchio, and Laurent Vanbever. Oscilloscope: Detecting BGP hijacks in the data plane. *arXiv:2301.12843*, 2023.

[25] R. Bush and R. Austein. The resource public key infrastructure (RPKI) to router protocol. RFC 6810, IETF, 2013.

[26] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM Conference on Computer and Communications Security*, pages 317–329, New York, NY, USA, 2007. ACM.

[27] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Congestion control for web real-time communication. *IEEE/ACM Transactions on Networking*, 25(5):2629–2642, 2017.

[28] Celer Bridge incident analysis. `https://www.coinbase.com/blog/celer-bridge-incident-analysis`.

[29] Hyunseok Chang, Matteo Varvello, Fang Hao, and Sarit Mukherjee. Can you see me now? A measurement study of Zoom, Webex, and Meet. In *ACM Internet Measurement Conference*, pages 216–228, New York, NY, USA, 2021. ACM.

[30] Kuan-Ta Chen, Yu-Chun Chang, Po-Han Tseng, Chun-Ying Huang, and Chin-Laung Lei. Measuring the latency of cloud gaming systems. In *Proceedings of the 19th ACM International Conference on Multimedia*, pages 1269–1272, 2011.

[31] Xiaoqi Chen. Implementing AES encryption on programmable switches via scrambled lookup tables. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*, pages 8–14, 2020.

[32] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. Measuring TCP round-trip time in the data plane. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*, pages 35–41, New York, NY, USA, Aug 2020. ACM.

[33] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. Fast reroute on programmable switches. *IEEE/ACM Transactions on Networking*, 29(2):637–650, 2021.

[34] Albert Choi, Mehdi Karamollahi, Carey Williamson, and Martin Arlitt. Zoom session quality: A network-level view. In *Passive and Active Measurement Conference*, pages 555–572, Berlin, Germany, 2022. Springer.

[35] Catalin Cimpanu. KlaySwap crypto users lose funds after BGP hijack. `https://therecord.media/klayswap-crypto-users-lose-funds-after-bgp-hijack/`.

[36] David D Clark. The end-to-end argument and application design: The role of trust. *Fed. Comm. LJ*, 63:357, 2010.

[37] NTT Communications. A study of WebRTC security, 2015. Retrieved January 28, 2025, from `https://webrtc-security.github.io`.

[38] Paolo Comparetti, Gilbert Wondracek, Christopher Krügel, and Engin Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, pages 110–125, New York, NY, USA, May 2009. IEEE.

[39] The World Wide Web Consortium. W3C editor's draft: WebRTC encoded transform, 2025. Retrieved July 24, 2025, from `https://w3c.github.io/webrtc-encoded-transform`.

[40] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, USA, 2007. USENIX Association.

[41] Alexandre da Silveira Ilha, Ângelo Cardoso Lapolli, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. Euclid: A fully in-network, P4-based approach for real-time DDoS attack detection and mitigation. *IEEE Transactions on Network and Service Management*, 18(3):3121–3139, 2020.

[42] Natural Earth Data. Natural Earth data-free vector and raster map data. *`http://www.naturalearthdata.com` (accessed 10.12.2024)*, 2011.

[43] Luca De Cicco, Gaetano Carlucci, and Saverio Mascolo. Congestion control for WebRTC: Standardization status and open issues. *IEEE Communications Standards Magazine*, 1(2):22–27, 2017.

[44] Peter de Rivaz and Jack Haughton. AV1 bitstream and decoding process specification, 2019. Retrieved July 24, 2025, from `https://aomediacodec.github.io/av1-spec/av1-spec.pdf`.

[45] Xinhao Deng, Qi Li, and Ke Xu. Robust and reliable early-stage website fingerprinting attacks via spatial-temporal distribution analysis. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1997–2011, 2024.

[46] Yunhua Deng, Yusen Li, Xueyan Tang, and Wentong Cai. Server allocation for multiplayer cloud gaming. In *Proceedings of the 24th ACM International Conference on Multimedia*, pages 918–927, 2016.

[47] Hao Ding and Michael Rabinovich. TCP stretch acknowledgements and timestamps: Findings and implications for passive RTT measurement. *ACM SIGCOMM Computer Communication Review*, 45(3):20–27, 2015.

[48] Jon Dugan, Seth Elliott, Bruce A Mah, Jeff Poskanzer, and Kaustubh Prabhu. iperf3: Tool for active measurements of the maximum achievable bandwidth on IP networks, 2014. `https://github.com/esnet/iperf`.

[49] Thomas G. Edwards and Nick Ciarleglio. Timestamp-aware RTP video switching using programmable data plane, 2017. ACM SIGCOMM '17 Industrial Demos, Retrieved April 12, 2023, from `https://conferences.sigcomm.org/sigcomm/2017/files/program-industrial-demos/sigcomm17industrialdemos-paper2.pdf`.

[50] Mathis Engelbart and Jörg Ott. Congestion control for real-time media over QUIC. In *Workshop on Evolution, Performance and Interoperability of QUIC*, EPIQ '21, pages 1–7, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Augusto Espin and Christian Rojas. The impact of the COVID-19 pandemic on the use of remote meeting technologies. *SSRN Electronic Journal*, Jan 2021.

[52] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. The lockdown effect: Implications of the COVID-19 pandemic on Internet traffic. In *ACM Internet Measurement Conference*, pages 1–18, New York, NY, USA, 2020. ACM.

[53] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, page 13, USA, 2005. USENIX Association.

[54] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 267–282, USA, 2018. USENIX Association.

[55] Boni García, Micael Gallego, Francisco Gortázar, and Antonia Bertolino. Understanding and estimating quality of experience in WebRTC applications. *Computing*, 101(11):1585–1607, Nov 2019.

[56] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of TCP. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 61–74. ACM, 2017.

[57] Sharon Goldberg. Why is it taking so long to secure Internet routing? *Communications of the ACM*, 57(10):56–63, 2014.

[58] Dan Goodin. Strange snafu misroutes domestic US Internet traffic through China Telecom. *Ars Technica*, 6, 2018.

[59] Dan Goodin and Andre Smack. Russian-controlled telecom hijacks financial services' Internet traffic. https://arstechnica.com/information-technology/2017/04/russian-controlled-telecom-hijacks-financial-services-internet-traffic/, Apr 2017.

[60] Alex Gouaillard. Breaking point: WebRTC SFU load testing, 2018. Retrieved July 24, 2025, from https://webrtchacks.com/sfu-load-testing/.

[61] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the ACM SIGCOMM Conference*, pages 357–371, 2018.

[62] Craig Gutterman, Katherine Guo, Sarthak Arora, Trey Gilliland, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, and Gil Zussman. Requet: Real-time QoE metric detection for encrypted YouTube traffic. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 16(2s):1–28, 2020.

[63] Jamie Hayes and George Danezis. *k*-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security Symposium*, pages 1187–1203, 2016.

[64] Rahul Hiran, Niklas Carlsson, and Nahid Shahmehri. Crowd-based detection of routing anomalies on the Internet. In *IEEE Conference on Communications and Network Security (CNS)*, pages 388–396. IEEE, 2015.

[65] Todd Hoff. A short on how Zoom works, 2020. Retrieved July 24, 2025, from http://highscalability.com/blog/2020/5/14/a-short-on-how-zoom-works.html.

[66] Thomas Holterbach, Thomas Alfroy, Amreesh Phokeer, Alberto Dainotti, and Cristel Pelsser. A system to detect forged-origin BGP hijacks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1751–1770, 2024.

[67] Nicholas Hopper, Eugene Y Vasserman, and Eric Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security*, 13(2):1–28, 2010.

[68] Intel Corp. Intel Tofino, 2022.

[69] ITU-T. End-user multimedia QoS categories. Recommendation G.1010, International Telecommunication Union, Geneva, Switzerland, Nov 2001.

[70] ITU-T. One-way transmission time. Recommendation I.371, International Telecommunication Union, Geneva, Switzerland, May 2003.

[71] Mukund Iyengar. WebRTC architecture basics: P2P, SFU, MCU, and hybrid approaches, 2021. Retrieved July 24, 2025, from https://medium.com/securemeeting/webrtc-architecture-basics-p2p-sfu-mcu-and-hybrid-approaches-6e7d77a46a66.

[72] Hao Jiang and Constantinos Dovrolis. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review*, 32(3):75–88, 2002.

[73] Sunghyun Jin, Serae Kim, Sangtae Ha, and Kyunghan Lee. End-to-end coordination of RAN and edge server for latency-critical inference serving over cellular networks. *Proceedings of the ACM on Networking*, 3(CoNEXT4):1–23, 2025.

[74] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, pages 121–136, 2017.

[75] Jitsi video bridge: Open-source video conferencing for developers, 2023. Retrieved July 24, 2025, from `https://jitsi.org/jitsi-videobridge/`.

[76] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal. RFC 8445, Jul 2018.

[77] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. Offloading media traffic to programmable data plane switches. In *IEEE International Conference on Communications (ICC)*, pages 1–7, 2020.

[78] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, Lawrence J Wobker, et al. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, volume 15, pages 1–2, 2015.

[79] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, pages 90–106, 2020.

[80] Duckwoo Kim, SeungEon Lee, and KyoungSoo Park. A case for SmartNIC-accelerated private communication. In *Asia-Pacific Workshop on Networking (APNet)*, pages 30–35, 2020.

[81] Hyojoon Kim and Arpit Gupta. ONTAS: Flexible and scalable online network traffic anonymization system. In *ACM SIGCOMM Workshop on Network Meets AI & ML*, pages 15–21, New York, NY, USA, 2019. ACM.

[82] Firat Kiyak, Brent Mochizuki, Eric Keller, and Matthew Caesar. Better by a hair: Hardware-amenable Internet routing. In *IEEE International Conference on Network Protocols*, pages 83–92, 2009.

[83] Shaguftha Zuveria Kottur, Krishna Kadiyala, Praveen Tammana, and Rinku Shah. Implementing ChaCha based crypto primitives on programmable SmartNICs. In *ACM SIGCOMM Workshop on Formal Foundations and Security of Programmable Network Infrastructures*, pages 15–23, 2022.

[84] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC transport protocol: Design and Internet-scale deployment. In *ACM SIGCOMM*, pages 183–196, 2017.

[85] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *USENIX Annual Technical Conference (ATC)*, pages 403–415, 2015.

[86] Insoo Lee, Jinsung Lee, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Demystifying commercial video conferencing applications. In *ACM International Conference on Multimedia*, pages 3583–3591, New York, NY, USA, 2021. ACM.

[87] Sanghwan Lee, Zhi-Li Zhang, and Srihari Nelakuditi. Exploiting AS hierarchy for scalable route selection in multi-homed stub networks. In *ACM Internet Measurement Conference*, pages 294–299, 2004.

[88] Tamás Lévai, Balázs Edvárd Kreith, and Gábor Rétvári. Supercharge WebRTC: Accelerate TURN services with eBPF/XDP. In *Workshop on EBPF and Kernel Extensions*, eBPF '23, pages 70–76, New York, NY, USA, 2023. Association for Computing Machinery.

[89] Dave Levin, Youndo Lee, Luke Valenta, Zhihao Li, Victoria Lai, Cristian Lumezanu, Neil Spring, and Bobby Bhattacharjee. Alibi routing. In *ACM SIGCOMM*, pages 611–624, Aug 2015.

[90] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems (APoCS)*, pages 31–44. SIAM, 2020.

[91] Luis López, Miguel París, Santiago Carot, Boni García, Micael Gallego, Francisco Gortázar, Raul Benítez, Jose A. Santos, David Fernández, Radu Tom Vlad, Iván Gracia, and Francisco Javier López. Kurento: The WebRTC modular media server. In *ACM International Conference on Multimedia*, MM '16, pages 1187–1191, New York, NY, USA, 2016. ACM.

[92] Justin Ma, Kirill Levchenko, Christian Kreibich, Stefan Savage, and Geoffrey M. Voelker. Unexpected means of protocol inference. In *ACM SIGCOMM Conference on Internet Measurement*, pages 313–326, New York, NY, USA, 2006. ACM.

[93] Kyle MacMillan, Tarun Mangla, James Saxon, and Nick Feamster. Measuring the performance and network utilization of popular video conferencing applications. In *ACM Internet Measurement Conference*, pages 229–244, New York, NY, USA, 2021. ACM.

[94] Andrew Mahr, Meghan Cichon, Sophia Mateo, Cinthya Grajeda, and Ibrahim Baggili. Zooming into the pandemic! a forensic analysis of the Zoom application. *Forensic Science International: Digital Investigation*, 36, 2021.

[95] Bill Marczak and John Scott-Railton. Move fast and roll your own crypto, 2020.

[96] Video conferencing market forecast & statistics, 2023. Retrieved July 24, 2025, from `https://www.marketsandmarkets.com/Market-Reports/video-conferencing-market-99384414.html`.

[97] Philip Matthews, Jonathan Rosenberg, Dan Wing, and Rohan Mahy. Session traversal utilities for NAT (STUN). RFC 5389, Oct 2008.

[98] LLC MaxMind. GeoIP, 2006.

[99] Nick McKeown. Software-defined networking. *INFOCOM keynote talk*, 17(2):30–32, 2009.

[100] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[101] Measurement Lab. The M-Lab NDT data set. `https://measurementlab.net/tests/ndt`, 2024. Bigquery table `measurement-lab.ndt.download` Data range: 2024-12-01 – 2024-12-05.

[102] MediaSoup: Cutting-edge WebRTC video conferencing, 2025. Retrieved July 24, 2025, from `https://mediasoup.org`.

[103] Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. Enabling passive measurement of Zoom performance in production networks. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, pages 244–260, New York, NY, USA, 2022. Association for Computing Machinery.

[104] Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. Zoom analysis source code, 2022.

[105] Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. Scalable video conferencing using SDN principles. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 1213–1231, 2025.

[106] Oliver Michel, Satadal Sengupta, Hyojoon Kim, Ravi Netravali, and Jennifer Rexford. Scalable video conferencing using SDN principles. *arXiv preprint arXiv:2503.11649*, 2025.

[107] Alexandros Milolidakis, Tobias Bühler, Kunyu Wang, Marco Chiesa, Laurent Vanbever, and Stefano Vissicchio. On the effectiveness of BGP hijackers that evade public route collectors. *IEEE Access*, 11:31092–31124, 2023.

[108] Abhijit Mondal, Satadal Sengupta, Bachu Rikith Reddy, MJV Koundinya, Chander Govindarajan, Pradipta De, Niloy Ganguly, and Sandip Chakraborty. Candid with YouTube: Adaptive streaming behavior and implications on data consumption. In

Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video, pages 19–24, 2017.

[109] Giovane CM Moura, John Heidemann, Wes Hardaker, Pithayuth Charnsethikul, Jeroen Bulten, Joao Ceron, and Cristian Hesselman. Old but gold: Prospecting TCP to engineer and real-time monitor DNS anycast. In *Passive and Active Measurement Conference*, pages 264–292, Berlin, Germany, Mar 2022. Springer.

[110] Mozilla. WebRTC API: Signaling and video calling, 2025. Retrieved January 28, 2025, from `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling`.

[111] Maurizio M. Munafo and Martino Trevisan. Zoom PCAP cleaner, 2020.

[112] John Nagle. RFC 0896: Congestion control in IP/TCP internetworks, 1984.

[113] John Narayan, Sandeep K. Shukla, and T. Charles Clancy. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys*, 48(3), Dec 2015.

[114] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the ACM SIGCOMM Conference*, pages 85–98, 2017.

[115] RIPE NCC. RIPE Atlas. `https://atlas.ripe.net/`, 2021.

[116] Kathleen Nichols. pping (Pollere passive ping). `https://github.com/pollere/pping`, 2017.

[117] Shaun Nichols. AWS DNS network hijack turns MyEtherWallet into ThievesEtherWallet. The Register, 2018.

[118] Antonio Nistico, Dena Markudova, Martino Trevisan, Michela Meo, and Giovanna Carofiglio. A comparative study of RTC applications. In *IEEE International Symposium on Multimedia*, pages 1–8, New York, NY, USA, 2020. IEEE.

[119] Karl Norrman, David McGrew, Mats Naslund, Elisabetta Carrara, and Mark Baugher. The secure real-time transport protocol (SRTP). RFC 3711, Mar 2004.

[120] Isaac Oliveira, Emídio Neto, Roger Immich, Ramon Fontes, Augusto Neto, Fabrício Rodriguez, and Christian Esteve Rothenberg. DH-AES-P4: On-premise encryption and in-band key-exchange in P4 fully programmable data planes. In *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 148–153. IEEE, 2021.

[121] Shawn Ostermann. Tcptrace homepage. *`http://www.tcptrace.org/`*, 2007.

[122] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[123] Colin Perkins, Mark J. Handley, and Van Jacobson. SDP: Session description protocol. RFC 4566, Jul 2006.

[124] Adrian Perrig, Pawel Szalachowski, Raphael M. Reischuk, and Laurent Chuat. *SCION: A secure Internet architecture*. Springer Verlag, 2017.

[125] Louis Poinsignon. BGP leaks and cryptocurrencies. *The Cloudflare Blog, April*, 2018.

[126] RIPE NCC. YouTube hijacking: A RIPE NCC RIS case study. `https://www.ripe.net/publications/news/youtube-hijacking-a-ripe-ncc-ris-case-study/`.

[127] Andrei Robachevsky. Routing security getting better, but no reason to rest!, 2019.

[128] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *IEEE International Conference on Cloud Computing*, pages 500–507, 2011.

[129] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[130] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *ACM SIGCOMM Computer Communication Review*, 29(5):71–78, 1999.

[131] Johann Schlamp, Ralph Holz, Quentin Jacquemart, Georg Carle, and Ernst W Biersack. HEAP: Reliable assessment of BGP hijacking attacks. *IEEE Journal on Selected Areas in Communications*, 34(6):1849–1861, 2016.

[132] Brandon Schlinker, Todd Arnold, Italo Cunha, and Ethan Katz-Bassett. PEERING: Virtualizing BGP at the edge for research. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 51–67, Orlando, FL, Dec 2019.

[133] Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmüller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle. Cryptographic hashing in P4 data planes. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6. IEEE, 2019.

[134] Eve Schooler, Jonathan Rosenberg, Henning Schulzrinne, Alan Johnston, Gonzalo Camarillo, Jon Peterson, Robert Sparks, and Mark J. Handley. SIP: Session initiation protocol. RFC 3261, Jul 2002.

[135] Lars-Christian Schulz, Robin Wehner, and David Hausheer. Cryptographic path validation for SCION in P4. In *European P4 Workshop (EuroP4)*, pages 17–23, 2023.

[136] Henning Schulzrinne and Stephen L. Casner. RTP profile for audio and video conferences with minimal control. RFC 3551, Jul 2003.

[137] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A transport protocol for real-time applications. RFC 3550, Jul 2003.

[138] Henning Schulzrinne and Jonathan Rosenberg. An offer/answer model with session description protocol (SDP). RFC 3264, Jul 2002.

[139] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, 2007.

[140] Satadal Sengupta, Niloy Ganguly, Pradipta De, and Sandip Chakraborty. Exploiting diversity in Android TLS implementations for mobile app traffic classification. In *The World Wide Web Conference*, pages 1657–1668, 2019.

[141] Satadal Sengupta, Hyojoon Kim, Daniel Jubas, Maria Apostolaki, and Jennifer Rexford. Data-plane telemetry to mitigate long-distance BGP hijacks. *arXiv preprint arXiv:2507.14842*, 2025.

[142] Satadal Sengupta, Hyojoon Kim, Daniel Jubas, Maria Apostolaki, and Jennifer Rexford. Passive data-plane telemetry to mitigate long-distance BGP hijacks. In *New Ideas in Networked Systems (NINeS)*, 2026.

[143] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Fine-grained RTT monitoring inside the network. *Measuring Network Quality for End-Users*, 2021.

[144] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. Continuous in-network round-trip time monitoring. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 473–485, New York, NY, USA, 2022. Association for Computing Machinery.

[145] Pavlos Sermpezis, Vasileios Kotronis, Petros Gigis, Xenofontas Dimitropoulos, Danilo Cicalese, Alistair King, and Alberto Dainotti. ARTEMIS: Neutralizing BGP hijacking within a minute. *IEEE/ACM Transactions on Networking*, 26(6):2471–2486, 2018.

[146] Taveesh Sharma, Tarun Mangla, Arpit Gupta, Junchen Jiang, and Nick Feamster. Estimating WebRTC video QoE metrics without using application headers. In *Proceedings of the 2023 ACM Internet Measurement Conference*, pages 485–500, 2023.

[147] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with fine-grained parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 87–102, 2022.

[148] Meng Shen, Yiting Liu, Siqi Chen, Liehuang Zhu, and Yuchao Zhang. Webpage fingerprinting using only packet length information. In *IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2019.

[149] Xingang Shi, Yang Xiang, Zhiliang Wang, Xia Yin, and Jianping Wu. Detecting prefix hijackings in the Internet with Argus. In *Proceedings of the 2012 ACM Internet Measurement Conference*, pages 15–28, 2012.

[150] Jean-Pierre Smith, Luca Dolfi, Prateek Mittal, and Adrian Perrig. QCSD: A QUIC client-side website-fingerprinting defence framework. In *USENIX Security Symposium*, pages 771–789, 2022.

[151] Jean-Pierre Smith, Prateek Mittal, and Adrian Perrig. Website fingerprinting in the age of QUIC. *Proceedings on Privacy Enhancing Technologies*, 2021(2):48–69, 2021.

[152] Thomas Stockhammer. Dynamic adaptive streaming over HTTP: Standards and design principles. In *Proceedings of the 2nd ACM Conference on Multimedia Systems*, pages 133–144, 2011.

[153] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. D2R: Policy-compliant fast reroute. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 148–161, 2021.

[154] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, 2012.

[155] Gary J. Sullivan and Thomas Wiegand. Video compression: From concepts to the H.264/AVC standard. *Proceedings of the IEEE*, 93(1):18–31, 2005.

[156] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. RAPTOR: Routing attacks on privacy in Tor. In *USENIX Security Symposium*, pages 271–286, 2015.

[157] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. sRDMA: Efficient NIC-based authentication and encryption for remote direct memory access. In *USENIX Annual Technical Conference (ATC)*, pages 691–704, 2020.

[158] IP Info Team. IP info, 2017.

[159] Vineeth Sagar Thapeta, Komal Shinde, Mojtaba Malekpourshahraki, Darius Grassi, Balajee Vamanan, and Brent E Stephens. Nimble: Scalable TCP-friendly programmable in-network rate-limiting. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 27–40, 2021.

[160] The Alliance for Open Media, AV1 Real-Time Communications Subgroup. RTP payload format for AV1, 2023. Retrieved October 5, 2023, from `https://aomediacodec.github.io/av1-rtp-spec/`.

[161] Pavlos Tsikrikas and George Xylomenos. A selective forwarding unit implementation in P4. In *2024 IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 181–186, 2024.

[162] Ismael Valenzuela and Douglas McKee. Hacking proprietary protocols with sharks and pandas, 2021.

[163] Jean-Marc Valin, Koen Vos, and Tim Terriberry. Definition of the Opus audio codec. RFC 6716, Sep 2012.

[164] Abhinau K. Venkataramanan, Chengyang Wu, Alan C. Bovik, Ioannis Katsavounidis, and Zafar Shahid. A hitchhiker's guide to structural similarity. *IEEE Access*, 9:28872–28896, 2021.

[165] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems*, New York, NY, USA, 2015. Association for Computing Machinery.

[166] W3C. WebRTC extended use cases, 2023. Retrieved January 28, 2025, from `https://www.w3.org/TR/webrtc-nv-use-cases`.

[167] W3C. MediaStreamTrack insertable media processing using streams, 2024. Retrieved January 28, 2025, from `https://w3c.github.io/mediacapture-transform`.

[168] W3C. Identifiers for WebRTC's statistics API, 2025. Retrieved July 24, 2025, from `https://www.w3.org/TR/webrtc-stats`.

[169] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. Raven: Stateless rapid IP address variation for enterprise networks. *Privacy Enhancing Technologies Symposium (PETS)*, 2023.

[170] Nikolas Wehner, Michael Seufert, Joshua Schuler, Sarah Wassermann, Pedro Casas, and Tobias Hossfeld. Improving web QoE monitoring for encrypted network traffic through time series modeling. *ACM SIGMETRICS Performance Evaluation Review*, 48(4):37–40, 2021.

[171] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[172] Wireshark. The Wireshark network protocol analyzer, 2022.

[173] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In *Network and Distributed System Security Symposium*, Reston, VA, USA, 2008. Internet Society.

[174] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Hongyi Liu, Matty Kadosh, Alan Lo, Aditya Akella, Thomas Anderson, Arvind Krishnamurthy, TS Eugene Ng, et al. A vision for runtime programmable networks. In *Proceedings of the 20th ACM Workshop on Hot Topics in Networks*, pages 91–98, 2021.

[175] Wenquan Xu, Zijian Zhang, Yong Feng, Haoyu Song, Zhikang Chen, Wenfei Wu, Guyue Liu, Yinchao Zhang, Shuxin Liu, Zerui Tian, et al. ClickINC: In-network computing as a service in heterogeneous programmable data-center networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 798–815, 2023.

[176] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. Using Trio: Juniper Networks' programmable chipset for emerging in-network applications. In *ACM SIGCOMM Conference*, pages 633–648, 2022.

[177] Fan Yi, Haoran Wan, Kyle Jamieson, and Oliver Michel. Automated, cross-layer root cause analysis of 5G video-conferencing quality degradation. In *Proceedings of the 2025 ACM Internet Measurement Conference*, pages 835–850, 2025.

[178] Fan Yi, Haoran Wan, Kyle Jamieson, Jennifer Rexford, Yaxiong Xie, and Oliver Michel. Athena: Seeing and mitigating wireless impact on video conferencing and beyond. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, pages 103–110, 2024.

[179] Sophia Yoo and Xiaoqi Chen. Secure keyed hashing on programmable switches. In *ACM SIGCOMM Workshop on Secure Programmable network INfrastructure*, pages 16–22, 2021.

[180] Sophia Yoo, Xiaoqi Chen, and Jennifer Rexford. SmartCookie: Blocking large-scale SYN floods with a split-proxy defense on programmable data planes. In *USENIX Security Symposium*, pages 217–234, 2024.

[181] Dan York. BGP hijacking in Iceland and Belarus shows increased need for BGP security. `https://www.internetsociety.org/blog/2014/02/bgp-hijacking-in-iceland-and-belarus-shows-increased-need-for-bgp-security/`, Feb 2014.

[182] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *ACM SIGCOMM*, pages 296–309, 2020.

[183] Minlan Yu, Lavanya Jose, and Rui Miao. Software-defined traffic measurement with OpenSketch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–42, 2013.

[184] Huanhuan Zhang, Anfu Zhou, Yuhan Hu, Chaoyue Li, Guangping Wang, Xinyu Zhang, Huadong Ma, Leilei Wu, Aiyun Chen, and Changhui Wu. Loki: Improving long tail performance of learning-based real-time video adaptation by fusing rule-based models. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 775–788, New York, NY, USA, 2021. Association for Computing Machinery.

[185] Zheng Zhang, Ying Zhang, Y Charlie Hu, Z Morley Mao, and Randy Bush. iSPY: Detecting IP prefix hijacking on my own. In *ACM SIGCOMM*, pages 327–338, 2008.

[186] Yufei Zheng, Xiaoqi Chen, Mark Braverman, and Jennifer Rexford. Unbiased delay measurement in the data plane. In *Symposium on Algorithmic Principles of Computer Systems (APoCS)*, pages 15–30. SIAM, 2022.

[187] Xiaoqing Zhu, Rong Pan, Michael A. Ramalho, and Sergio Mena de la Cruz. Network-assisted dynamic adaptation (NADA): A unified congestion control scheme for real-time media. RFC 8698, Feb 2020.

[188] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, pages 479–491, 2015.

[189] Zoom. Zoom account API, 2025. Retrieved January 27, 2025, from `https://developers.zoom.us/docs/api/rest/reference/account/methods/`.

[190] Zoom Video Communications, Inc. Connection process, 2020.

[191] Zoom Video Communications, Inc. Accessing meeting and phone statistics, 2021.

[192] Zoom Video Communications, Inc. How QoS metrics are determined in the Zoom API, 2021.

[193] Zoom Video Communications, Inc. Zoom encryption, 2021.

[194] Zoom Video Communications, Inc. End-to-end (E2EE) encryption for meetings, 2022.

[195] Zoom Video Communications, Inc. Zoom dashboard API, 2022.

[196] Zoom Video Communications, Inc. Zoom meeting SDKs, 2022.

[197] Zoom Video Communications, Inc. Zoom network firewall or proxy server settings, 2022.